实用程序设计理论

A Practical Theory of Programming

2018 年 6 月版

［加拿大］Eric C. R. Hehner 著

郭佳宁 万剑怡 郑宇华 译

实用程序设计理论

2018 年 6 月版

[加拿大] Eric C. R. Hehner 著

郭佳宁 万剑怡 郑宇华 译

电脑科学系

多伦多大学

地址 Toronto ON M5S 2E4 Canada

本书第一版中文翻译版由科学出版社出版，北京，2010，
ISBN 978-7-03-027425-0，www.sciencep.com

目前的版本可从以下链接免费获取
www.cs.utoronto.ca/~hehner/aPToP

关于本书的英文网络课程链接
www.cs.utoronto.ca/~hehner/FMSD

作者的个人网页
www.cs.utoronto.ca/~hehner

封面图片是一个因努伊特石堆，也就是石头堆成的拟人图形。因努伊特石堆存在于加拿大的整个北极地区。它们是因纽特人建造的，寓意是“你在正确的道路上。”。

目录

---目录结束

第零章 序 言

程序设计理论有什么用? 谁需要它呢? 成千上万的程序员在没有任何理论的情况下依然每天编程。那么又何必费心去学一门理论呢？该问题的答案其实和任何其他理论一样。譬如, 为什么人们要学习运动学理论? 人们可以在不懂运动学理论的情况下行动自如，也可以在不懂它的情况下投球。但是，在中学讲授运动学理论却是相当重要的。

对这个的问题的回答之一是，通过提供演算的方法，数学理论达到了一个高得多的精确程度。没有关于运动学的理论，几乎不可能发射火箭到木星。甚至投球手们也发现理论专家可以帮助他们改进投球技术。同样, 没有理论的帮助, 许多一般的程序设计也可以完成, 但对更复杂的程序设计, 没有好的理论就难以正确无误的完成。软件产业有数不清的因程序错误造成的惨痛教训无一不证明了这点。况且, 即使是一般的编程也会因适当运用理论而得到改进。

而回答之二是, 理论可以提高理解能力。当人们学会了运动学的理论时, 对运动的控制和预见能力从艺术走向了科学。同样, 当人们学会以理解数学定理的方式来理解程序时, 程序设计也从艺术走向了科学。有了科学的观点，就可以改变对世界的看法。更少地依赖灵感或运气，更明白什么是可能的，什么是不可能的。对任何人而言，这都是宝贵的教育。

专业工程学在社会上维持很高的声誉, 就是因为它坚持这样一点: 要成为一名专业工程师, 一个人必须懂得和应用相关理论。一个土木工程师必须懂得和运用几何和材料力学。一个电气工程师必须懂得和运用电磁理论。软件工程师要想名副其实，也必须懂得和运用程序设计理论。

本书的主题有时称为 “程序设计方法学”,有时又称为 “程序设计科学”, “程序设计逻辑”, “程序设计理论”, “程序开发的形式化方法”,或 “程序证明”。它涉及程序设计中经得起数学证明的那些方面。一个好的理论能帮助人们写出精确的规范，并设计出其执行可被证明满足规范的程序。本书将考虑计算的状态、计算的时间、计算所需的存储空间以及计算的交互。但有一些其他的软件设计和制作的重要方面本书没有论及：人员管理, 用户界面, 文档化和测试。

第一个有用的程序设计理论通常称为 “霍尔逻辑”，其中一条规范是一对谓词：前置条件和后置条件（这两个术语及其后提及的所有术语都将在以后的相应章节中给出定义）。另一个紧密相关的理论是戴克斯特拉（Dijkstra）的最弱前置条件谓词转换器, 它将程序和后置条件通过转换得到前置条件，它后来发展成了 Back 的精化演算。Jones 的维也纳开发方法已被某些产业采用并获益；其中, 一条规范是一对谓词（正如霍尔逻辑）, 但第二个谓词是一个关系。另外还有一些专门用于实时程序设计、概率程序设计、交互式程序设计的理论。

本书中的理论比上面提到的都要简单。在该理论中, 一条规范就是一个二元表达式。精化就是通常的蕴含式。该理论也比上述理论更为通用, 同时适用于终止和非终止计算, 顺序和并行计算, 以及独立和交互式计算。本书可以同时有只对其初值和终值感兴趣的变量、对其值始终感兴趣的变量、只知道其可能值的变量和用于计算时间和空间的变量。它们都适用于同一理论中，其基础在于将规范表示成二元表达式这一标准的科学实践，而二元表达

式的（非局部）变量可以是任何感兴趣的变量。

有一种通过穷举测试所有输入来证明程序的方法，称为模型检测。它优于本书中理论的一点在于可以完全自动化。通过明智的二元表达式的表示（见练习 15），模型检测当前声称可以处理最多可达约 10^{60} 个状态。这几乎和银河系中所估计的原子个数一样多！这个数字令人印象深刻，直到我们意识到 10^{60} 约为 2^{200}，这意味着讨论的是 200 位二进制数。也就是 6 个 32 位二进制数变量的状态空间。对任何一个超过六个变量的程序进行模型检测需要抽象化，每个抽象需要证明其保持所感兴趣的性质。而这些抽象和证明不是自动的。为了保证实用，模型检测必须与其他的证明方法相结合，例如本书中的方法。

本书还要一直强调的一点是，程序开发的每一步都是伴随证明的，而不是在开发以后进行证明。

有一个基于本书的网上课程在www.cs.utoronto.ca/~hehner/FMSD 。有关该课程的练习解答在www.cs.utoronto.ca/~hehner/aPToP/solutions 。

0.0 当前版本

从本书的第一版开始，增加了空间约束和概率程序设计的新的资料。归纳了 **for** 循环的规则。简化了对并发的处理。且对并行进程之间的合作提供了选择：通信（如第一版），和交互变量，交互变量是共用存储空间正式的易处理的版本。本书加强了解释和说明，并增加了更多整理过的例子。

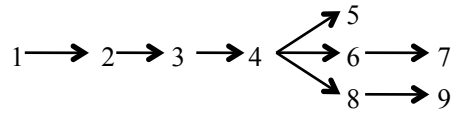
在增加内容的同时，也有删减。为了保证本书的精练，任何在课程中通常略过的材料被删除了。本书只有 159 页，后面的只是练习和参考材料。

----------当前版本结束

0.1 快速浏览

本书用到的所有术语都会在书中得到解释。每一条应该学习的新术语都有下划线标志。在解释这些术语时, 本书尽可能采用释义的方法而少用纪念性的人名。例如，只有两个值的数据类型被称作为“二元（binary）”而不是“布尔（boolean）”。前一个词是描述数据类型；但后者是为了纪念乔治布尔（George Boole）。本书不使用缩略词、首字母缩略语或其他一些难懂的语言，以便读者更易理解。本书假定读者不具备专门的数学知识或程序设计经验。但在本书第 1、2、和 3 章中, 有关二元值、数字、表和函数的预备知识只作了简单介绍，具有预先的知识或经验可能有帮助。

下面的图表示了各章与先前章节的依赖关系:



第 4 章，程序理论，是本书的核心章节。其后的章节可以根据上图和各人的兴趣进行

取舍。上图仅有的例外是第 9 章使用了 5.0.0 小节介绍的变量说明, 并且可选节 9.1.3 要用到第 6 章的知识。在每章中，有可选标志的节或小节可以忽略，不会对后面章节的理解产生影响。

第 10 章全部由练习组成, 这些练习所用到的理论都已在前面各章中作过介绍。“程序理论”一节的全部练习可利用第 4 章所介绍的方法完成；不过, 其后几章中又介绍了一些新的方法，可以用后面所学的知识重做同样的练习。

本书的最后一章, 即第 11 章包含了参考材料。11.0 节“释疑”（Justification）将回答以前章节的一些问题，诸如: 为什么要用这种方法呈现?, 为什么要呈现这个?，为什么不是呈现别的? 等等。也许只有那些已经具有丰富程序设计理论知识的教师和研究工作者才对此章感兴趣, 而那些第一次接触到形式化方法的学生可能不感兴趣。不过, 如果发现有上述那样的问题时, 不妨参阅“释疑”这一节。

第 11 章还包括术语的索引（中文版略去索引，替代以中英文术语对照）和本书所用到的所有规则的清单。对于那些热衷于程序设计的学生, 这些规则将会成为他们的朋友。本书最后几页列出了书中使用的全部记号。读者并不需要在阅读本书前预先知道这些记号；因为它们在首次使用时都会被解释。欢迎读者发明一些新的记号, 当然必须说明它们的用途。有时记号的选择会导致解决一个问题的能力差异。

----------快速浏览结束

0.2 致谢

感谢国际信息处理联合会（IFIP）的2.3工作小组（程序设计方法学），特别是Edsger Dijkstra、David Gries、Tony Hoare、Jim Horning、Cliff Jones、Bill McKeeman、Carroll Morgan、Greg Nelson、John Reynolds 和Wlad Turski, 他们给予我启发和指导；特别感谢Doug McIlroy的鼓励。感谢我的研究生和助教, 他们给了我不少有益的提示, 特别是Ray Blaak、Benet Devereux、Lorene Gupta、Peter Kanareitsev、Yannis Kassios、Victor Kwan、Albert Lai、Chris Lengauer、Andrew Malton、Theo Norvell、Rich Paige、Dimi Paun、Mark Pichora、Hugh Redelmeier、Alan Rosenthal、Anya Tafliovich、Justin Ward、和Robert Will。十分感谢Wim Hesselink、Jim Horning和 Jan van de Snepscheut，他们对初稿进行了严格和有益的审阅。感谢Ralph Back、Eike Best、Wim Feijen、Netty van Gasteren、Nicolas Halbwachs、Gilles Kahn、Leslie Lamport、Alain Martin、Joe Morris、Martin Rem、Pierre-Yves Schobbens、Mary Shaw、Bob Tennent和Jan Tijmen Udding，他们向我提出了好的想法。感谢Jules Desharnais、Andy Gravell、Peter Lauer、Ali Mili、Bernhard Moeller、Helmut Partsch、Jorgen Steensgaard-Madsen和Norbert Voelker, 他们阅读了文稿并提出了改进意见。感谢我班上的学生，他们找出了本书中的一些错误。

----------致谢结束

----------序言结束

第一章　基本理论

我们所需的基本理论有二元理论、数理论、和字符理论。

1.0 二元理论

二元理论（Binary Theory）, 又称为布尔代数或者逻辑, 原本是设计出来作为推理的辅助工具的, 这里用它对计算进行推理。二元理论的表达式称为二元表达式。二元表达式分为两类: 一类叫定理, 另一类叫反定理。

二元表达式可用来表示客观世界命题；定理表示真命题, 反定理表示假命题。这是二元理论的原本应用, 是该理论的设计原由, 且给出了大量的术语。二元理论的另一个十分合适的应用是数字电路设计。此时二态表达式代表电路；定理代表了输出为高电压的电路, 而反定理代表了输出为低电压的电路。总的来说，二元理论可用于任何有两个值的应用。

两个最简单的二元表达式是 T 和 ⊥。前者 T 是定理, 后者是反定理。当二元理论用于原本应用时, 将 T 读为 “真”, ⊥读成 “假”, 因为前者表示任意一个为真的命题, 后者表示任意一个为假的命题。当二元理论用于数字电路设计时, 将 T 读成 “高电压”, ⊥读成 “低电压”, 或者分别读成 “火线”和 “地线”。同样地根据不同的领域它们有不同的命名。或者，为了独立于不同的应用，我们可以称他们为“上”和“下”。它们是无运算元二元运算符, 因为它们无需运算对象。

单目二元运算符共有四个, 这里只讨论其中之一。其符号为¬, 读作 “非”。它是一个前缀运算符（置于运算对象之前）。对于形如 $\neg x$ 的表达式称作否定式。如果将定理求否就得到反定理; 而将反定理求否就得到定理。可以用下面的真值表描述这一求否过程:

¬	T	⊥
	⊥	T

在横线上方, T 表示运算元为定理, ⊥表示运算元为反定理。在横线下方, T 表示结果为定理, ⊥表示结果为反定理。

双目二元运算符共有十六个。出于传统习惯, 只使用其中的六个，尽管其他运算符也很有意思。它们都是中缀运算符（置于两个运算对象之间）。以下是它们的符号和一些读法。

- ∧　与
- ∨　或
- ⇒　蕴含, 等于或强于
- ⇐　由...导出，被...蕴含，弱于或等于
- =　等于，当且仅当
- ≠　不同于，不等于，异或，二元加法

表达式形如 $x \wedge y$ 称作合取式, 运算对象 x 和 y 称作合取因子。表达式形如 $x \vee y$ 称为析取式, 其中的运算对象称作析取因子。表达式形如 $x \Rightarrow y$ 称为蕴含式, 其中 x 称作前件, y 称作后件。表达式形如 $x \Leftarrow y$ 称为反向蕴含式, x 为后件, y 为前件。表达式形如 $x = y$ 称为等式, 其中的运算对象分别称作左式, 和右式。表达式形如 $x \neq y$ 称为不等式, 其中的运算对象也分别称作左

式和右式。

下面的真值表显示了不同种类的运算对象在双目运算符操作下的不同结果。横线之上, T T 表示两个运算对象都是定理; T⊥表示左运算对象是定理, 右运算对象是反定理; 依此类推。横线之下, T 表示结果是定理, ⊥表示结果是反定理。

	T T	T⊥	⊥T	⊥⊥
∧	T	⊥	⊥	⊥
∨	T	T	T	⊥
⇒	T	⊥	T	T
⇐	T	T	⊥	T
=	T	⊥	⊥	T
≠	⊥	T	T	⊥

中缀运算符可能造成一些表达式变得模糊。例如, ⊥∧T ∨T 可被理解为合取式⊥∧T, 结果为反定理, 然后再与 T 做析取，最后结果为定理。还可以被理解为⊥与析取式 T ∨T 做合取, 最后结果为反定理。利用圆括号可以区别表示上述的不同理解: (⊥ ∧ T) ∨T 或⊥∧ (T ∨T)。为了避免过多使用圆括号, 可引入优先级表, 它被列在本书的最后一页上。在这张优先级表中, ∧优先级为 9, ∨ 的优先级为 10；这意味着在缺省括号的情形下, ∧的运算优先于∨ 。因此, 上例 ⊥ ∧T ∨T 的运算结果应是定理。

运算符 = ⇒ ⇐在优先级表中分别出现了两次, 大的 = ⇒ ⇐优先级为 16，应用于所有运算符之后。小的运算符除了优先级不同外, 其它操作性质都与相应的大运算符一样。在一定的使用限制下, 这些重复的运算符有时可以进一步减少圆括号的使用量，从而提高表达式的可读性。提醒一句: 适当使用少数圆括号, 尽管根据优先级这些括号可能是不必要的, 但可以帮助理解表达式结构。请根据具体情况决定是否使用它们。

三目运算符共有 256 个, 这里只介绍一个。称为条件组合, 写成 **if** *x* **then** *y* **else** z **fi**。下面是它的真值表:

	TTT	TT⊥	T⊥T	T⊥⊥	⊥TT	⊥T⊥	⊥⊥T	⊥⊥⊥
if then else fi	T	T	⊥	⊥	T	⊥	T	⊥

对于每个自然数 *n*， *n* 目运算符共有 2^{2^n} 个, 但现在介绍的已足够了。

在前面介绍的合取表达式 *x* ∧ *y* 中, *x* 和 *y* 为变量, 它们分别可以被任意的二元表达式替换, 这样 *x* ∧ *y* 就可以代表所有的合取表达式。例如, 用(⊥ ⇒ ¬ (⊥∨T))代替 *x*, 用(⊥∨T)代替 *y*, 就得到下面的合取式:

$$(\bot \Rightarrow \neg\ (\bot \vee T\ )) \wedge (\bot \vee T)$$

用表达式替换变量叫做置换, 或者叫做实例化。由于变量可以被置换, 所以表达式中允许使用变量，但应注意以下两点：

- 在用表达式置换变量时, 为了不改变运算符执行的优先次序, 有时需将表达式用括号括起来。例如，在前述的置换例子中, 用蕴含式⊥ ⇒ ¬ (⊥∨T)代替 *x*, 但由于合取运算优先于蕴含, 所以必须在该蕴含式外加上括号。类似地在用析取式 ⊥∨T 代替 *y* 时也应加上括号。

- 当同一个变量在一个表达式中出现两次或两次以上时，必须使用相同的表达式去置换该变量的每一次出现。例如, 表达式 $x \wedge x$ 可以被置换成 $\mathsf{T} \wedge \mathsf{T}$, 但不能置换成 $\mathsf{T} \wedge \bot$。但对于不同的变量可以用相同的或不相同的表达式来置换, 例如, $x \wedge y$ 可以被置换成 $\mathsf{T} \wedge \mathsf{T}$ 或 $\mathsf{T} \wedge \bot$。

在介绍其他理论时, 将引入一些利用该理论中的表达式而构成的新二元表达式, 并且将它们进行归类。例如, 当介绍数论时, 需要引入自然数表达式 1+1 和 2，从而引入二元表达式 1+1=2, 这个二元表达式归类为定理。不能将一个二元表达式既归类成定理又归类成反定理。客观世界中的命题在同一含义下不能够既是真又是（在同一逻辑上）假; 一个电路的输出不能既是高电压又是低电压。如果不小心将一个二元表达式同时归类成这两个结果, 就犯了一个严重的错误。但是允许一个二元表达式不被分类。例如, 1/0=5, 它既不是定理又不是反定理。一个无类别的表达式可以对应于这样一个命题, 即不知道或者不关心它是真或是假, 或对应于这样一个电路, 它的输出无法预测。如果一个理论中不存在既是定理又是反定理的二元表达式, 则称该理论是一致的; 反之, 若存在既是定理又是反定理的二元表达式, 则称为不一致的。如果一个理论中每个完全实例化的二表元达式或者是定理或者是反定理, 那么称该理论是完备的; 反之, 若存在一些完全实例化的二元表达式既不是定理又不是反定理, 则称该理论是不完备的。

1.0.0 公理和证明规则

提出一个理论，并指出它的表达式是什么，它的定理与反定理是什么。要证明一个二元表达式是定理或是反定理, 必须遵循下面五条规则。下面首先对它们进行叙述, 然后讨论。

公理(Axiom)规则: 如果一个二元表达式是公理, 那么它是定理。如果一个二元表达式是反公理, 那么它是反定理。

求值(Evaluation)规则: 如果一个二元表达式的所有二元子表达式都已归类, 那么该二元表达式可根据其真值表进行归类。

完备性(Completion)规则: 如果一个二元表达式含有无类别的二元子表达式, 并且对无类别的二元子表达式的所有归类方法都将该二元表达式归成同一类, 那么该二元表达式就属于该类。

一致性(Consistency)规则: 如果一个已归类的二元表达式含有若干二元子表达式, 并且只有一种对二元子表达式的归类方法是一致的, 那么它们就按这种方法归类。

实例化(Instance)规则: 如果一个二元表达式是已归类的, 那么它的所有实例化值都为那一类。

一个声明为定理的二元表达式称为公理。类似地, 一个声明为反定理的二元表达式称为反公理。在二元理论中只有一个公理 T 和一个反公理 $\bot$。因此根据公理规则, T 是定理, $\bot$是反定理。当引入更多理论时，将会给出它们的公理和反公理；它们与其它的证明规则一起，就可以决定新理论中的新的定理和反定理。

在形式逻辑发明之前, “公理”一词用来表示那些显然为真的命题。在现代数学中, 公理

是设计和阐述理论的一部分。不同的公理可以形成不同的理论, 而不同的理论可以具有不同的应用。在设计理论时, 可以任意选用一些公理, 不过不好的选择可能导致无用的理论。

在双目运算符真值表中, 第一个表项不是要说明 T ∧T = T, 而是要说明两个定理的合取是定理。要证明 T ∧T = T 是定理, 需要的是: 二元公理 （证明 T 是定理）, 真值表∧所在行的第一个表项（证明 T ∧T 是定理）以及运算符=所在行的第一个表项（证明 T ∧T = T 是定理）。

二元表达式

$$T \vee x$$

含有一个无类别的二元子表达式, 所以不能使用求值规则得出它属于哪一类。如果 *x* 是定理, 根据求值规则可导出整个表达式是定理。如果 *x* 是反定理, 根据求值规则导出整个表达式仍是定理。于是, 利用完备性规则, 可得出整个表达式的确是定理。利用完备性规则也可以得出：

$$x \vee \neg x$$

是定理，那么在数论中,

$$1/0=5 \vee \neg 1/0=5$$

是定理。在利用完备性规则时, 也不必确信某个子表达式是无类别的。只要忽略它的类别, 假设它是无类别的, 就可以利用完备性规则得到仍然是正确的结论。

在一个已归类的二元表达式中, 如果将其二元子表达式假定为某一类别, 得出的二元表达式结果与原来的归类不一致, 利用一致性规则, 就可认为该二元子表达式属于与原来假定相反的那类。例如, 假定已知表达式 *expression* 0 是定理, 并且 *expression* 0 ⇒ *expression* 1 也是定理, 那么能否确定 *expression* 1 属于哪一类别? 如果 *expression* 1 是反定理, 利用求值规则, *expression* 0⇒ *expression* 1 将是反定理, 这就出现了不一致性。因此, 根据一致性规则, *expression* 1 应为定理。一致性规则的这种应用传统上称为 “分离定律”(detachment)或 “假言推理”(modus ponens)。另一个例子, 如果¬*expression* 是定理, 根据一致性规则, 可以得出 *expression* 是反定理。

由于有否定运算符和一致性规则，可以不必讨论反公理和反定理。与其说 *expression* 是反定理或反公理, 不如说 ¬*expression* 是定理或公理。这里应提醒一句: 如果一个理论是不完备的， 那么有可能 *expression* 和 ¬*expression* 都不是定理。因此, “反定理”是不同于 “不是定理”的。这里选用定理而不选用反定理进行讨论, 可以简化一些说法。有时提及二元表达式, 例如 1+1=2, 如果无任何说明, 就意味着它是定理。如果想证明什么, 就意味着将证明它是定理。

----------公理和证明规则结束

现在把二元反公理（⊥）替换成一个公理（¬⊥）。利用两个公理（T 和 ⊥）和五个证明规则, 现在就可以证明定理了。有一些定理是非常有用的, 于是给它们起了名, 便于记忆, 或至少放在速查表里。这样的定理称之为定律。二元理论的一些定律列在本书的最后部分。定律中都没有使用⇐, 这是因为任何使用⇒的定律都可以很容易改变成用⇐的形式而含义不变。所有定律都可以用完备性规则证明, 这可通过穷举变量所有可能的值, 并分别计算它们来实现。不过当变量数目大于 2 时, 这种证明方法是相当低效的。较好的方法是利用已证明过的旧定律来证明新定律, 在下一节里将看到如何证明。

1.0.1 表达式和证明格式

利用本书最后一页的优先级表, 可以分析一个不含括号的表达式的含义。为了方便阅读，可在依据优先级表而省略的括号处留出一些空格。考虑下面表达式的两种留空方法:

$$a \wedge b \ \vee \ c$$
$$a \ \wedge \ b \vee c$$

根据优先规则, 缺省的括号应在 $a \wedge b$ 两边, 所以第一种留空方法是有效的, 而第二种则会产生误导。

表达式如果太长, 一行写不下, 可以分成若干部分。有几种合理做法, 这里是一种建议: 被括起的长表达式可在它的主运算符处一分为二, 将主运算符另起一行, 置于左括号之下。例如

$$\begin{array}{l} (\ \textit{first part} \\ \wedge\ \textit{second part}\ ) \end{array}$$

缺省了括号的长表达式, 可在主运算符处一分为二, 将主运算符另起一行置于被缺省的左括号之下。例如,

$$\begin{array}{l} \textit{first part} \\ =\ \textit{second part} \end{array}$$

选用适当的书写格式可使我们更容易地理解复杂表达式。

证明是一个二态表达式，很清楚地它是一个定理。对一个人来说清楚的东西不一定对另一个人也清楚，所以证明是为特定的读者写的。证明可以写成伴随短提示的连续等式的格式。

expression 0	短提示 0
= *expression* 1	短提示 1
= *expression* 2	短提示 2
= *expression* 3	

以上连续的等式是以下这个较长的二态表达式的简写法。

expression 0 = *expression* 1
∧ *expression* 1 = *expression* 2
∧ *expression* 2 = *expression* 3

右边的短提示当需要的时候使用，可以帮助理解为什么该连续等式是定理。最好的短提示是定律的名字。“短提示 0”用以解释为什么 *expression* 0 = *expression* 1 是定理; “短提示 1”解释为什么 *expression* 1 = *expression* 2 是定理, 依次类推。根据=的传递性，上述等式证明了 *expression* 0 = *expression* 3 是定理。形式化证明是每一步都符合作为短提示的定律的格式的证明。使证明形式化的好处是每一步都可被电脑检测，并且其有效性不成问题。

例如，假定只使用本书后面定律列表中的前面部分证明移动定律（laws of portation）的第一条

$$a \wedge b \Rightarrow c = a \Rightarrow (b \Rightarrow c)$$

以下是证明过程：

$a \wedge b \Rightarrow c$	实质蕴含
$= \neg(a \wedge b) \vee c$	对偶定律
$= \neg a \vee \neg b \vee c$	实质蕴含

$= a \Rightarrow \neg b \vee c$ 实质蕴含

$= a \Rightarrow (b \Rightarrow c)$

证明的第一行使用了“实质蕴含”, 它是包含定律的第一条, 以得到证明中的第二行。前两行加在一起

$$a \wedge b \Rightarrow c \ = \neg(a \wedge b) \vee c$$

符合实质蕴含定律的格式，也就是

$$a \Rightarrow b \ = \neg a \vee b$$

因为证明中的 $a \wedge b$ 符合定律中 a 的角色，而证明中的 c 符合定律中 b 的角色。下一个短提示是“对偶定律”，利用对偶性定律的第一条, 将$\neg(a \wedge b)$用$\neg a \vee \neg b$ 代替就得到第三行。在第三行中没有括号, 因此可利用析取的结合律, 为下一步做准备。下一个提示又是 “实质蕴含”, 这里作了一个反向应用, 将第一个析取用蕴含式代替。接着再利用 “实质蕴含”, 将余下的析取用蕴含式代替。这样通过=的传递性, 证明了移动定律的第一条是定理。

下面是相同的证明, 但使用了另外一种证明格式

$(a \wedge b \Rightarrow c = a \Rightarrow (b \Rightarrow c))$ 实质蕴含，三次

$= (\neg(a \wedge b) \vee c = \neg a \vee (\neg b \vee c))$ 对偶定律

$= (\neg a \vee \neg b \vee c = \neg a \vee \neg b \vee c)$ =的自反性

$= \mathsf{T}$

最后一行是定理, 因此其它每一行也都是定理, 因此第一行也就是定理。这种证明格式比上一种有以下优势：首先，它将证明等同于到 T 的简化。其次，尽管任何第一种格式的证明可以写成第二种格式的证明，但反之不成立。例如证明：

$(a \Rightarrow b = a \wedge b) = a$ =的结合律

$= (a \Rightarrow b = (a \wedge b = a))$ 包含定律

$= \mathsf{T}$

就无法转换成其他形式。最后，简化成 T 的第二种证明格式还可用于非等式；该二元表达式的主运算符可为任意运算符，包括 ∧ , ∨ , 或 ¬ 。

本书中的证明的意图是给读者读的，而不是给电脑读的。有时从上一行到下一行的推导不用提示已很清楚, 这时可不必给出提示。提示是可选项，应根据需要决定是否给出。有时提示太长了, 在一行的余空里写不下。例如有证明

expression 0 短提示

= *expression* 1 接着是一个很长的提示, 就像写出的这样, 能写多少行就写多少行, 然后再跟上

= *expression* 2

我们不能因为一行的余空限制了应该有的提示内容。

----------表达式和证明格式结束

1.0.2 单调性和反单调性

如前面所看到的，证明可以是连续等式的形式。证明也可以是连续蕴含式, 或者是连续的蕴含式和等式的混合。例如要证明归并定律的第一条规则：

$$(a \Rightarrow b) \wedge (c \Rightarrow d) \Longrightarrow a \wedge c \Rightarrow b \wedge d$$

从右边开始，证明如下：

$a \wedge c \Rightarrow b \wedge d$ 将⇒分布作用到第二个∧上

$=$ $(a \wedge c \Rightarrow b) \wedge (a \wedge c \Rightarrow d)$ 反分布作用两次

$=$ $((a \Rightarrow b) \vee (c \Rightarrow b)) \wedge ((a \Rightarrow d) \vee (c \Rightarrow d))$ 将∧分布作用于∨两次

$=$ $(a \Rightarrow b) \wedge (a \Rightarrow d) \vee (a \Rightarrow b) \wedge (c \Rightarrow d) \vee (c \Rightarrow b) \wedge (a \Rightarrow d) \vee (c \Rightarrow b) \wedge (c \Rightarrow d)$ 普遍化

$\Leftarrow$ $(a \Rightarrow b) \wedge (c \Rightarrow d)$

由$=$和$\Leftarrow$的相互可传递性, 就证明了

$$a \wedge c \Rightarrow b \wedge d \Leftarrow (a \Rightarrow b) \wedge (c \Rightarrow d)$$

重写该式, 很容易就得到了所要的定理。

蕴含运算符是自反的 $a \Rightarrow a$、反对称的 $(a \Rightarrow b) \wedge (b \Rightarrow a) = (a = b)$ 和传递的 $(a \Rightarrow b) \wedge (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$。因此它是一个序（正如 $\leq$ 是数的一个序）。将 $a \Rightarrow b$ 读作“a 蕴含 b”，或为了强调序，读作“a 强于或等于 b”。类似地，$a \Leftarrow b$ 被读作“a 被 b 蕴含”，或读作“a 弱于或等于 b”。“强于”和“弱于”的说法起源于哲学；可以忽略它们的其他含义而只关注其二元序的含义，那么 $\bot$ 强于 T。

单调性定律 $a \Rightarrow b \implies c \wedge a \Rightarrow c \wedge b$ 可以（粗略地）读作：若 a 削弱为 b，则 $c \wedge a$ 削弱为 $c \wedge b$。（更小心的说法是“削弱或相等”）。如果削弱 a，则 $c \wedge a$ 也被削弱。换句话说，如果增强 b，则 $c \wedge b$ 也被增强。对合取因子发生的（削弱或增强），也对合取式发生。则称合取式对其合取因子是单调的。

反单调定律 $a \Rightarrow b \implies (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$ 说明对前件发生的（削弱或增强），对整个蕴含式正好相反。因此称蕴含式对其前件是反单调的。

下面是二元表达式的单调性和反单调性性质：

$\neg a$ 对 a 是反单调的

$a \wedge b$ 对 a 和 b 是单调的

$a \vee b$ 对 a 和 b 是单调的

$a \Rightarrow b$ 对 a 是反单调的，对 b 是单调的

$a \Leftarrow b$ 对 a 是单调的，对 b 是反单调的

if a **then** b **else** c **fi** 对 b 和 c 是单调的

这些性质在证明时是很有用的。例如，练习 6(k)中，证明 $\neg(a \wedge \neg(a \vee b))$，可以使用普遍化定律 $a \Rightarrow a \vee b$ 将 $a \vee b$ 增强为 a。这样就削弱了 $\neg(a \vee b)$，并削弱了 $a \wedge \neg(a \vee b)$，并增强了 $\neg(a \wedge \neg(a \vee b))$。

$\neg(a \wedge \neg(a \vee b))$ 使用普遍化定律

$\Leftarrow$ $\neg(a \wedge \neg a)$ 使用非矛盾定律

$=$ T

因此证明了 $\neg(a \wedge \neg(a \vee b)) \Leftarrow$ T，根据同一律，这就等同于证明了 $\neg(a \wedge \neg(a \vee b))$。换句话说，$\neg(a \wedge \neg(a \vee b))$ 弱于或等于 T，而因为没有什么能比 T 弱，因此它必等于 T。因此当以 T 为目标时，证明的左边可以是$=$和$\Leftarrow$的任意混合。

类似地，也能以 $\bot$ 为目标，那么证明的左边可以是$=$和 $\Rightarrow$ 的混合。例如：

$a \wedge \neg(a \vee b)$ 使用普遍化定律

$\Rightarrow a \wedge \neg a$ 使用非矛盾定律

$= \bot$

这称为“反证法”。它证明了 $a \wedge \neg(a \vee b) \Rightarrow \bot$，这等同于证明 $\neg(a \wedge \neg(a \vee b))$。任何反证都能转变为以 T 为目标的简化证明，只需将每一行加上 ¬ 。

----------单调性与反单调性结束

1.0.3 上下文

证明或证明的一部分可以利用局部假设。例如，证明可有如下格式:

assumption
⇒ (*expression*0
= *expression*1
= *expression*2
= *expression*3）

其中 *expression*0 = *expression*1 可以利用假设 *assumption*, 把它当成公理来使用。下一步 *expression*1 = *expression*2 也可以这样, 依此类推。在括号里我们完成了证明；而这个证明可以是任意一种证明, 甚至还可以包括更深一层的局部假定。因此, 可以在证明中嵌套证明, 以适当的缩进方式书写。如果子证明是求证 *expression*0 = *expression*3, 那么整个证明就是求证

$$assumption \Rightarrow (expression0 = expression3)$$

如果子证明是求证 *expression*0 那么整个证明是求证

$$assumption \Rightarrow expression0$$

如果子证明是求证⊥, 那么整个证明是求证

$$assumption \Rightarrow \bot$$

它等价于¬*assumption*, 就是前面提到的 “反证法”。

类似地，还可以用 **if then else fi** 结构作为证明，或证明的一部分。其使用格式如下：

if *possibility*
then 第一个子证明
假设 *possibility*
是一个局部公理
else 第二个子证明
假设¬ *possibility*
是一个局部公理 **fi**

如果第一个子证明求证了 *something*, 而第二个子证明求证了 *anotherthing*, 那么整个证明就求证了

if *possibility* **then** *something* **else** *anotherthing* **fi**

如果两个子证明求证了相同的结论, 那么根据 “情况幂等定律”, 就完成了整个证明, 这是此结构最常用的方法。

考虑证明中的一步如下：

*expression*0 ∧ *expression*1
= *expression*0 ∧ *expression*2

当我们将 *expression*1 转换到 *expression*2 时，可以假设将 *expression*0 为这一步的局部公理。若 *expression*0 的确是定理，那么这个假设是无害的。反之，若 *expression*0 是反定理，那么

不管 *expression*1 和 *expression*2 是什么， *expression*0 ∧ *expression*1 和 *expression*0 ∧ *expression*1 都是反定理，因此仍然可以这样假设。对称地，当证明

*expression*0 ∧ *expression*1
＝*expression*2 ∧ *expression*1

时，也可假设 *expression*1 是局部公理。然而，当证明

*expression*0 ∧ *expression*1
＝ *expression*2 ∧ *expression*3

时，不能假设 *expression*0 来证明 *expression*1 = *expression*3，同时假设 *expression*1 来证明 *expression*0 = *expression*2。例如，从 $a \wedge a$ 开始，可以假设第一个 a，将第二个 a 转换成 T，

$a \wedge a$　　假设第一个 a 以简化第二个 a
＝ $a \wedge \mathsf{T}$

或者假设第二个 a，将第一个 a 转换成 T，

$a \wedge a$　　假设第二个 a 以简化第一个 a
＝ $\mathsf{T} \wedge a$

但不能同时假设两个。

$a \wedge a$　　这一步是错误的
＝ $\mathsf{T} \wedge \mathsf{T}$

在这一段中，等号相当于任一方向的蕴含。

下面是证明中的上下文规则：

在 *expression*0 ∧ *expression*1 中，当转换 *expression*0 时，可以假定 *expression*1；
在 *expression*0 ∧ *expression*1 中，当转换 *expression*1 时，可以假定 *expression*0；
在 *expression*0 ∨ *expression*1 中，当转换 *expression*0 时，可以假定¬*expression*1；
在 *expression*0 ∨ *expression*1 中，当转换 *expression*1 时，可以假定¬*expression*0；
在 *expression*0 ⇒ *expression*1 中，当转换 *expression*0 时，可假定¬*expression*1；
在 *expression*0 ⇒ *expression*1 中，当转换 *expression*1 时，可以假定 *expression*0；
在 *expression*0 ⇐ *expression*1 中，当转换 *expression*0 时，可以假定 *expression*1；
在 *expression*0 ⇐ *expression*1 中，当转换 *expression*1 时，可假定¬*expression*0；
在 **if** *expression*0 **then** *expression*1 **else** *expression*2 **fi** 中，当转换 *expression*0 时，
　可以假定 *expression*1 ≠ *expression*2；
在 **if** *expression*0 **then** *expression*1 **else** *expression*2 **fi** 中，当转换 *expression*1 时，
　可以假定 *expression*0；
在 **if** *expression*0 **then** *expression*1 **else** *expression*2 **fi** 中，当转换 *expression*2 时，
　可以假定¬*expression*0。

在上一小节中证明了练习 6(k)：$\neg(a \wedge \neg(a \vee b))$。这里使用上下文规则重新证明：

$\neg(a \wedge \neg(a \vee b))$　　假定 a，以简化 $\neg(a \vee b)$
＝ $\neg(a \wedge \neg(\mathsf{T} \vee b))$　　对称定律和 ∨ 的基定律
＝ $\neg(a \wedge \neg \mathsf{T})$　　¬ 的真值表
＝ $\neg(a \wedge \bot)$　　∧ 的基定律
＝ $\neg \bot$　　二元公理，或 ¬ 的真值表
＝ T

----------上下文结束

1.0.4 形式化

用计算机来求解问题, 或提供服务或仅仅为了娱乐。 起初通常是以非形式化的方法描述所期望的计算机行为, 如用自然语言（如英语）, 或许带上若干图示, 或许带上一些手势, 而不是用形式化的方法, 如使用数学公式（记号）。但是到了最后, 计算机的行为还是要用形式化方法描述成程序。所以程序员必须能够将非形式化的描述翻译成形式化的描述。

用自然语言描述的命题，其含义可能是含糊的、模棱两可的或者难以捉摸的, 而且要依赖于大量的文化背景。这使得形式化变得困难, 但也成为必要。这里不可能从通用的角度来介绍形式化方法；因为它需要对自然语言有一个精确而全面的了解, 而这本身是个一直在讨论的课题。本小节指出在将自然语言翻译成二元表达式时几个易犯的错误。

最佳的翻译方法并不是单词与符号的一对一置换。同样一个单词在不同的地方可以被翻译成不同的符号；相反地，不同的单词也可能被翻译成相同的符号。例如单词 “和”, “也”, “但是”, “还”, “然而”以及 “此外”都可能译成∧。若仅表示将若干东西彼此并列有时也可译成∧。例如， “它们是红色的、熟的、多汁的，但不甜。”可翻译成 *红色的* ∧ *熟的* ∧ *多汁的* ∧ ¬ *甜*。

自然语言中的 “或”有时译成∨, 而有时也译成 ≠。例如句子 “它们小或者腐烂了。”就包含了它们既 “小” 又已“腐烂了”这种可能性, 所以应译成 *小*∨*腐烂*; 但句子 “我们会吃了它们或者储藏它们。 ”则表示不可能两者都做, 因此最好译成 *吃*≠*储藏*。

另外, 自然语音中的“如果（if）”有时译成⇒, 而有时也译成 =。例如，句子 “如果下雨了，我们就待在家里。” 并不排除即使不下雨也呆在家里的可能性, 因而应译成 *下雨*⇒*待家里*； 但句子“如果下雪了，我们就去滑雪。” 则同时表示 “如果没下雪，就不去。”，因此最好译成 *下雪*=*滑雪*。

----------形式化结束

----------二元理论结束

1.1 数论

数论, 又称算术, 用于表示数量。在本书介绍的数论中, 数表达式由下列方法构成：

由一个或多个十进制数字组成的一个序列

∞	“无穷大”
+*x*	“加 *x*”
−*x*	“减 *x*”
x+*y*	“*x* 加 *y*”
x−*y*	“*x* 减 *y*”
x×*y*	“*x* 乘以 *y*”
x / *y*	“*x* 除以 *y*”
x^y	“*x* 的 *y* 次幂”
if *a* **then** *x* **else** *y* **fi**	

其中 *x* 和 *y* 是任意数表达式, *a* 是任意二元表达式。无穷大数表达式∞在讨论程序的执行时间时是必不可少的。这里也引入几种构造二元表达式的新方法：

$x < y$ “x 小于 y”
$x \le y$ “x 小于或等于 y”
$x > y$ “x 大于 y”
$x \ge y$ “x 大于或等于 y”
$x = y$ “x 等于 y”, “x 与 y 相等”
$x \ne y$ “x 不同于 y”, “x 与 y 不相等”

数论的公理列在本书最后。列表很长, 但其中大多数公理应该很熟悉。特别注意以下两公理：

$-\infty \le x \le \infty$ 极值
$-\infty < x \Rightarrow \infty + x = \infty$ 吸收律

数论是不完备的。例如, 二元表达式 1/0=5 和 $0<(-1)^{1/2}$ 既不能证明是定理又不能证明是反定理。

----------数论结束

1.2 字符理论

最简单的字符表达式写成图形的形状被左双引号和右双引号左右围住。例如 “A”是 “大写A”字符, “1”是 “一”字符, “ ” 是“空格”字符。左右双引号字符必须写两次，并被围住，例如这样： ““““”和 ““””””。 字符理论很简单, 其运算符有 *succ*(后继), *pred*(前趋), 以及= ≠ < ≤ > ≥ **if then else fi**。该理论的详细内容留待有兴趣的读者自己学习。

----------字符理论结束

这里所有的理论都使用了操作符 = ,≠, **if then else fi**, 因此本书最后列出的有关它们的定律都归于 “通用符号”一类, 表示它们在每个理论中都适用。这些定律不必作为二元理论的公理；例如, x=x 可以利用完备性规则和求值规则证明得到。但是在数论和其它理论中, 它们是公理, 否则连 5=5 都无法证明。

运算符< ≤ > ≥ 可以用于某些类型的表达式, 而不是所有。每当可以用到它们时, 都可运用其相应列于书后 “通用符号”栏下的公理。

----------基本理论结束

本章中介绍了二元表达式、数表达式以及字符表达式。在余下章节中，还要介绍束表达式、集合表达式、字符串表达式、表表达式,、函数表达式、谓词表达式、关系表达式、规范表达式，以及程序表达式等许多。为简洁起见, 以后各章在提及上述表达式时均省略“表达式”一词, 而只说二元、数、字符、束、集合、串、表、函数、谓词、关系、规范和程序，来表示在每个情况中是一种表达式。如果觉得这样会使自己混淆的话, 就请在需要的地方默默在心里加上“表达式”一词。

第二章 基本数据结构

数据结构是数据的集合体或聚集体。数据可以是二元值, 数, 字符或数据结构。我们考虑的基本的结构性质是包装性和索引性。这两种性质可以通过组合得到四种基本数据结构。

非包装的, 无索引的： 束
包装的, 无索引的： 集合
非包装的, 有索引的： 串
包装的, 有索引的： 表

2.0 束论

束代表了一个对象集合体。与之对照, 集合代表了一个在包裹或容器里的对象集合体。束是集合的内容。这一含糊的描述将在下文中加以精确化。

任何数, 字符或二元值(以及后面的集合、元素的串、元素的表)都是基本束, 或称元素。例如, 数 2 是一个基本束, 或称元素。实际上, 每个表达式都是一个束表达式, 虽然它们未必都是基本的。

如果 A 和 B 是束, 那么

A , B “A 并 B”
$A \text{ ‘ } B$ “A 交 B”

是束,

$\not\subset A$ “A 的大小”, “A 的基数”

是数, 而

$A : B$ “A 在 B 中”, “A 被 B 包含”

是二元值。

束的大小等于束所包含的元素个数。元素则是大小为 1 的束。例如

$\not\subset 2 = 1$
$\not\subset (0 , 2 , 5 , 9) = 4$

以下是三个简单的束包含例子:

$2 : 0 , 2 , 5 , 9$
$2 : 2$
$2 , 9 : 0 , 2 , 5 , 9$

第一例表示 2 在由 0, 2, 5, 9 组成的束之中。第二例表示 2 在仅由一个元素 2 组成的束中。注意我们不说 “一个束包含若干元素”, 而说 “一个束由若干元素组成”。第三例表示 2 和 9 都在束 0, 2, 5, 9 之中, 换句话说, 束 2, 9 包括在束 0, 2, 5, 9 之中。

以下是束论的公理。在这些公理中, x 和 y 是元素（基本束）, A, B, C 是任意的束。

$x : y \ = \ x=y$	基本公理
$x : A , B \ = \ x : A \ \vee \ x : B$	复合公理
$A , A \ = \ A$	幂等性
$A , B \ = \ B , A$	对称性
$A, (B, C) = (A, B), C$	结合性

$A \text{ ‘} A = A$	幂等性
$A \text{ ‘} B = B \text{ ‘} A$	对称性
$A \text{ ‘} (B \text{ ‘} C) = (A \text{ ‘} B) \text{ ‘} C$	结合性
$A, B : C = A:C \wedge B:C$	反分配性
$A : B \text{ ‘} C = A:B \wedge A:C$	分配性
$A : A , B$	普遍化
$A \text{ ‘} B : A$	特定化
$A : A$	自反性
$A : B \wedge B : A = A=B$	反对称性
$A : B \wedge B : C \Rightarrow A : C$	传递性
$\mathbb{C} x = 1$	大小
$\mathbb{C} (A , B) + \mathbb{C} (A \text{ ‘} B) = \mathbb{C} A + \mathbb{C} B$	大小
$\neg x : A \Rightarrow \mathbb{C} (A \text{ ‘} x) = 0$	大小
$A : B \Rightarrow \mathbb{C} A \leq \mathbb{C} B$	大小

从这些公理, 可以推导出许多定律。以下是其中一部分:

$A , (A \text{ ‘} B) = A$	吸收律
$A \text{ ‘} (A , B) = A$	吸收律
$A : B \Rightarrow C , A : C , B$	单调性
$A : B \Rightarrow C \text{ ‘} A : C \text{ ‘} B$	单调性
$A : B = A , B = B = A = A \text{ ‘} B$	包含性
$A , (B , C) = (A , B) , (A , C)$	分配性
$A , (B \text{ ‘} C) = (A , B) \text{ ‘} (A , C)$	分配性
$A \text{ ‘ } (B , C) = (A \text{ ‘} B) , (A \text{ ‘} C)$	分配性
$A \text{ ‘ } (B \text{ ‘} C) = (A \text{ ‘} B) \text{ ‘} (A \text{ ‘} C)$	分配性
$A : B \wedge C : D \Rightarrow A , C : B , D$	归并
$A : B \wedge C : D \Rightarrow A \text{ ‘} C : B \text{ ‘} D$	归并

以下是几个非常有用的束：

null	空束
bin = T, ⊥	二元值
nat = 0,1,2,...	自然数
int = ..., −2,−1,0,1,2,...	整数
rat = …, −1,0, 2/3 , ...	有理数
real = ..., $2^{1/2}$,...	实数
xnat = 0,1,2,…, ∞	广义自然数
xint = −∞, ...,-2,-1,0,1,2,..., ∞	广义整数
xrat = −∞, ...,-1,0,2/3,..., ∞	广义有理数
xreal = − ∞,...,∞	广义实数
char = ...,“a”, “*A*”,...	字符值

在上述等式中，三个点表示“可推测是什么”。这种用法是非形式化的，因此这里不能作为定义，只作为说明。本书将在后面定义这些束。

运算符 , ‘ $\mathbb{C}$: = ≠ **if then else fi** 根据提供的公理应用于运算对象上。一些其它的运

算符应用于束上时可以理解为应用于束的元素上。换句话说，它们对束的并满足分配律。例如：

$$-null = null$$
$$-(A,B) = -A,-B$$
$$A + null = null + A = null$$
$$(A,B) + (C,D) = A + C, A + D, B + C, B + D$$

这使得表示复数自然数(*nat*+2)，偶自然数($nat \times 2$)，平方(nat^2)，二的幂(2^{nat})等更容易。（对束的并满足分配律的运算符在最后一页列出。）

用以下公理定义空束 *null*：

$$null : A$$
$$\not\subset A = 0 = A = null$$

这又可得出以下三个定律:

$A,null = A$	恒等性
$A`null = null$	基
$\not\subset null = 0$	大小

束 *bin* 由公理

$$bin = \mathsf{T}, \perp$$

定义，束 *nat* 由以下两个公理定义

$0, nat + 1 : nat$	构造
$0, B + 1 : B \Rightarrow nat : B$	归纳

构造说明0,1,2,等在 *nat* 中。归纳说明在所有满足构造公理的束 *B* 中， *nat* 是最小的，因此说明在 *nat* 中不再包含别的数。在一些书中，特别是比较老的书中，自然数从 1 开始。这里采用当前更有用的术语，从 0 开始。束$\text{int}, rat, xnat, x\text{int}$和 *xrat* 可以定义如下：

$$\text{int} = nat, -nat$$
$$rat = \text{int}/(nat + 1)$$
$$xnat = nat, \infty$$
$$x\text{int} = -\infty, \text{int}, \infty$$
$$xrat = -\infty, rat, \infty$$

对实数（*real*）的定义将推迟到下一章（函数）。束 *real* 在定义之前不能使用，除非说

$$xreal = -\infty, real, \infty$$

对于束 *char* 这里不予定义。

本书还将使用如下记号：

x ,.. y　　“从 x 到 y”（不包括 y）

其中 *x* 是整数，*y* 是广义整数,并且 $x \le y$ 。其上的公理为：

$$i : x \text{ ,.. } y = x \le i < y$$

其中 *i* 是广义整数。记号 ,.. 是非对称的, 它表明所表示的区间包括最左边的值而不包括最右边的值。例如，

$$0,.. \infty = nat$$
$$5,..5 = null$$
$$\not\subset (x,..y) = y - x$$

,..记号是形式化的。因为使用了公理定义,..记号，因此不需要猜测它包括什么。

----------束论结束

2.1 集合论　　可选节

令 *A* 为任意束（任何东西）, 于是

{*A*}　　“包含 *A* 的集合”

是集合。那么{*null*}就是空集, 而包含头三个自然数的集合表示成{0,1,2}或{0,..3}。所有集合都是元素, 但并非所有束都是元素；这就是集合与束的区别。例如可以构造一个包含两个元素 1, {3,7}的束, 从这个束又可以构造包含两个元素的集合{1,{3,7}}, 用这种方式就构造了一个具有嵌套结构的集合。

集合形成的反运算也是有用的, 如果 *S* 是任意集合, 那么

~*S*　　“*S* 的内容”

是它的内容。例如

~{0,1} = 0,1

幂（power）运算符⚡作用于束, 生成只包含束里面元素的集合。例如，

⚡（0,1） = {*null*},{0},{1},{0,1}

这里 “引进”束运算符来获取集合运算符 $ ∈ ⊆ ∪ ∩ =。有如下公理:

{*A*} ≠ *A*	结构
{~*A*} = *A*	集合形成
~ {*A*} = *A*	“内容”
$ {*A*} = ℭ *A*	“大小”, “基数”
A ∈ {*B*} ＝ *A* : *B*	“元素”
{*A*} ⊆ {*B*} ＝ *A* : *B*	“子集”
{*A*} ∈ ⚡{*B*} ＝ *A* : *B*	“幂”
{*A*} ∪ {*B*} = {*A* , *B* }	“并集”
{*A*} ∩ {*B*} = {*A* ‘*B*}	“交集”
{*A*} = {*B*} ＝ *A*=*B*	“等式”

----------集合论结束

正如束和集合分别是非包装的和包装的集合体, 字符串和表分别是非包装的和包装的序列。可以有集合的集合, 表的表, 但没有束的束和串的串。

2.2 串论

最简单的串是

nil　　空串

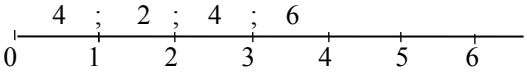
任何数、字符、二元值、集合（甚至以后的表和函数）都是单项串或称项。例如, 数 2 就是一个单项串，或称项。一个项的非空束也是项。多个串可以通过分号连接成一个更长的串，例如，

4 ; 2 ; 4 ; 6

是一个四项串。串的长度可由运算符↔获得。

$$\leftrightarrow (4;2;4;6) = 4$$

可以利用一个串度量尺来度量串长, 如下图所示。



标尺上的数码称作索引。因为每个项总是位于两个索引之间, 所以当从头至尾考察一个串中的项时，如果说现在位于索引 *n* 处, 则清楚表明 *n* 以前的项已经考察过, 而 *n* 以后的项尚未考察过。（如果项是位于某一索引处, 那么说现在位于索引 *n* 处就会产生这样的疑问: 索引为 *n* 的那一项是否已被考察过?）

上图这种表示方法避免了一种混淆, 但也引起了另一个麻烦：即必须通过索引来引用项, 但每一项都有两个不同的索引等距地靠近它。这里采用习惯作法, 尽量避免表达式中使用“+1”或“–1”, 即约定: 一个项的索引等于位于该项之前的项的数目。换句话说, 索引从 0 开始, 如同人的生命从 0 岁开始, 高速公路从 0 英里起步, 等等。索引不是一个任意标号, 它表示在该项之前已有多少项。以 “项 0”, “项 1”, “项 2”，等依此类推的方式来引用串中的项；而不用 “第三项”这种说法, 以避免混淆 “项 2”和 “项 3”。当现在位于索引 *n* 处, 则表明已考察过了 *n* 项, 而项 *n* 将是下一个要考察的。

通过下标运算可获得串中的项。例如,

$$(3; 5; 7; 9\ )_2 = 7$$

通常, S_n 指串 *S* 中的项 *n*。甚至可以取出一整串的项, 例如

$$(3; 5; 7; 9\ )_{2;1;2} = 7; 5; 7$$

如果 *n* 是广义自然数, *S* 是串, 那么 *n***S* 表示将 *n* 个 *S* 连接成一个串。

$$3*(0; 1) = 0; 1; 0; 1; 0; 1$$

如果省略了左操作数, **S* 就表示所有由任意多个 *S* 连接成的串。

$$*(0; 1) = nil,\ 0;1,\ 0;1;0;1, ...$$

如果 *S* 是串，*n* 是 *S* 的索引，且 *i* 是一个项（不一定属于 *S*），那么 $S\lhd n\rhd i$ 表示 *S* 的索引 *n* 的位置换成 *i*。例如

$$(3; 5; 9) \lhd 2\rhd 8 = 3; 5; 8$$

串可以进行相等比较= ≠ 。两个串相等, 必须满足它们具有相同的长度, 并且在每个索引处都有相同的项。如果串里面的项能比较顺序< ≤ > ≥，那么串也可以。两个串的序由它们之间第一个不同的项决定, 例如,

$$3; 6; 4; 7 < 3; 7; 2$$

如果尚未到达不同处已有一串到了结尾, 那么短串位于长串之前。

$$3; 6; 4 < 3; 6; 4; 7$$

这种顺序就是所谓的词典顺序, 用于字典中。

以下是串的语法定义。假设 *i* 是项, *S* 和 *T* 是串, *n* 是广义自然数, 于是

nil	空串
i	项
S ; *T*	“*S* 连接 *T*”
S_T	“*S* 以 *T* 为下标运算”
*n***S*	“*n* 个 *S* 的拷贝”
$S\lhd n\rhd i$	“*S*，但是第 *n* 项是 *i*”

都是串，

$*S$　　“S 的拷贝”

是串束（由串组成的束）, 而

$\leftrightarrow S$　　“S 的长度”

是广义自然数。

以下是串论的公理。在这些公理中 S, T 和 U 是串, i 和 j 是项, n 是广义自然数。

$nil\ ;\ S\ =\ S\ ;\ nil\ =\ S$　　恒等性

$S\ ;\ (T\ ;\ U\ ) = (S\ ;\ T\ )\ ;\ U$　　结合性

$\leftrightarrow nil = 0$　　基

$\leftrightarrow i = 1$　　基

$\leftrightarrow (S;T) = \leftrightarrow S + \leftrightarrow T$

$S_{nil}\ =\ nil$

$\leftrightarrow S<\infty \Longrightarrow (S\ ;\ i\ ;T\ )_{\leftrightarrow S} = i$

$S_{T\ ;U} = S_T\ ;\ S_U$

$S_{(T_U\ )} = \left(S_T\right)_U$

$0*S = nil$

$(n+1)*S = n*S\ ;\ S$

$\leftrightarrow S<\infty \Longrightarrow S\ ;\ i\ ;T \lhd\!\leftrightarrow S\rhd j = S\ ;\ j\ ;T$

$\leftrightarrow S<\infty \Longrightarrow nil \le S < S\ ;\ i\ ;\ T$

$\leftrightarrow S<\infty \Longrightarrow (i < j \Rightarrow S\ ;\ i\ ;\ T <S\ ;\ j\ ;\ U)$

$\leftrightarrow S<\infty \Longrightarrow (i = j\ =\ S\ ;\ i\ ;\ T = S\ ;\ j\ ;\ T)$

也使用以下记号

$x\ ;..\ y$　　“从 x 到 y”（与 $x,..y$ 的读法相同）

其中 x 是整数，y 是广义整数并且 $x \le y$。如同类似的束记号, 它包括 x 但不包括 y, 从而

$$\leftrightarrow (x\ ;..\ y) = y–x$$

以下是关于该记号的公理。

$x;..\ x = nil$

$x;..\ x+1 = x$

$(x;..\ y)\ ;\ (y;..\ z) = x;..\ z$

文本（text）记号是字符表的另一种写法。一段文本由双引号开头, 后跟任意个数的字符（其中若含有双引号，必须重复写两次）, 最后由双引号结尾。以下是一段长度为 15 的文本。

“Don't say ““no””.”

= “D”; “o”; “n”; “’”; “t”; “ ”; “s”; “a”; “y”; “ ”; “““”; “n”; “o”; “””; “.”

空文本 “” 是 nil 的另一种写法。用一个索引表对一段文本进行组合可得到一段子文本。例如

$$\text{“abcdefghij”}_{3;..6} = \text{“def”}$$

以下是一个自描述表达式（自复制自动机（self-reproducing automaton））。

$$\text{“““}_{0;0;(0;..29);28;28;(1;..28)}\text{””” 0;0;(0;..29);28;28;(1;..28)}$$

请实施索引看看能得到什么。

串的连接运算可分配到束的并运算上:

A ; $null$; $B = null$

(A,B) ; $(C,D) = A;C$, $A;D$, $B;C$, $B;D$

因此，一个束串（由束组成的串）等价于一个串束。例如有

$$0;1;2:\ nat;\ 1;\ (0,..10)$$

这是因为 0: *nat*, 1: 1, 而且 2: (0,..10)。只有当一个串的所有项为元素时，这个串也为一个元素（基本束）；因此 0;1;2 是一个元素，但 *nat*;1;(0,..10)不是。扩展到更大的束，

$$0;1;2:\ nat;\ 1;\ (0,..10):\ 3*nat:\ *nat$$

运算符*可分配到束的并运算上, 但仅能运用于左运算元。

$null*A = null$

$(A,B)*C = A*C$, $B*C$

利用这种左分配性, 可以用如下公理定义一个单目*运算:

$$*A = nat*A$$

以上定义的串的索引是自然数且长度是广义自然数。通过增加一种新的运算符，反连接运算符, 可得到索引和长度都为负数的串。这个变化将留作练习 55。

----------串论结束

2.3 表论

表是包装过的串。例如,

$$[0;\ 1;\ 2]$$

是一个三项表, 表括号[ ]可以分配到束的并运算上。

$$[null] = null$$

$$[A,\ B] = [A],\ [B]$$

由于 $0 : nat$ ， $1 : 1$ 和 $2 : 0,..10$, 有

$$[0;\ 1;\ 2]:\ [nat;\ 1;\ (0,..10)]$$

冒号的左边是一个整数表, 右边是一个束表（由束组成的表）, 或等价地说, 是一个表束（由表组成的束）。只有当一个表的所有的项为元素时，一个表是一个元素（基本束）。$[0;1;2]$是一个元素，但$[nat;1;(0,..10)]$不是。进一步发展到大型束, 有

$$[0;\ 1;\ 2]:\ [nat;\ 1;\ (0,..10)]:\ [3*nat]:\ [*nat]$$

以下是表的语法定义。令 *S* 为串, *L* 和 *M* 为表, *n* 为自然数, *i* 为项, 于是，

$[S]$	“包含 *S* 的表”
$L\ M$	“*L M*” 或 “由 *M* 组成 *L*”
$L+M$	“*L* 连接 *M*”
$n \rightarrow i \mid L$	“将表 *L* 的第 *n* 项映射为 *i*，其他项不变”

是表，

$\sim L$	“*L* 的内容”

是串，

$\# L$	“*L* 的长度”

是广义自然数, 而

$L\ n$	“*L n*” 或 “*L* 在索引 *n* 的项”

是项。当然, 圆括号可用于任何表达式, 因此也可写成 *L*(*n*)。如果索引 *n* 不是一个简单数, 必

须用括号括起。当不会产生混淆时, 在书写 *Ln* 时，其间可不留空, 不过当使用多字符名字时, 必须留空。

表的内容是它所包含的项构成的串。

$$\sim [3;5;7;4] = 3;5;7;4$$

表的长度等于它所包含的项的数目。

$$\#\,[3;\ 5;\ 7;\ 4\ ] = 4$$

如同串索引, 表索引从 0 开始。表中的项可以通过毗连该表和一个索引（紧靠着）得到。

$$[3;\ 5;\ 7;\ 4\ ]2 = 7$$

将一个索引表作用于表可得到一个由所选项组成的表。例如

$$[3;\ 5;\ 7;\ 4]\ [2;\ 1;\ 2] = [7;\ 5;\ 7]$$

它称为表组合。表连接可用一个稍许上提的加号 $^{+}$ 表示。

$$[3;\ 5;\ 7;\ 4\ ]+[2;\ 1;\ 2\ ] = [3;\ 5;\ 7;\ 4;\ 2;\ 1;\ 2\ ]$$

记号 $n\rightarrow i\,|\,L$ 作用于表, 其结果是将表 L 的项 n 的值映射为 i。

$$2\rightarrow 22|\ [10;..15] = [10;\ 11;\ 22;\ 13;\ 14\ ]$$

$$2\rightarrow 22\ |\ 3\ \rightarrow\ 33\ |\ [10;..15] = [10;11;22;33;14]$$

令 $L = [10;..15]$, 于是

$$2\ \rightarrow\ L3\ |\ 3\ \rightarrow\ L2\ |\ L = [10;11;13;12;14]$$

表可以进行相等比较$=\neq$。两个表相等, 必须满足它们具有相同的长度, 并且在每个索引处都有相同的项。如果表里面的项能比较顺序$<\ \leq\ >\ \geq$，那么表也可以。采用词典次序, 如同串。

以下是表公理。令 L 是表，令 S 和 T 为串, n 为自然数, i 和 j 为项。

$[S]\neq S$	结构
$[\sim L]=L$	表形成
$\sim[S]=S$	内容
$\#[S] = \leftrightarrow S$	长度
$[S]^{+}[T] = [S;T]$	连接
$[S]n = S_n$	索引
$[S][T] = [S_T]$	组合
$n\rightarrow i\|[S] = [S\triangleleft n\triangleright i]$	修改
$[S] = [T]\ =\ S = T$	等式
$[S]<[T]\ =\ S<T$	序

现在可以证明一些定理, 例如对表 L, M, N 和自然数 n, 有

$(L\ M)\ n = L\ (M\ n)$	
$(L\ M)\ N = L\ (M\ N)$	结合性
$L(M+N) = L\ M+L\ N$	分配性

当一个表被另一个表进行索引时, 可得到一个结果表。例如：

$$[1;4;2;8;5;7;1;4][1;3;7]=[4;8;4]$$

若 M 可以通过一个递增的索引表从表 L 中获得，称 M 为 L 的一个子表。因此[4;8;4]是[1;4;2;8;5;7;1;4]的一个子表。若索引表不仅递增而且连续$[i;..j]$，则该子表又称为段。

若一个表被另一个表索引，则结果是一个表。更一般而言, 串和表可被任意结构索引, 结果会是同一结构。让 *A* 和 *B* 为束，让 *S*，*T*，和 *U* 为串，再有 *L* 是一个表。

$S_{null} = null$ $\quad$ $L\ null = null$

$S_{A;B} = S_A, S_B$ $\quad$ $L\ (A\ ,\ B\ ) = L\ A\ ,\ L\ B$

$S_{\{A\}} = \{S_A\}$ $\quad$ $L\ \{A\} = \{L\ A\}$

$S_{nil} = nil$ $\quad$ $L\ nil = nil$

$S_{T;U} = S_T; S_U$ $\quad$ $L\ (S; T) = L\ S; L\ T$

$S_{[T]} = [S_T]$ $\quad$ $L\ [S\ ] = [L\ S\ ]$

下面是一个有趣的串例子。令 *S*=10;11;12 ，有

$S_{0,\{1,[2;1];0\}}$

$= S_0,\{S_1,[S_2; S_1]; S_0\}$

$= 10,\{11,[12;11];10\}$

下面是一个有趣的表例子。令 *L* = [10;11;12], 有

$L\ (0,\{1,[2;1];0\})$

$= L\ 0,\{L\ 1,[L\ 2\ ;\ L\ 1];\ L\ 0\}$

$= 10,\{11,[12;11];10\}$

2.3.0 多维结构

表可以嵌套。例如, 令

A = [[6;3;7;0];
[4;9;2;5];
[1;5;8;3]]

那么 *A* 是二维数组, 或具体地说是 3 × 4 维数组。形式化描述为 *A*: [3*[4**nat*]]。

给 *A* 一个索引得到一个表

A 1 = [4;9;2;5]

对结果再作一次索引, 得到一个数

A 1 2 = 2

警告: 记号 *A*(1,2)和 *A*[1,2]在许多程序设计语言中用来对二维数组进行索引。不过, 在本书中，

A(1,2) = *A* 1 , *A* 2 = [4;9;2;5] , [1;5;8;3]

A[1,2] = [*A*1 , *A*2] = [[4;9;2;5] , [1;5;8;3]]=[[4;9;2;5]],[[1;5;8;3]]

刚刚看到的是矩形数组, 一个非常规则的结构, 它通过两个索引得到一个数。由表组成的表也可能具有非常不规则的形状, 不但所包含的表长度可以不同, 维数也可以不同。例如, 令

B = [[2;3];4; [5;[6;7]]]

现在 *B* 0 0 = 2, *B* 1 = 4, 但 *B* 1 1 没有定义。这时获取一个数所需的索引数目可以不同。可以用以下方法重新获得一些规则性。令 *L* 为表, *n* 为索引, *S* 和 *T* 为索引串, 那么

L@ *nil* = *L*

L@ *n* = *L n*

L@ (*S ; T*) = *L* @ *S* @ *T*

这样总能用一个单串作“索引”, 该串称为指针, 其作用结果和依次用串中各项作索引的结果

相同。在上述例子中

$$B@(2;1;0) = B\ 2\ 1\ 0 = 6$$

将记号 $S\rightarrow i \mid L$ 一般化, 允许 S 为索引串。其公理为：

$nil \rightarrow i \mid L = i$

$(S;T) \rightarrow i \mid L = S \rightarrow (T \rightarrow i \mid L@S) \mid L$

于是, $S \rightarrow i \mid L$ 为与 L 相似的表, 所不同的是将指针 S 所指向的项映射为 i。例如,

$(0;1) \rightarrow 6 \mid [\ [\ 0\ ;\ 1\ ;\ 2\ ]\ ;$
$[\ 3\ ;\ 4\ ;\ 5\ ]\ ] = [\ [\ 0\ ;\ 6\ ;\ 2\ ]\ ;$
$[\ 3\ ;\ 4\ ;\ 5\ ]\ ]$

----------多维结构结束

----------表论结束

----------基本数据结构结束

第三章　函数理论

在解释规则时, 经常需要发明一些新语法。新语法的一个来源是名字（标识符）, 它的使用规则可通过公理简单地描述出来。通常, 引入名字和公理时, 它们作用于一定的局部区域。读者应该清楚它们的作用域（scope）, 即它们可以作用的区域, 避免在此区域之外使用它们。虽然名字和公理是形式化的（在本书的形式体系中为表达式）, 但到目前为止本书使用句子非形式化地引入它们。但这种非形式化引入的名字和公理的作用域并不总是十分明确。本章将提出一种形式化记号来引入局部名字和公理。

一个变量是一个为对其进行实例化（替代它）而引入的名字。例如，规则 $x\times1=x$ 使用了变量 x 告诉我们任何数乘以 1 还等于那个数。常量是一个不被实例化的名字。例如，引入名字 *pi* 和一些公理，并证明 $3.14<pi<3.15$，但并不是指所有数都在 3.14 和 3.15 之间。类似地，可能引入名字 i 和公理 $i^2=-1$，并且不想对 i 进行实例化。

函数记号表示是一个形式化的方式，它引入局部变量同时也引入一个局部公理，该公理说明什么表达式可用于对该变量进行实例化。

3.0 函数

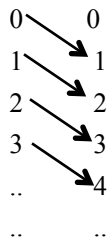
令 v 为一个名字, D 为一个项束（可能使用以前引入过的名字, 但不使用 v），并且令 b 为任意表达式（可以使用以前引入的名字, 也可使用 v）。于是

$\langle v : D \rightarrow b\rangle$　　“映射 D 中的 v 到 b”, “将 D 中的局部名 v 映射到 b”

是变量 v 的函数, 其中 D 为变量的定义域, b 为函数体。包含式 $v : D$ 为函数体 b 中的局部公理。括号⟨ ⟩表示变量的范围和公理。例如

$$\langle n: nat \rightarrow n+1\rangle$$

为自然数上的后继函数。以下是它的映射图:



如果 f 是函数, 那么

$$\Box f \qquad \text{“}f\text{的定义域”}$$

是它的定义域。定义域公理为:

$$\Box\langle v : D \rightarrow b\rangle = \mathrm{D}$$

可以说 D 是函数$\langle v : D \rightarrow b\rangle$的定义域, 也可以说 D 是变量 v 在函数体 b 中的定义域。函数的值域由置换函数体中的变量为其定义域上的每一个元素后所得的元素组成。后继函数的值域为 $nat+1$。

一个函数引入一个变量, 或称参数。引入变量的目的是帮助表示从定义域元素到值域元素的映射。名字的选取并不重要，只要它是新名字，未被使用过即可。换名公理含义如下: 如果 v 和 w 为名字，v 和 w 都不出现在 D 中, 并且 w 也不出现在 b 中, 那么

$$\langle v: D \rightarrow b\rangle = \langle w: D \rightarrow (\text{将 } b \text{ 中的 } v \text{ 置换成 } w)\rangle$$

如果 *f* 是函数, *x* 是其定义域上的元素, 那么

$$f\,x \qquad \text{“}f\text{应用到 }x\text{”, 或 “}x\text{ 的 }f\text{”}$$

是其值域中相应的元素。这就是函数应用, 其中 *x* 是实参。当然, 括号可用于任何表达式, 因此也可以写成 *f*(x)。如果函数或实参不是一个简单值, 必须用括号括起。当不致于引起混淆时,可以写成 *fx*, 中间不留空，但当使用多字符的名字时, 必须在函数和实参间留有空格。以下是函数应用的例子, 如果 $suc = \langle n{:}\ nat \rightarrow n+1 \rangle$, 那么

$$suc\ 3 = \langle n{:}\ nat \rightarrow n+1 \rangle\ 3 = 3+1 = 4$$

于是有应用公理: 如果元素 *x*: *D*, 那么

$$\langle v{:}\ D \rightarrow b \rangle\ x = \text{(置换 } b \text{ 中的 } v \text{ 为 } x\text{)}$$

运算符和函数是类似的；正如将运算符 - 应用于运算元 *x* 得到 $-x$，将函数 *f* 应用于实参 *x* 得到 *f x*。

含有一个以上变量的函数是一个体为函数的函数。这里有两个例子:

$max = \langle x{:}\ xrat \rightarrow \langle y{:}\ xrat \rightarrow \textbf{if}\ x \geqslant y\ \textbf{then}\ x\ \textbf{else}\ y\ \textbf{fi} \rangle \rangle$

$min = \langle x{:}\ xrat \rightarrow \langle y{:}\ xrat \rightarrow \textbf{if}\ x \leqslant y\ \textbf{then}\ x\ \textbf{else}\ y\ \textbf{fi} \rangle \rangle$

如果将函数 *max* 应用到一个实参上, 就可获得一个一元函数,

$$max\ 3 = \langle y{:}\ xrat \rightarrow \textbf{if}\ 3 \geqslant y\ \textbf{then}\ 3\ \textbf{else}\ y\ \textbf{fi} \rangle$$

再将此一元函数应用到另一个实参上就得到一个数。

$$\max\ 3\ 5 = 5$$

谓词是体为二元表达式的函数, 这里有两个例子:

$even = \langle i{:}\ int \rightarrow i/2{:}\ int \rangle$

$odd = \langle i{:}\ int \rightarrow \neg\ i/2{:}\ int \rangle$

关系是体为谓词的函数, 例如:

$divides = \langle n{:}\ nat{+}1 \rightarrow \langle i{:}\ int \rightarrow i/n{:}\ int \rangle \rangle$

$divides\ 2 = even$

$divides\ 2\ 3 = \bot$

另一个函数上的操作, 叫作选择合并（selective union）。如果 *f* 和 *g* 是函数, 那么

$$f \mid g \qquad \text{“}f\text{或者 }g\text{”, “}f\text{和 }g\text{ 的选择合并”}$$

是函数, 当它应用于 *f* 定义域中的实参时, 结果同 *f*, 否则结果同 *g*。其上的公理为

$\Box\,(f \mid g) = \Box f, \Box g$

$(f \mid g)\,x = \textbf{if}\ x : \Box f\ \textbf{then}\ f\,x\ \textbf{else}\ g\,x\ \textbf{fi}$

所有证明规则在作用到函数体时, 都遵循另外一个局部公理: 函数体中的新变量取值于函数的定义域。

3.0.0 简化的函数记号

为了方便和符合传统, 允许函数的记号有一些变化。第一种变化是集中变量说明。例如,

$$\langle x, y{:}\ xrat \rightarrow \textbf{if}\ x \geqslant y\ \textbf{then}\ x\ \textbf{else}\ y\ \textbf{fi} \rangle$$

是前面见到的 *max* 函数的一种简化写法。

如果函数的定义域已在上下文中说明, 那么可以省略定义域说明（以及前面的冒号）。

例如, 后继函数可以写成 ⟨n→n+1⟩, 只要从上下文中可以读出它的定义域是 *nat*。

如果函数体不包含任何变量, 也可省略变量说明（以及其后的冒号）。在这种情况下, 作用域括号⟨⟩也被省略。例如，2→3 是一个将 2 映射到 3 的函数，如果用一个未使用的变量 n，该函数就可写成 ⟨n: *2*→3⟩。

有些人把表达式看作带变量的函数。例如，

$$x+3$$

表达为 x 的函数。他们同时省略了形式化变量和定义域的引入，用非形式化方式表示。这种简化是有问题的。其中有个问题是，有可能一些变量没有在表达式里呈现。例如,

$$\langle x: int \rightarrow \langle y: int \rightarrow x+3 \rangle \rangle$$

使用了两个变量, 可以简写成

$$\langle x: int \rightarrow x+3 \rangle$$

另外一个问题是，它没有明确地指示变量的的作用域。还有另一个问题是，我们不知道变量被引入的顺序，所以我们不能这样化简这个方程。我们认为这个缩写太过精简，而它将不会被使用。我们指出这个问题只是因为它是常见术语，还有表明我们在前面的章节非正式地使用的变量是跟我们在函数里正式地使用的变量是一样的。

----------简化的函数记号结束

3.0.1 作用域和置换

如果变量定义在表达式内部（形式地）, 则它为表达式的局部（local）变量。如果变量定义在表达式之外（不论是形式地还是非形式地）, 则它为该表达式的非局部（nonlocal）变量。这里 “局部”和“非局部”是相对于某一特定表达式或子表达式而言的。

如果对于局部变量总使用新名, 那么置换就是对一个变量的所有出现进行替代。但如果重复使用了一个变量, 在置换时就需更加小心。在下例中, 间隔处代表了无关紧要的部分:

$$\langle x \rightarrow \quad x \quad \langle x \rightarrow \quad x \quad \rangle \quad x \quad \rangle 3$$

变量 x 被说明了两次: 在外层作用域里说明了一次, 又在内层作用域里说明了一次。在内层作用域中，x 是内层作用域上说明的变量。而外层作用域是一个函数，它应用于实参 3。假设 3 属于函数定义域, 根据应用公理，该表达式等价于将式中的 x 置换成 3。不过只能置换外层作用域上说明的 x, 不能置换内层作用域说明的 x。置换后的结果为

$$= \quad (\quad 3 \quad \langle x \rightarrow \quad x \quad \rangle \quad 3 \quad)$$

下面是一个更易混淆的例子。假设 x 是非局部变量, 并且在内层作用域再次说明了它。

$$\langle y \rightarrow \quad x \quad y \quad \langle x \rightarrow \quad x \quad y \quad \rangle \quad x \quad y \quad \rangle x$$

应用公理告诉我们应置换 y 的所有出现为 x。这里 y 的所有三次出现都是外层作用域引入的变量，因此这三次出现都必须用非局部变量 x 来置换。但这样就会将非局部变量 x 插入到一个重新定义 x 为局部变量的内层作用域内，使其看上去像一个局部变量。因此在置换之前，应先对内层应用换名公理。选择新名 z，换名后有

$$= \quad \langle y \rightarrow \quad x \quad y \quad \langle z \rightarrow \quad z \quad y \quad \rangle \quad x \quad y \quad \rangle x$$

再置换 y 后得到

$$= \quad (\quad x \quad x \quad \langle z \rightarrow \quad z \quad x \quad \rangle \quad x \quad x \quad)$$

应用公理（其元素 x: D）

$$\langle v{:}\ D\to b\rangle\ x\ =\ (\text{在 } b \text{ 里面，置换所有 } v \text{ 为 } x)$$

为我们提供置换的形式化记号。它是两个非形式化表达的公理中其中一个（这个是关于变量的引用；另外一个在第 5 章，关于变量去除），因为形式化它相当于写一个程序来执行置换。换名公理可以形式化写成如下：

$$\langle v{:}\ D\to b\rangle\ =\ \langle w{:}\ D\to\langle v{:}\ D\to b\rangle\ w\rangle$$

它不需要成为公理，因为他是扩展公理的一个实例

$$f\ =\ \langle w{:}\ \Box f\to f\ w\rangle$$

当定义域是显而易见的，或者很明显我们想要一个定义域包含 x，我们将“在 b 里面置换 v 为 x”写成 $\langle v\to b\rangle\ x$ 。例如，应用换名公理到两侧的参数 x

$$\langle v\to b\rangle x\ =\ \langle w\to\langle v\to b\rangle w\rangle x$$

说明了在左侧用 x 置换 v 正如在右侧用 w 置换 v，然后用 x 置换 w。

----------作用域和置换结束

----------函数结束

3.1 量词

一个量词是一个应用于函数上的一元前缀运算符。任意二元的、对称的、可结合的运算符都可用来定义量词。这里有四个例子: 运算符 ∧ ∨ + × 可分别用来定义量词 ∀ ∃ ∑ ∏。如果 p 是谓词, 那么全称量（universal quantification）$\forall p$ 为一个二元值，表示将 p 应用到其定义域上的每个元素所得结果的合取值。类似地, 存在量（existential quantification） $\exists p$ 也为一个二元值，表示将 p 应用到其定义域上的每个元素所得结果的析取值。如果 f 是一个值为数的函数, 那么$\sum f$为一个数，表示将 f 应用到其定义域上的每个元素所得结果的和; $\prod f$ 也为一个数，表示将 f 应用到其定义域上的每个元素所得结果的乘积。以下是四个例子。

$\forall\langle r : rat\to r<0 \vee r=0 \vee r>0\rangle$	“对 *rat* 中所有的 r …”
$\exists\langle n{:}\ nat\to n=0\rangle$	“*nat* 中存在一个 n, 满足…”
$\Sigma\langle n{:}\ nat{+}1\to 1/2^n\rangle$	“对 *nat*+1 中的元素 n, 计算…的和”
$\Pi\langle n{:}\ nat{+}1\to(4\times n^2)/(4\times n^2-1)\rangle$	“对 *nat*+1 中的元素 n, 计算...的积”

为了方便和习惯, 允许两种简化的量词写法。第一, 允许省略量词后的作用域括号⟨ ⟩；现在为了表达更清楚，需要把箭头改成一个靠上的圆点。例如可以写成

$$\forall r{:}\ rat\cdot\ r<0\ \vee\ r=0\ \vee\ r>0$$

$$\sum n{:}\ nat{+}1\cdot\ 1/2^n$$

这个简化的量词记号使得变量的作用域没那么清楚了，还使优先级定律更复杂了，但是它有很深厚的数学传统，因此我们会使用它。第二, 允许组合重复的量词后的变量。与其写成

$$\forall x{:}\ rat\cdot\ \forall\ y{:}\ rat\cdot x=y{+}1\Rightarrow x>y$$

可以写成

$$\forall x\ ,\ y{:}\ rat\cdot\ x=y{+}1\Rightarrow x>y$$

而与其写成

$$\sum n{:}\ 0,..10\cdot\ \sum m{:}\ 0,..10\cdot\ n\times m$$

可以写成

$$\sum n,m{:}\ 0,..10\cdot\ n\times m$$

根据定义量词的运算符是否具有幂等性可将量词公理分成两种模式。公理如下: (v 是名

字, A 和 B 是束, b 是二元表达式, n 是数表达式, x 是元素)。

$$\forall\, v{:}\ null\cdot\ b = \mathsf{T}$$
$$\forall\, v{:}\ x\cdot\ b = \langle v{:}\ x \to b\rangle\ x$$
$$\forall\, v{:}\ A\ ,\ B\cdot\ b = (\forall\, v{:}\ A\cdot\ b) \wedge (\forall\, v{:}\ B\cdot\ b)$$

$$\exists\, v{:}\ null\cdot\ b = \bot$$
$$\exists\, v{:}\ x\cdot\ b = \langle v{:}\ x \to b\rangle\ x$$
$$\exists\, v{:}\ A\ ,\ B\cdot\ b = (\exists\, v{:}\ A\cdot\ b) \vee (\exists\, v{:}\ B\cdot\ b)$$

$$\sum v{:}\ null\cdot\ n = 0$$
$$\sum v{:}\ x\cdot\ n = \langle v{:}\ x \to n\rangle\ x$$
$$(\sum v{:}\ A\ ,\ B\cdot\ n\ ) + (\ \sum v{:}\ A\ \text{`}B\cdot\ n\ ) = (\sum v{:}\ A\cdot\ n) + (\sum v{:}\ B\cdot\ n)$$

$$\prod v{:}\ null\cdot\ n = 1$$
$$\prod v{:}\ x\cdot\ n = \langle v{:}\ x \to n\rangle x$$
$$(\prod v{:}\ A\ ,\ B\cdot\ n\ ) \times (\prod v{:}\ A\ \text{`}B\cdot\ n) = (\prod v{:}\ A\cdot\ n) \times (\prod v{:}\ B\cdot\ n)$$

在将自然语言中的 “所有（all）”和“一些（some）”翻译成形式化记号 $\forall$ 和 $\exists$ 时需小心对待。例如, 句子 “所有的没有丢失。（All is not lost.）”不应当译成$\forall\, x\cdot\ \neg\ lost\ x$, 而应译成 $\exists\, x\cdot\ \neg\ lost\ x$, 或者$\neg \forall\, x\cdot\ lost\ x$, 或者$\neg \forall\ lost$。注意当量词应用到一个定义域为空的函数上时, 返回它所基于的运算符的恒等元。人们可能不会对零个数之和为 0 感到奇怪, 但对零个数之积为 1 可能会惊讶。另外，你会认可没有一个元素在定义域为空的函数上具有特性 b（不论 b 是什么）, 因此存在量词应用到空定义域上返回你预料中的那个结果。对空定义域上的所有元素都具有特性 b 这一点可能会很难理解, 但不妨这样设想：如果否认这一说法, 也就是说空定义域中存在一个元素不具有特性 b。但是由于空定义域中不存在这样的元素，没有任何元素是不具有特性 b 的, 因此所有（零）元素都具有特性 b。

也可以从自己定义的函数中构造量词。例如, 函数 min 和 max 是二元、对称、结合和幂等的函数，因此可以定义如下相应的量词 MIN 和 MAX:

$$MIN\ \ v : null\cdot n = \infty$$
$$MIN\ \ v : x\cdot n = \langle v{:}\ x \to n\rangle\ x$$
$$MIN\ \ v : A\ ,\ B\cdot n = min(MIN\ \ v : A\cdot n)\ (\ MIN\ \ v : B\cdot n)$$

$$MAX\ \ v : null\cdot n = -\infty$$
$$MAX\ \ v : x\cdot n = \langle v{:}\ x \to n\rangle\ x$$
$$MAX\ \ v : A\ ,\ B\cdot n = max(MAX\ \ v : A\cdot n)\ (\ MAX\ \ v : B\cdot n)$$

这些定义使 MIN 和 MAX 比传统更实用；例如，

$$MIN\ \ n{:}\ nat\cdot\ 1/(n+1) = 0$$

尽管 0 永远不是函数$\langle n{:}\ nat \to 1/(n+1)\rangle$的结果。

最后介绍一个应用于谓词上的量词。 解量词（solution quantifier）§ （“ …的解”, “那些”）给出一个谓词的解的束。以下是它的公理。

§ *v*: *null*· *b* = *null*

§ *v*: *x*· *b* = **if** ⟨*v*: *x*→*b*⟩ *x* **then** *x* **else** *null* **fi**

§ *v*: *A* , *B*· *b* = (§ *v*: *A*· *b*), (§ *v*: *B*· *b*)

根据求解方程式的知识, 可以很轻松地有，

§ *i*: *int*· $i^2 = 4$ = –2 , 2　“那些在 *int* 中的 *i* 满足...”

方程式是一种特别形式的二元表达式；此外也可以求解任何谓词。例如，

§ *n*: *nat*· *n* < 3 = 0,..3

再一次，为了传统和方便，当解量词在集合里使用，我们可以忽略量词来简化。例如，我们可以写{*n*: *nat*· *n*<3}来代替{§*n*: *nat*· *n*<3}，它是一个标准的集合符号。

关于定义域为 § 量词的结果时各量词的行为还满足更多公理；它们将与量词的其它定律一同列在本书的后面。这些定律在程序设计中将被反复使用, 因此必须努力学习直至熟悉它们。其中一些定律可以用一种更好的方法书写, 虽然未必符合传统。例如, 后面的特定化和普遍化定律表示如果 *x*: *D* ,

∀ *v*: *D*· *b* ⟹ ⟨*v*: *D*→*b*⟩ *x*

⟨*v*: *D*→*b*⟩ *x*⟹ ∃ *v*: *D*· *b*

两者合起来可以写成: 如果 *x*: □*f*

∀ *f* ⟹ *f x* ⟹ ∃ *f*

如果对定义域中的所有元素 *f* 的结果为 T，那么对于定义域中的元素 *x*，*f* 的结果也为 T。而如果对于定义域中的元素 *x*，*f* 的结果为 T，那么就存在定义域中的一个元素使 *f* 的结果为 T。

单点定律表示如果 *x*: *D*，且 *v* 不在 *x* 中出现，那么：

∀*v*: *D*· *v*=*x* ⇒ *b* = ⟨*v*: *D*→*b*⟩ *x*

∃*v*: *D*· *v*=*x* ∧ *b* = ⟨*v*: *D*→*b*⟩ *x*

例如，对于全称量∀*n*: *nat*· *n*=*3*⟹*n*<*10*，其中蕴含的前件是一个变量与一个元素的相等式。根据单点定律，可通过去除量词和前件来简化它，只保留后件，并使用元素替代变量。因此可以得到 3<10，并可进一步简化成 T。在存在量的情况下，需要一个变量与一个元素的相等式作为合取因子，然后可以做相同的简化。例如，∃*n*: *nat*· *n*=*3* ∧ *n*<*10* 简化成 3<10，并进一步简化成 T。若 *P* 是一个不含非局部变量 *x* 的谓词，且元素 *y* 在 *P* 的定义域中，则下面式子都等价：

∀*x*: □*P*· *x*=*y*⟹*Px*

= ∃*x*: □*P*· *x*=*y* ∧ *Px*

= ⟨*x*: □*P*→*Px*⟩ *y*

= *Py*

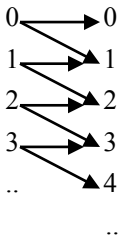
----------量词结束

3.2 函数若干点讨论　　可选节

下面来看一个体为一个束的函数：其定义域中的每个元素映射到值域中的零个或更多元素。例如，

⟨*n*: *nat*→*n*, *n*+1⟩

将每个自然数映射到两个自然数。



函数应用和通常一样适用：

$$\langle n\colon nat \rightarrow n,\ n{+}1\rangle\ 3 = 3, 4$$

一个有时不产生结果的函数称为“部分的”。一个总是至少产生一个结果的函数称为“完全的”。一个总是产生最多一个结果的函数称为“确定性的”。一个有时产生一个以上结果的函数称为“非确定性的”。函数$\langle n\colon nat \rightarrow 0,..n\rangle$是一个部分的非确定性的函数。

一个函数的并应用于一个实参得到结果的并：

$$(f, g)\ x = fx\ ,gx$$

一个函数应用于实参的并得到结果的并：

$$f\ null = null$$
$$f(A, B) = fA, fB$$
$$f(\S\, g) = \S\, y\colon f(\square g)\cdot \exists x\colon \square g\cdot fx{=}y\cdot gx$$

换句话说，函数应用对束的并有分配律。函数f的取值范围是$f(\square f)$。

一般而言，不能使用应用公理将函数应用于一个非基本束。例如，如果定义 $double = \langle n\colon nat \rightarrow n{+}n\rangle$那么可以说

$double(2, 3)$ 这一步是正确的
$= \quad double\ 2, double\ 3$
$= \quad 4, 6$

但不能说

$double(2, 3)$ 这一步是错误的
$= \quad (2, 3){+}(2, 3)$
$= \quad 4, 5, 6$

如果真的想将函数应用于项目的集合体，例如计算该集合体中是否有太多项目。那么这些项目必须用集合加以包装使之成为一个基本的实参。

如果函数体中只使用了一次它的变量，且在满足分配律的上下文中，那么函数可以应用于一个非基本的实参，因为其结果等同于使用分配律后所得的结果。例如，

$\langle n\colon nat \rightarrow n{\times}2\rangle\ (2,3)$ 这一步是可以的
$= \quad (2, 3)\ {\times}2$
$= \quad 4, 6$

3.2.0 函数包含和相等 可选节

如果函数g包含函数f，则根据函数包含公理有:

$$f\colon g\ =\ \square g\colon \square f\ \wedge\ \forall\, x\colon \square g\cdot f\, x\colon g\, x$$

双向利用此公理, 可得到两函数相等的定义:

$$f = g \;=\; \Box f = \Box g \;\wedge\; \forall\, x{:}\ \Box f{\cdot}\ f\,x = g\,x$$

现在来证明 *suc*: *nat*→ *nat*。函数 *suc* 在前面已经定义为 *suc* = ⟨*n*: *nat*→*n*+1⟩ 。函数 *nat*→ *nat* 是函数⟨*n*: *nat*→*nat*⟩的一个简写, 其中含有一个未被使用的变量。该函数是一个非确定的函数, 它将定义域 *nat* 上的每一个元素的结果映射到束 *nat* 上。它也是一个定义域包含 *nat*，值域被 *nat* 包含的函数的束。

suc: *nat*→ *nat* 利用函数包含定律

= *nat*: □*suc* ∧ ∀ *n*: *nat*• *suc n*: *nat* *suc* 的定义

= *nat*: *nat* ∧ ∀ *n*: *nat*• *n*+1: *nat* 自反性，和 *nat* 构造公理

= ⊤

类似地，可以证明有关本章开头一节里定义的其他函数的包含。

max : *rat* → *rat* → *rat*

min : *rat* → *rat* → *rat*

even : *int* → *bin*

odd : *int* → *bin*

divides : (*nat*+1) → *int* → *bin*

更一般地有，

$$f{:}\ A \to B = A{:}\ \Box f \ \wedge\ fA{:}\ B$$

----------函数包含与相等结束

前面使用以下公理定义了 *suc*:

$$suc = \langle n{:}\ nat \to n{+}1\rangle$$

该等式也可写成：

$$\Box suc = nat \wedge \forall n{:}\ nat{\cdot}\ suc\ n = n{+}1$$

还可使用一个更弱的公理定义 *suc*：

$$nat{:}\ \Box suc \wedge \forall n{:}\ nat{\cdot}\ suc\ n = n{+}1$$

该式在实践中很有用，如果需要，*suc* 可以扩展到一个更大的定义域。对于 *max* , *min* , *even* , *odd* , 和 *divides* 等有类似的结论。

3.2.1 高阶函数 可选节

高阶函数是一种形参（parameter）是函数值，并且它的实参（argument）必须是一个函数。如果 *g*: *A*→*B*，那么

$$\langle f{:}\ (A \to B) \to \ldots f \ldots \rangle\ g$$

应用一个高阶函数 *g* 到函数的实参。一个形参表示一个定义域里面的元素，并且应用公理要求实参是一个定义域里的元素，但是函数不是元素。所以我们认为高阶函数应用于实参，如上，应该是

$$\langle f{:}\ \lightning(A \to B) \to \ldots \sim f \ldots \rangle\ \{g\}$$

的简化。幂集运算符 ⚡ 和集合括号 { } 让形参和实参变成元素，正如所要求的，并且内容运算符~解除了集合结构。

以下是一个形参为函数的谓词。

$$\langle f{:}\ ((0,..10) \to int) \to \forall n{:}\ 0,..10{\cdot}\ even\ (f\,n)\rangle$$

该谓词称为 *check*。 *check* 的实参必须是一个定义域包含 0,..10 的函数, 这是因为 *check* 将应

用其实参到 0,..10 中的所有元素上。当应用到前 10 个自然数时，其结果应包含在 *int* 中, 这是因为这些结果还将进一步被测试是否为偶数。*check* 的实参可能具有较大的定义域（额外的定义域元素将被忽略）, 和较小的值域。如果 $A: B$ ， $f: B \to C$ 并且 $C: D$ 那么 $f: A \to D$, 于是，*suc*: (0,..10)→*int* 。可以将 *check* 应用于 *suc*, 结果为 ⊥。

----------高阶函数结束

3.2.2 函数组合　　可选节

令 *f* 和 *g* 为函数, 并且 *f* 不在 *g* 的定义域中(¬ *f*: □*g*), 那么 *g f* 就是 *g* 与 *f* 的组合, 它可由函数组合公理定义如下:

$$\Box\,(g\,f) \;=\; \S\, x{:}\,\Box f \cdot f\,x{:}\,\Box\, g$$

$$(g\,f)\,x \;=\; g\,(f\,x)$$

例如, 由于 *suc* 不在 *even* 的定义域中,

$$\Box\,(even\ suc) = \S\, x{:}\,\Box suc \cdot suc\ x{:}\,\Box even = \S\, x{:}\,nat \cdot x{+}1 : int = nat$$

$$(even\ suc)3 = even\ (suc\ 3) = even\ 4 = \mathsf{T}$$

假设 $x, y: int$, 且 $f, g: int \to int$, 并且 $h: int \to int \to int$ 。那么

	$h\,f\,x\,g\,y$	毗连是左结合的
=	$(\,(\,(h\,f)\,x\,)\,g\,)\,y$	利用函数组合于 $h\,f$
=	$(\,(h\,(f\,x)\,)\,g\,)\,y$	利用函数组合于 $(\,(h\,(f\,x)\,)\,g$
=	$(\,h\,(f\,x)\,)\,(g\,y)$	去掉多余的括号
=	$h\,(f\,x)\,(g\,y)$	

组合公理使得无需括号就可以写出函数和实参的复杂组合。它们可根据各自的功能很好地自我排序。（这称作 “光滑前缀”记法。）

组合和应用是密切关联的。假设 $f: A \to B$ ， $g: B \to C$ 且¬ *f*: □*g*, 于是 *g* 可和 *f* 组合。 虽然 *g* 不可以应用到 *f* 上, 但可将 *g* 改成函数 $g': (A \to B) \to (A \to C)$, 而 g' 可以应用到 *f* 上并获得与组合相同的结果: $g'\,f = g\,f$。下面是一个例子, 定义

$$double = \langle n{:}\,nat \to n{+}n \rangle$$

可以将 *suc* 与 *double* 组合。

	(*double suc*)3	运用组合
=	*double* (*suc* 3)	将 *double* 应用到 *suc*3 上
=	*suc* 3 + *suc* 3	

从 *double* 可以构造一个新函数

$$double' = \langle f \to \langle n \to f\,n + f\,n \rangle \rangle$$

它可以被应用到 *suc* 上

$$(double'\ suc)3 = \langle n \to suc\ n + suc\ n \rangle\ 3 = suc\ 3 + suc\ 3$$

得到与前面相同的结果。组合和应用之间的这一紧密联系引导人们在记号上采用一种捷径: 先将 *double* 应用到 *suc* 上, 尽管此时还不能应用, 然后再将下一个实参分配到 *suc* 的所有出现处。其开头为

(*double suc*) 3	“应用” *double* 到 *suc* 上
(*suc*+*suc*) 3	接着将 3 分配到 *suc* 的所有出现处
suc 3 + *suc* 3	然后得到正确的结果。

如同本例, 这一简写方法通常是有效的, 但必须小心: 有时它会导致不一致性。（“并

置”（apposition）一词可作为“应用”和“组合”的缩写，它也很好地描述了该表示方法！）

正如应用一样，组合对束的并也有分配律。

$f(g,h) = f\,g, f\,h$

$(f,g)h = f\,h, f\,g$

运算符和函数实际上是相似的。它们都应用于 运算元上产生结果。正如对函数进行组合一样，对运算符也可进行组合，还可将运算符与函数进行组合。例如，可以将 – 和函数 f 进行组合, 产出一个数获得一个新的函数。

$$(-f)\,x = -(f\,x)$$

特别地，

$$(-suc)\ 3 = -(suc\ 3) = -4$$

类似地, 如果 p 是谓词, 那么，

$$(\neg\, p)\,x = \neg\,(p\,x)$$

可将 ¬ 与 *even* 组合, 再次得到 *odd*。

$$\neg even = odd$$

还可将对偶定律写成:

$$\neg\,\forall f = \exists\,\neg f$$

$$\neg\,\exists f = \forall \neg f$$

或甚至写成如下两式，就更好了：

$$\neg\,\forall = \exists\,\neg$$

$$\neg\,\exists = \forall \neg$$

----------函数组合结束

----------函数若干点讨论结束

3.3 表作为函数

在特定情况下，表可看成是函数的一种。表 L 的定义域为 0,..#L。相反地，如果一个函数的定义域是 0,..n （n 是自然数），并且它的主体是一个项，那么这个函数有时候可以被视为一种表。索引一个表与函数应用是一样的，并同样记作 $L\ n$。表组合与函数组合一样，也同样记作 $L\ M$。将运算符和表与其它函数组合在一起既方便又无害。例如,

– [3; 5; 2] = [–3; –5; –2]

suc[3; 5; 2] = [4; 6; 3]

也可以将表和其它函数以一种选择合并的方法组合。以函数 1→21 为左运算元，表 [10;11;12] 为右运算元, 可得到，

$$1\rightarrow 21 \mid [10;11;12] = [10;21;12]$$

这正与在表中定义的一样。

量词也可以应用到表上。因为表 L 对应于函数⟨n: 0,..#L → Ln⟩, 那么 ∑L 就等同于 ∑n: 0,..#L· Ln , 它方便地表示了表 L 中的项之和。

在某些情况下，表和函数又是不同的。连接操作和求长度操作可应用于表, 但不可应用

于函数。表上定义了序, 而函数没有。表包含和函数的包含也是不一致的。

----------表作为函数结束

3.4 极限与实数 可选节

令函数为 $f:nat\to rat$，那么 *f0 f1 f2 ...*为一个有理数的序列。函数的极限（或序列的极限）用 $LIMf$ 表示。例如：

$$LIMn:nat\cdot(1+1/n)^n$$

是自然对数的基，经常用 e 表示，大约等于 2.718281828459。使用下面的极限公理来定义 *LIM*：

$$(MAXm\cdot MINn\cdot f(m+n))\le(LIMf)\le(MINm\cdot MAXn\cdot f(m+n))$$

上式所有定义域都为 *nat* 。该公理给出了 *LIM f* 的一个下界和一个上界。当它们相等时，极限公理给出 *LIM f* 的确切值。例如，

$$LIMn\cdot 1/(n+1)=0$$

对某些函数，极限公理说明的内容更少一些。例如，

$$-1\le(LIMn\cdot(-1)^n)\le 1$$

更一般地：

$$(MINf)\le(LIMf)\le(MAXf)$$

对于单调（非递减）函数 f，$LIMf=MAXf$ 。对于反单调（非递增）函数 f，$LIMf=MINf$ 。

现在可以定义广义实数。

$$x:xreal=\exists f:nat\to nat\cdot x=LIMf$$

对所有定义域为 *nat* 和值域为 *rat* 的函数求极限。则实数为：

$$r:real=r:xreal\wedge-\infty<r<\infty$$

对该定义的探索是一个内容丰富的课题，称为实数分析，本书不作讨论。

令 $p:nat\to bin$，*p* 是一个谓词，*p0,p1,p2...*是一个二元表达式的序列。以下公理定义 *p* 的极限：

$$\exists m\cdot\forall n\cdot p(m+n)\Rightarrow LIMp\Rightarrow\forall m\cdot\exists n\cdot p(m+n)$$

其上的定义域均为 *nat*。谓词的极限公理与数值函数的极限公理类似。理解它的一种方式是将它划分成两个分离的蕴含式，并如下所示改变第二个变量：

$$\exists m\cdot\forall i\cdot i\ge m\Rightarrow pi\Longrightarrow LIMp$$

$$\exists m\cdot\forall i\cdot i\ge m\Rightarrow\neg pi\Longrightarrow\neg LIMp$$

对任意特定的（非局部）变量的赋值，第一个蕴含式表示如果在序列 *p0,p1,p2...*中存在一点，经过这点后 *p* 永远为 T，那么 $LIMp$ 为 T。第二个蕴含式表示如果在序列中存在一点，经过这点后 *p* 永远为⊥，那么 $LIMp$ 为⊥。例如，

$$\neg LIMn\cdot 1/(n+1)=0$$

即使 $1/(n+1)$ 的极限为 0，$1/(n+1)=0$ 的极限为⊥。

如果对变量的一些特定的赋值，序列永远不停留于某个二元值，那么对这些赋值而言，公理不能决定 $LIMp$ 的值。

----------极限与实数结束

函数的目的在于引进局部变量。但必须记住任何表达式所关心的变量都是非局部变量。例如，

$$\exists n : nat \cdot x = 2 \times n$$

从上式知 x 是偶数。局部变量 n，也可以是 m 或任何其他的名字，是用于帮助表示 x 为偶数。该表达式所讨论的是 x，而不是 n。

----------函数理论结束

第四章 程序理论

从一个非常简单的计算模型入手。计算机有一块主存储器, 从中可以观察到其中的内容, 或称状态。计算的输入就是提供一个初始状态, 或称前置状态。一段时间之后, 计算的输出就是终结状态，或称后置状态。虽然主存储器内容在物理上可能是位（bits）的串, 但是可将它看成是一个由任意项组成的串, 这只需对若干位进行组合, 并从编码中查看它们。例如, 状态 σ (sigma)可为：

$$\sigma = -2 \; ; \; 15 \; ; \; \text{“A”} \; ; \; 3.14$$

其中一个状态内项的索引通常称为“地址”。由所有可能状态组成的束称作状态空间。例如, 状态空间可以为：

$$int \; ; \; (0,..20) \; ; \; char \; ; \; rat$$

如果主存储处于状态σ, 那么主存储中的项就为 σ_0, σ_1, σ_2, 等等。在引用主存储的项时, 采用不同的名, 诸如 i , n , c , x 等比使用地址更为方便。用来指代状态的项的名字称作状态变量。这里必须说明状态变量是什么，它们的定义域是什么，但不必去说明状态变量对应的地址。形式化地，有一个函数 *address* 来说明每一个状态的变量在那里。例如：

$$x \; = \; \sigma_{address \; \text{“}x\text{”}}$$

一个状态就是对状态变量的一次赋值。

上例中的状态空间是无穷的, 这不现实, 因为任何物理主存储姥姥，都是有限的。允许这种与现实的偏差是为了方便描述。整数理论比整数模 2^{32} 的理论简单, 而有理数理论比 32 位浮点数理论更简单。在任何一种理论设计中, 都必须决定该考虑客观世界的哪些方面，而将其它方面留待别的理论考虑。如果需要，可以自由地开发和使用更为复杂的理论, 但是尽管不考虑物理主存储的有限性, 也将会遇到足够多的困难。

在本章的开始，只考虑比较重要的主存储的前置状态和后置状态。在本章的后部，考虑执行时间和变化的空间要求，在第 9 章将考虑计算过程中的通信和中间状态。但开始时只考虑一个初始输入和一个最终输出。计算的终止的问题是执行时间的问题；终止意味着执行时间是有限的。在一个可终止的计算中，有限时间后可获得最终输出；而在非终止计算中，最终输出是永不可达的，或者说在无限时间上可达。所有关于终止性的讨论将延后到讨论执行时间时进行。

4.0 规范

规范 （specification）是一个二元表达式, 其变量代表感兴趣的量。要说明计算机行为, （此时）感兴趣的量为前置状态σ和后置状态σ′。将前置状态作为输入，接着通过计算产生后置状态作为输出。满足规范的计算过程必须提交一个符合要求的后置状态。换句话说, 所给定的前置状态和计算出的后置状态必须使规范为真。当规范（正确）描述了每一步计算过程，则称为一个实现。如果规范是可实现的, 那么对它的每一个输入状态至少存在一个符合要求的输出状态。

根据对每一输入可得到的符合要求的输出数目, 有以下四个定义。

规范 *S* 对前置状态σ是不可满足的: $\not\exists \; (\S \sigma' \cdot S) < 1$

规范 *S* 对前置状态σ是可满足的: $\not\exists \; (\S \sigma' \cdot S) \geqslant 1$

规范 *S* 对前置状态σ是确定的: $\not\exists \; (\S \sigma' \cdot S) \leqslant 1$

规范 S 对前置状态σ是非确定的: $\not\subset (\S\,\sigma' \cdot S) > 1$

也可将可满足的定义重写为：

规范 S 对前置状态σ是可满足的: $\exists\,\sigma' \cdot S$

最后，

规范 S 是可实现的: $\forall\sigma \cdot \exists\,\sigma' \cdot S$

为方便起见, 可以在书写规范时把一些状态变量的初始值记作 x , y , ...而将其最终值记作 x' , y' , ...（状态变量和其初始值在书写记号上不做区别）。下面是一个例子。假设有两个状态变量 x 和 y, 定义域都是 *int*。那么

$$x' = x+1 \ \wedge \ y' = y$$

说明了一个如下的计算机行为：把 x 的值加 1, 保持 y 的值不变。先来检查一下这一规范是否可实现。根据交换定律（Commutative Laws）变量的次序无关紧要，因此可用$\forall\, x$, y 或$\forall\, y$, x 替换定义中的$\forall\,\sigma$, 用$\exists\, x'$, y' 或$\exists\, y'$, x' 替换$\exists\,\sigma'$ 。于是有

$\forall\, x, y \cdot \exists\, x', y' \cdot \ x' = x+1 \ \wedge \ y' = y$ 应用单点定律两次

$= \ \forall\, x, y \cdot \mathsf{T}$ 应用恒等定律两次

$= \ \mathsf{T}$

该规范是可实现的。它对每一个前置状态都是确定的。

用同样的状态变量, 下面是第二个规范。

$$x' > x$$

要满足这个规范只要在计算过程中将 x 增加任意数量；而 y 保持不变或变成任意的整数。对于每一个初始状态, 该规范都是非确定的。满足前面的规范 $x' = x+1 \ \wedge \ y' = y$ 的计算机行为也同样满足本规范，但满足本规范的行为不一定满足前面的规范。一般而言，越弱的规范越容易实现，越强的规范越难实现。

考虑一个极端情形 T ，它是最容易实现的规范, 因为所有的计算机行为都可满足它。另一个极端情形是⊥, 它不能被任何计算机行为所满足。但是⊥并不是唯一的不可实现的规范, 下面还有一例。

$$x \geq 0 \ \wedge \ y' = 0$$

如果 x 的初始值非负, 只要将变量 y 的值置为 0 就能满足该规范。但当 x 的初始值为负时, 就无法满足这个规范。也许定义者并不想提供一个负的输入, 此时规范应写成：

$$x \geq 0 \Rightarrow y' = 0$$

对于非负初始值 x, 该规范仍要求将变量 y 赋值为 0。如果不提供负值给 x，则不必关心如果提供了负值, 会有什么结果。这一点恰恰体现在该规范中：对于负值的 x，任何结果都符合要求。这就允许实现者在 x 的初始值为负时提供某种错误提示。若希望得到一个特定的错误提示, 则必须强化规范来指明。可以使用 $x \geq 0$ 来描述可接受的输入，而不是计算机行为。可将可接受输入和计算机行为一并描述为 $x \geq 0 \wedge (x \geq 0 \Rightarrow y' = 0)$ ，简化为 $x \geq 0 \wedge y' = 0$ 。但 $x \geq 0 \wedge y' = 0$ 不能作为计算机行为来实现，因为计算机不能控制其输入。

在数学术语和计算术语之间存在不幸的冲突，对这点不得不明白与接受。在数学术语中，变量是一个可被实例化的东西，常量是一个不能被实例化的东西。在计算术语中，变量是能改变状态的东西，常量是不能改变状态的东西。计算变量也被称为“状态变量”，计算常量也被称为“状态常量”。状态变量 x 对应于两个数学上的变量 x 和 x'。状态常量是一个单独的数学变量，它可实例化，但不会改变状态。

4.0.0 规范记号

这里不对规范语言作定义或限制, 允许使用任何易懂的记号。通常会采用应用领域使用的记号。但如果有助于使一个规范表达得更清楚和更易懂, 也可使用新的公理定义和发明新记号。

除了已介绍过的记号, 这里再增加两个。

$ok \quad = \sigma' = \sigma$

$= x' = x \wedge y' = y \wedge ...$

$x := e \quad = \sigma' = \sigma \lhd address\ "x" \rhd e$

$= x' = e \wedge y' = y \wedge ...$

记号 *ok* 表示所有变量的最终值等于其相应的初始值。计算机什么也不做就能满足这个规范。赋值 $x := e$ 称为“ x 被赋值为 e ”，或“ x 得到了 e ”，或“ x 变成了 e ”。在赋值记号中, x 是任意不加撇号的状态变量, e 是 x 定义域中的任意不加撇号表达式。例如, 对整数变量 x 和 y 有，

$$x := x+y \ = x' = x+y \wedge y' = y$$

所以 $x := x + y$ 表明 x 的最终值为 x 和 y 的初始值之和, y 值不变。

规范是二元表达式, 因此它们可以利用二元理论中的任意运算符进行组合。如果 S 和 R 是规范, 那么 $S \wedge R$ 也是规范, 它被任意同时满足 S 和 R 的计算过程满足。类似地, $S \vee R$ 也是规范, 它被任意同时满足 S 和 R 的计算过程满足。同样, $\neg S$ 也是规范, 它被任意不满足 S 的计算过程满足。一个特别有用的运算符是 **if** *b* **then** *S* **else** *R* **fi**, 其中 *b* 是初始状态的二元表达式；它可通过计算机如下实现：首先计算 *b*, 然后根据 *b* 的值, 决定是执行 *S* 还是 *R*。运算符∨和 **if then else fi** 具有一个良好的特性, 那就是如果它们的运算元是可实现的, 那么结果也是可实现的; 而运算符 ∧和¬就不具备这一特性。

规范也可通过相关组合 （dependent composition）连接在一起, 它描述了一种顺序执行过程。如果 *S* 和 *R* 是规范, 那么 *S. R* 也是规范, 它要求计算机首先执行 *S*, 然后执行 *R*, 其中 *S* 的终结状态作为 *R* 的初始状态。（相关组合的符号读作“点”, 它与函数中量词的简化格式用到的那个上提的点不同。）相关组合定义如下。

$S.R = \exists \sigma'' \cdot \langle \sigma' \rightarrow S \rangle \sigma'' \ \wedge \ \langle \sigma \rightarrow R \rangle \sigma''$

$= \exists x'', y'', ... \cdot \langle x', y', ... \rightarrow S \rangle\ x''\ y''\ ...\ \wedge\ \langle x, y, ... \rightarrow R \rangle\ x''\ y''\ ...$

$= \exists\ x'', y'', ... \cdot$ (替换 *S* 中的 x' , y',...为 x'', y'',...) $\wedge$ (替换 *R* 中的 x , y,...为 x'', y'',...)

下面是一个例子。有一个整型变量 *x*, 其规范 $x' = x \vee x' = x+1$ 表示保持 x 的值不变或将 x 的值加 1。若对它自身作组合，有，

$x' = x \vee x' = x+1\ .\ x' = x \vee x' = x+1$

$= \exists\ x'' \cdot (x'' = x \vee x'' = x+1) \wedge (x' = x'' \vee x' = x''+1)$ 将∧分配到∨上

$= \exists\ x'' \cdot x'' = x \wedge x' = x'' \ \vee \ x'' = x+1 \wedge x' = x'' \ \vee \ x'' = x \wedge x' = x''+1 \ \vee \ x'' = x+1 \wedge x' = x''+1$

将∃分配到∨上

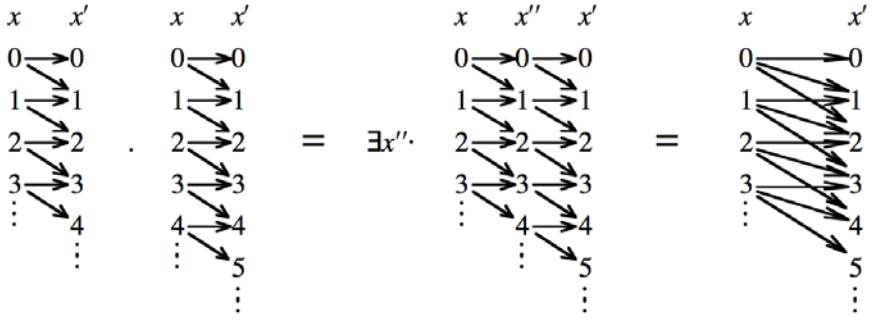
$= (\exists\ x'' \cdot x'' = x \wedge x' = x'') \ \vee \ (\exists\ x'' \cdot x'' = x+1 \wedge x' = x'')$

$\vee$ $(\exists\, x'' \cdot x'' = x \wedge x' = x''+1) \vee (\exists\, x'' \cdot x''=x+1 \wedge x' = x''+1)$ 应用单点定律 4 次

$=$ $x' = x \vee x' = x+1 \vee x' = x+2$

该结果表明如果我们保持 x 不变, 或加 1, 然后再作用一次让 x 不变或加 1, 这种嵌套结果等价于保持 x 不变, 或加 1, 或加 2。

以下是该例的一个示意图。图中 a 到 b 的箭头表示规范允许变量 x 的值从 a 变成 b。图中可以看到如果在相关组合的左操作元中 x 可以从 a 变成 b, 而在右操作元中 x 可以从 b 变成 c, 那么在最后结果里 x 可以从 a 变成 c。



在 $S\,.\,R$ 的定义中，需要明确什么是（置换 S 中的 $x', y',...$为 $x'', y'',...$）和（置换 R 中的 $x, y,...$为 $x'', y'',...$）。首先, 不能因为名字 S 和 R 中没提及任何状态变量而断定置换是不可能的; 而应假设 S 和 R 代表了或者等价于包含一些状态变量的表达式。其次, 当 S 或 R 是赋值运算时, 赋值记号中的状态变量在置换前应当被其等价的数学变量 $x, x', y, y',...$来代替。最后，当 S 和 R 是相关组合时, 内层的置换必须先进行。这里有一个例子, x 和 y 仍然是整型变量。

$x := 3\ .\ y := x+y$ 消去赋值运算

$=$ $x' = 3 \wedge y' = y\ .\ x' = x \wedge y' = x+y$ 消去相关组合

$=$ $\exists x'', y'' : int \cdot\ x'' = 3 \wedge y'' = y \wedge x' = x'' \wedge y' = x''+y''$ 使用单点定律两次

$=$ $x' = 3 \wedge y' = 3+y$

----------规范记号结束

4.0.1 规范定律

下面的定律有的前面已见过。这里 P, Q, R 和 S 是规范, b 是二元值。

$ok.\ P = P.\ ok = P$ 恒等定律

$P.\ (Q.\ R) = (P.\ Q).\ R$ 结合定律

if b **then** P **else** P **fi** $= P$ 幂等定律

if b **then** P **else** Q **fi** $=$ **if** $\neg b$ **then** Q **else** P **fi** 反情况定律

$P =$ **if** b **then** $b \Rightarrow P$ **else** $\neg b \Rightarrow P$ **fi** 情况创建定律

if b **then** S **else** R **fi** $= b \wedge S \vee \neg b \wedge R$ 情况分析定律

if b **then** S **else** R **fi** $= (b \Rightarrow S) \wedge (\neg b \Rightarrow R)$ 情况分析定律

$P \vee Q.\ R \vee S = (P.\ R) \vee (P.\ S) \vee (Q.\ R) \vee (Q.\ S)$ 分配定律

if b **then** P **else** Q **fi** $\wedge R =$ **if** b **then** $P \wedge R$ **else** $Q \wedge R$ **fi** 分配定律

$x :=$ **if** b **then** e **else** f **fi** $=$ **if** b **then** $x := e$ **else** $x := f$ **fi** 函数—命令式定律

在第二个分配定律中，$\wedge$ 可以为任何其他二元运算符。当 b 是前置状态的二元表达式（关于不加撇的变量）时，$\wedge$ 甚至可以替换为相关组合运算符：

if b **then** P **else** Q **fi**. R $=$ **if** b **then** $P.\ R$ **else** $Q.\ R$ **fi** 分配定律

最后，若 e 是前置状态的任何表达式（关于不加撇的变量），有：

$x := e.\ P \ = \ \langle x \rightarrow P \rangle e$ 置换定律

置换定律说明任意规范后的赋值和规范是一样的，只不过被赋值的变量被赋值表达式替换。练习 110 解释了所有困难的情况, 现在来做这个练习。其中状态变量为 x 和 y。

(a) $x{:=}y{+}1.\ y' > x'$

因为 x 在 $y' > x'$ 中不出现, 所以对它进行替换没有变化。

$= \quad y' > x'$

(b) $x{:=}x{+}1.\ y' > x \wedge x' > x$

x 在 $y' > x \wedge x' > x$ 中的两次出现都用 $x{+}1$ 替换。

$= \quad y' > x{+}1 \wedge x' > x{+}1$

(c) $x{:=}\ y{+}1.\ y' = 2 \times x$

由于乘法运算优先于加法, 当置换 $y' = 2x$ 中的 x 为 $y{+}1$ 时必须加上括号。

$= \quad y' = 2 \times (y{+}1)$

(d) $x{:=}1.\ x \geq 1 \Rightarrow \exists x \cdot\ y' = 2 \times x$

在 $x \geq 1 \Rightarrow \exists x \cdot\ y' = 2 \times x$ 中, 第一个 x 的出现是非局部的, 而最后一个是局部的。只有非局部的 x 才被替换。这里的局部变量 x 应当使用任意其它名, 从而避免任何可能的混淆。

$= \ 1 \geq 1 \Rightarrow \exists x \cdot\ y' = 2 \times x$

$= \ even\ y'$

(e) $x{:=}y.\ x \geq 1 \Rightarrow \exists\, y \cdot\ y' = x \times y$

在置换前必须对局部变量 y 换名, 避免置换后将非局部变量 y 置入局部变量 y 的作用域中。

$= \ x := y.\ x \geq 1 \Rightarrow \exists k \cdot y' = x \times k$

$= \ y \geq 1 \ \Rightarrow \ \exists k \cdot\ y' = y \times k$

(f) $x{:=}\ 1.\ ok$

名 ok 由公理 $ok = x' = x \wedge y' = y$ 定义, 所以它与 x 相关。

$= \ x{:=}\ 1.\ x' = x \wedge y' = y$

$= \ x' = 1 \wedge y' = y$

(g) $x{:=}\ 1.\ y := 2$

虽然 x 不出现在 $y := 2$ 中, 但答案不是 $y := 2$。必须记住 $y{:=}2$ 由公理定义, 且它与 x 相关。

$= x := 1.\ x' = x \wedge y' = 2$

$= \ x' = 1 \wedge y' = 2$

（$x' = 1 \wedge y' = 2$ 是否是 $x := 1\ .\ y{:=}2$ 的简化写法值得怀疑。）

(h) $x{:=}\ 1.\ P$ 其中 $P = y{:=}\ 2$

该例结合了(f)和(g)的情况。

$= x' = 1 \wedge y' = 2$

(i) $x := 1.\ y := 2.\ x := x + y$

在(g)中我们知道 $x := 1.\ y := 2 = x' = 1 \wedge y' = 2$ 。如果利用这一点, 则面临相关组合 $x' = 1 \wedge y' = 2.\ x := x + y$ ，无法使用置换定律。因此在一连串赋值运算中, 最好从右向左使用置换定律。

$= x := 1.\ x' = x + 2 \wedge y' = 2$

$= x' = 3 \wedge y' = 2$

(j) $x := 1.$ **if** $y > x$ **then** $x := x + 1$ **else** $x := y$ **fi**

本例没有特别之处。仅说明置换定律可应用于 **if** 语句上。

$=$ **if** $y > 1$ **then** $x := 2$ **else** $x := y$ **fi**

(k) $x := 1.\ x' > x.\ x' = x + 1$

先将置换定律作用到前两个相关组合表达式上, 得到

$= x' > 1.\ x' = x + 1$

再利用相关组合的公理, 得到进一步的简化。

$= \exists x'', y'' \cdot x'' > 1 \wedge x' = x'' + 1$

$= x' > 2$

第一步中避免了的错误是将组合中第三表达式 $x' = x + 1$ 中的 x 用 1 置换。

----------规范定律结束

4.0.2 精化

规范 P 和 Q 等价当且仅当任何时候，只要其中之一满足, 另一个也满足。形式地表示为：

$$\forall \sigma, \sigma' \cdot P = Q$$

如果用户要求实现一个给定规范， 其实可以实现一个与之等价的规范, 用户同样会满意。

假设用户所给的规范是 P，而实现了一个更强的规范 S。因为 S 蕴含了 P, 所有满足 S 的计算机行为也满足 P, 所以用户仍然会满意。因此允许改变用户的规范, 但仅能改成等价的或更强的。

规范 P 被规范 S 精化, 当且仅当任何时候 S 被满足，P 也被满足。

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

规范 P 的精化也就意味着寻找另一个规范 S, 它在任何地方都等于或强于 P。这里称 P 为“问题”, S 为“解”。在实际中, 要证明 P 被 S 精化，必须使用全称量词并证明 $P \Leftarrow S$。在这种上下文环境下, 将 $P \Leftarrow S$ 读成“P 被 S 精化”。

下面是一些精化的例子。

$x' > x \Leftarrow x' = x + 1 \wedge y' = y$

$x' = x + 1 \wedge y' = y \Leftarrow x := x + 1$

$x' \le x \Leftarrow$ **if** $x = 0$ **then** $x' = x$ **else** $x' < x$ **fi**

$x' > y' > x \Leftarrow y := x+1.\ \ x := y+1$

以上各例中的每一问题（左式），对于它所有变量的所有初始值和最终值，都被其相应解（右式）精化（紧接着，被蕴含）。

----------精化结束

4.0.3 条件 可选节

条件是至多表示一个状态的规范。表示初始状态（前置状态）的条件称为初始条件或前置条件；表示终结状态（后置状态）的条件称为终结条件或后置条件。在下面两个定义中，令 P 和 S 为规范。

P 被 S 精化的精确前置条件是 $\forall \sigma' \cdot P \Leftarrow S$

P 被 S 精化的精确后置条件是 $\forall \sigma \cdot P \Leftarrow S$

例如，虽然 $x' > 5$ 不能被 $x := x+1$ 精化，但可以计算（用一个整数变量）：

$(x' > 5$ 被 $x := x+1$ 精化的精确前置条件$)$

$= \forall x' \cdot x' > 5 \Leftarrow (x := x+1)$

$= \forall x' \cdot x' > 5 \Leftarrow x' = x+1$ 单点定律

$= x+1 > 5$

$= x > 4$

这意味着满足 $x := x+1$ 的计算也满足 $x' > 5$，当且仅当它从 $x > 4$ 开始计算。如果仅对前置状态 $x > 4$ 感兴趣，那么利用 $x > 4$ 作前件，可以将问题弱化，获得精化解：

$$x > 4 \Rightarrow x' > 5 \Leftarrow x := x+1$$

对于后置条件也有类似的情况。例如，虽然 $x > 4$ 是不可实现的，

$(x > 4$ 被 $x := x+1$ 精化的精确后置条件$)$

$= \forall x \cdot x > 4 \Leftarrow (x := x+1)$

$= \forall x \cdot x > 4 \Leftarrow x' = x+1$ 单点定律

$= x' - 1 > 4$

$= x' > 5$

这意味着满足 $x := x+1$ 的计算也满足 $x > 4$，当且仅当它以 $x' > 5$ 结束。如果仅对后置条件 $x' > 5$ 感兴趣，那么利用 $x' > 5$ 作前件，也可以将问题弱化，获得精化解：

$$x' > 5 \Rightarrow x > 4 \Leftarrow x := x+1$$

为了更易于理解，可用换位定律（contrapositive law）将规范 $x' > 5 \Rightarrow x > 4$ 重新写成其等价的规范 $x \leq 4 \Rightarrow x' \leq 5$。

现在可以找出 P 被 S 精化的精确前置条件和精确后置条件。任何前置条件，如果它蕴含了精确前置条件，称为充分前置条件；如果它被精确前置条件蕴含，称为必要前置条件。任何后置条件，如果它蕴含了精确后置条件，称为充分后置条件；如果它被精确后置条件蕴含，称为必要后置条件。因此精确前置条件是必要和充分的前置条件，类似地，精确后置条件是必要和充分的后置条件。

练习 131(c)求 $x := x^2$ 的使整数变量 x 远离 0 的精确前置条件和精确后置条件。为了求解，首先必须形式化描述使 x 远离 0 是什么意思：$abs\ x' > abs\ x$（其中 abs 是取绝对值函数；它的

定义在第 11 章中）。现在可以计算，

$(abs\ x' > abs\ x$ 被 $x := x^2$ 精化的精确前置条件)

$= \forall x' \cdot abs\ x' > abs\ x \Leftarrow x' = x^2$ 单点定律

$= abs(x^2) > abs\ x$ 利用 abs 和 x^2 的算术特性

$= x \neq -1 \ \wedge \ x \neq 0 \ \wedge \ x \neq 1$

$(abs\ x' > abs\ x$ 被 $x := x^2$ 精化的精确后置条件)

$= \forall x \cdot abs\ x' > abs\ x \Leftarrow x' = x^2$ 几个步骤之后包括定义域分解、变量转变及 $abs\ x$ 和 x^2 的算术特性

$= x' \neq 0 \ \wedge \ x' \neq 1$

如果 x 的初始值是不为 –1,0 或 1 的任意值, 可以保证 x 将远离 0；如果 x 的最终值是不为 0 或 1 的任意值, 可以保证 x 的确一直远离 0。

令 P 和 Q 为任意规范, C 为前置条件, 且令 C'是相应的后置条件（换句话说, C'与 C 相比，除了在所有的变量上都加有一撇之外，其余都相同）。那么以下是相关定律。

$C \wedge (P.\ Q) \Leftarrow C \wedge P.\ Q$

$C \Rightarrow (P.Q) \Leftarrow C \Rightarrow P.Q$

$(P.\ Q) \wedge C' \Leftarrow P.\ Q \wedge C'$

$(P.Q) \Leftarrow C' \Leftarrow P.Q \Leftarrow C'$

$P.\ C \wedge Q \Leftarrow P \wedge C'.\ Q$

$P.\ Q \Leftarrow P \wedge C'.\ C \Rightarrow Q$

前置条件定律：

C 是 P 被 S 精化的充分前置条件, 当且仅当 $C \Rightarrow P$ 被 S 精化。

后置条件定律:

C'是 P 被 S 精化的充分后置条件, 当且仅当 $C' \Rightarrow P$ 被 S 精化。

----------条件结束

4.0.4 程序

程序是计算机行为的描述或规范。计算机执行一个程序, 就是根据程序完成动作，满足程序要求。人们常常将程序同计算机行为混淆。他们谈论程序“做”什么；然而程序仅仅在纸上或屏幕上, 实际上是执行程序的计算机才能去做些什么。人们会问一个程序是否会“终止”, 程序当然会终止，只有特定的行为才可能不会终止。程序不是行为, 而是行为的规范。更进一步, 计算机可能无法按照一个程序要求执行动作, 其原因有多种：例如，磁头可能坏了，编译程序可能有错误，资源可能用完了（堆栈溢出, 数溢出）等等。所以程序与计算机行为之间是有明显区别的。

程序是计算机行为的规范；就现在来说, 它是一个与前置状态和后置状态有关的二元表达式。并不是每一个规范都是程序。程序是已实现的规范, 也就是已提供了实现的规范, 以便计算机能执行它。本章仅需要非常少的一些程序设计记号，它们与许多流行程序语言中采用的记号类似。它们是：

(a) ok 是程序。

(b)如果 x 是任意状态变量， e 是由初始值组成的已实现二元表达式, 那么 $x := e$ 是程序。

(c) 如果 *b* 是由初始值组成的已实现的二元表达式, 并且 *P* 和 *Q* 是程序, 那么
if *b* **then** *P* **else** *Q* **fi** 是程序。
(d) 如果 *P* 和 *Q* 是程序, 那么 *P*. *Q* 是程序。
(e) 被一个程序精化的可实现的规范是程序。

其中(b)和(c)中提到的“已实现的表达式”是指第 1 章和第二章中出现的表达式：二元值、数、字符、束、集合、字符串、和表, 以及由它们的所有运算符组成的表达式。省去了函数和量词，因为它们很难实现，但是在规范中允许使用函数和量词。

(e)指出：如果已有 *S* 是程序使得 $P \Leftarrow S$ 是定理, 那么任何可实现的规范 *P* 是程序。要执行 *P*, 只需执行 *S*。这个精化就像过程（除了函数，方法）的说明；*P* 作为过程的名, *S* 作为过程的体, 使用名字 *P* 可以进行调用。而且允许使用递归，因为在 *S* 中可能出现对 *P* 的调用。

下面是一个在整型变量 *x* 上的精化实例：

$$x \geq 0 \Rightarrow x' = 0 \Leftarrow \textbf{if } x=0 \textbf{ then } ok \textbf{ else } x := x-1.\ \ x \geq 0 \Rightarrow x' = 0 \textbf{ fi}$$

问题是 $x \geq 0 \Rightarrow x' = 0$ 。解是 **if** x=0 **then** *ok* **else** $x := x-1.\ \ x \geq 0 \Rightarrow x' = 0$ **fi** 。在解中问题又重新出现。根据(e), 如果解是程序, 那么问题也是程序。而如果 $x \geq 0 \Rightarrow x' = 0$ 是程序, 那么解就是程序。因为“允许使用递归”, 因此可通过说明 $x \geq 0 \Rightarrow x' = 0$ 是程序来打破僵局。 允许递归是因为知道如何实现递归。计算机根据解来执行动作 $x \geq 0 \Rightarrow x' = 0$, 每当又遇到同一问题时, 再次根据这个解执行动作。

现在证明上述精化，如下：

if x=0 **then** *ok* **else** x:=$x-1$. $x \geq 0 \Rightarrow x' = 0$ **fi**	替换 *ok* ； 置换定律
= **if** x=0 **then** $x' = x$ **else** $x-1 \geq 0 \Rightarrow x' = 0$ **fi**	使用上下文 $x = 0$ 修改 **then** 部分，使用上下文 $x \neq 0$ 和 x:int 修改 **else** 部分
= **if** x=0 **then** $x \geq 0 \Rightarrow x' = 0$ **else** $x \geq 0 \Rightarrow x' = 0$ **fi**	情况幂等
= $x \geq 0 \Rightarrow x' = 0$	

----------程序结束

规范就像客户和程序员之间签订的合同, 客户要求计算机按某一方式执行, 而程序员编制程序让计算机按预期方式执行。为了这一目的, 规范必须书写得尽可能清楚和容易理解。接着程序员对规范进行精化，获得一个计算机可以执行的程序。有时最清楚的、最明了的规范已经是程序了。在这种情形下，就不需要任何其它的规范, 也不需要精化。不过程序设计记号仅仅是规范记号的一部分：是恰巧可以实现的那部分。规范可以使用任何记号，只要它们有助于使其表述的更清楚, 包括但不限于程序设计记号。

----------规范结束

4.1 程序开发

4.1.0 精化定律

一旦有了规范，可以精化它, 直至得到程序。在进行精化时，只能选择五种程序设计记号。其中的两种 *ok* 和赋值语句已经是程序, 不需要进一步的精化。另外三个则通过提出新的可以进一步精化的问题来解决原来的精化问题。当那些新问题被解决时, 根据精化定律的第一条, 它们的解就可以作为原来问题的解了。

逐步精化法（逐步求精）（单调性,传递性）

如果 $A \Leftarrow$ **if** b **then** C **else** D **fi**, 和 $C \Leftarrow E$ 和 $D \Leftarrow F$ 是定理，

那么 $A \Leftarrow$ **if** b **then** E **else** F **fi** 是定理。

如果 $A \Leftarrow B.\ C$, 和 $B \Leftarrow D$ 和 $C \Leftarrow E$ 是定理, 那么 $A \Leftarrow D.\ E$ 是定理。

如果 $A \Leftarrow B$ 和 $B \Leftarrow C$ 是定理, 那么 $A \Leftarrow C$ 是定理。

逐步精化法可以向最终解引入一个程序设计结构。下一个定律可以将问题以不同的方法分成若干部分处理。

部分精化法（单调性，归并）

如果 $A \Leftarrow$ **if** b **then** C **else** D **fi** 和 $E \Leftarrow$ **if** b **then** F **else** G **fi** 是定理,

那么 $A \wedge E \Leftarrow$ **if** b **then** $C \wedge F$ **else** $D \wedge G$ **fi** 是定理。

如果 $A \Leftarrow B.\ C$ 和 $D \Leftarrow E.\ F$ 是定理，那么 $A \wedge D \Leftarrow B \wedge E.\ C \wedge F$ 是定理。

如果 $A \Leftarrow B$ 和 $C \Leftarrow D$ 是定理, 那么 $A \wedge C \Leftarrow B \wedge D$ 是定理。

当在后几章的保留节目中增加一些程序设计操作符时, 这些新操作符必须遵循类似的逐步精化和部分精化定律。下面是最后一个精化定律。

情况精化法

$P \Leftarrow$ **if** b **then** Q **else** R **fi** 是定理，当且仅当 $P \Leftarrow b \wedge Q$ 和 $P \Leftarrow \neg b \wedge R$ 都是定理。

以下是一个情况精化法的例子, 证明

$$x' \le x \Leftarrow \textbf{if } x=0 \textbf{ then } x' = x \textbf{ else } x' < x \textbf{ fi}$$

可通过证明以下两式来完成:

$$x' \le x \Leftarrow x=0 \wedge x' = x$$

和

$$x' \le x \Leftarrow x \neq 0 \wedge x' < x$$

----------精化定律结束

4.1.1 表求和

作为程序开发的例子, 可以来做练习 169：写一个程序求表中所有数的和。令 L 为数表, s 为数变量, 它的最终值将是 L 中项的和。现在 s 是状态变量, 因而它对应两个数学变量 s 和 s'。由于解不改变表 L, 所以 L 是状态常量（一个数学变量）。

第一步将问题尽可能清楚和简单地进行描述。一个可能的写法是:

$$s := \sum L$$

假设赋值符号右边的表达式还没有实现, 那么这个规范在作进一步精化之前还不是程序。该规范不仅说明 s 具有正确的结果值, 而且说明所有其它变量的值不会改变，这使得实现有一点困难。因此可以选择一个更弱的规范, 使之更容易实现。

$$s' = \sum L$$

该式的算术含义很明显：依次累加表中的每一项, 求出最终的和。为了实现它, 需要一个累加变量, 也使用 s。还需要一个变量作为表的索引, 用于统计已经考察过多少项了;令其为自然数变量 n。一开始, 令 s 和 n 的值都为 0, 表示已经计算过 0 项。然后, 通过将余下的项（也就是项的全部）累加到和上, 从而完成这个求和任务。

$$s' = \sum L \Leftarrow s{:=}0.\ n{:=}0.\ s' = s + \sum L\ [n;..\#L]$$

（记住: 表索引从 0 开始, 表$[n;..\#L]$包括 n 但不包括$\#L$。）这个定理可以通过两次应用置换定律容易地证明。至此就解决了原来的问题, 但又带来了新问题：$s' = s + \sum L[n;..\#L]$。当精化这一新问题时, 必须忽略它出现的上下文环境, 尤其是忽略 $s{=}0 \wedge n{=}0$。这个新规范表示的问题是：对任意的 n, 当前 n 个项已经求和时, 对余下的项求和。n 的一个可能值是$\#L$, 这表示所有的项都被累加过了。这提示出下一步可以利用情况创建定律:

$$s' = s + \sum L[n;..\#L] \Leftarrow \textbf{if } n = \#L \textbf{ then } n = \#L \Rightarrow s' = s + \sum L[n;..\#L]$$
$$\textbf{else } n \neq \#L \Rightarrow s' = s + \sum L[n;..\#L] \textbf{ fi}$$

现在又产生了两个新问题, 不过其中一个是简单问题。

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

另一个问题中尚未累加所有的项$(n \neq \#L)$, 这意味着至少有一个项没有被累加, 所以增加一项到累加和中。为了完成精化, 必须加入所有余下的项。

$$n \neq \#L \Rightarrow\ s' = s + \sum L[n;..\#L] \Leftarrow s{:=}\ s + Ln\ .\ n{:=}n+1.\ s' = s + \sum L[n;..\#L]$$

这个精化可通过两次应用置换定律求证。由于最后一个规范已经被精化, 因此就完成了程序设计。

这里有一点值得注意, 那就是使用 $n \neq \#L$ 来表示项没有被全部累加完。而实际上需要的是 $n<\#L$ 来表示至少有一个项未被累加。在它出现的规范

$$n \neq \#L\ \Rightarrow\ s' = s + \sum L[n;..\#L]$$

中，也使用了记号 $n;..\#L$, 它的定义要求 $n \leq \#L$。因此, 可以认为这一记号的使用隐含了$n \leq \#L$。这个与 $n \neq \#L$ 相结合就可得出所需要的 $n <\#L$。

在第一次精化时，还可使用一个更弱的规范来表示 n 项已被累加, 余项待加。可以用：

$$s' = \sum L\ \Leftarrow\ s{:=}0.\ n{:=}0.\ 0 \leq n \leq \#L \wedge s = \sum L[0;..n] \Rightarrow s' = s + \sum L[n;..\#L]$$

对于那些对前面段落中所包含的隐含信息不太习惯的人，前件的第一部分（$0 \leq n \leq \#L$）使所需的 n 的区间变得明确。而前件的第二部分（$s = \sum L[0;..n]$）在任何地方都没有使用。

当编译器将一个程序转换成机器语言时, 它将每一个精化了的规范当作一个标识符。例如, 上面的累加程序对编译器来说看上去就像:

$A \Leftarrow s{:=}0.\ n{:=}0.\ B$

$B \Leftarrow \textbf{if } n = \#L \textbf{ then } C \textbf{ else } D \textbf{ fi}$

$C \Leftarrow ok$

$D \Leftarrow s{:=}\ s + Ln.\ n{:=}n+1.\ B$

利用逐步精化定律, 编译器可将对 C 和 D 的调用编译成内部（宏扩展）形式，创建出

$B \Leftarrow$ **if** $n = \#L$ **then** *ok* **else** $s := s+Ln.\ n := n+1.\ B$ **fi**

所以, 为了执行的高效, 无需将所有的片段都拼凑在一起, 也无需担心要使用多少次精化。

如果想在计算机上运行这个程序, 必须将它转换成那台计算机上已经实现了的某种程序设计语言。例如, 可以将这个累加程序转换成如下的 C 程序：

```
void B(void){if (n == sizeof(L)/sizeof(L[0])); else {s=s+L[n]; n=n+1; B( ); }}
s=0; n=0; B( );
```

在精化解的最后有一个调用将执行，如这里的 B，该调用可编译为一个分支（或跳转）指令。很多编译器对调用的编译比较糟糕，所以也许使用“go to”会更好，它也将被编译成一个分支指令。

```
s=0; n=0;
B; if (n == sizeof(L)/sizeof(L[0])); else{s=s+L[n]; n=n+1; goto B; }
```

这里不需担心调用，甚至递归调用的低效，因为大多数调用不是转换成什么也不做（内部），就是转换成分支。

----------表求和结束

4.1.2 二的指数幂

现在来做练习 176：给定自然数变量 x 和 y, 不使用指数运算编写一个程序求 $y' = 2^x$。这里给出一个解, 它不是最简单的, 也不是最有效的。只想用它来说明若干问题。

$y' = 2^x \Leftarrow$ **if** $x=0$ **then** $x=0 \Rightarrow y' = 2^x$ **else** $x>0 \Rightarrow y' = 2^x$ **fi**

$x=0 \Rightarrow y' = 2^x \Leftarrow y := 1.\ x := 3$

$x > 0 \Rightarrow y' = 2^x \Leftarrow x>0 \Rightarrow y' = 2^{x-1}.\ y' = 2 \times y$

$x > 0 \Rightarrow y' = 2^{x-1} \Leftarrow x' = x-1.\ y' = 2^x$

$y' = 2 \times y \Leftarrow y := 2 \times y.\ x := 5$

$x' = x-1 \Leftarrow x := x-1.\ y := 7$

第一个精化式将原问题分成两种情况；在第二种情况中 $x \neq 0$, 并且因为 x 是自然数, 所以 $x>0$。在第二个精化式中, 因为 $x=0$, 所以希望得到 $y' = 1$, 这可通过赋值语句 $y := 1$ 得到。而另一个赋值句 $x := 3$ 是多余的, 不用它会使解更简单些, 这里用它只是为了表明规范允许这样做。第三个精化式将 $y' = 2^x$ 分成两步做: 首先做 $y' = 2^{x-1}$, 然后将 y 乘以 2。前件 $x > 0$ 保证了 2^{x-1} 为自然数。第五和第六个精化式再一次包括多余的赋值句。如果没有程序设计理论, 会担心这些多余的赋值句在某些方面导致错误的结果。不过有了理论, 只需证明这六个精化式就能保证执行结果的正确性。

该解的构造使得执行比较困难。可以将其中的非程序设计记号换成单个的字母使程序看起来更熟悉。

$A \Leftarrow$ **if** $x=0$ **then** B **else** C **fi**

$B \Leftarrow y := 1.\ x := 3$

$C \Leftarrow D.\ E$

$D \Leftarrow F.\ A$

$E \Leftarrow y := 2 \times y.\ x := 5$

$F \Leftarrow x := x-1.\ y := 7$

利用逐步精化法可以减少精化式的数目。

$$A \Leftarrow \textbf{if } x=0 \textbf{ then } y:=1.\ x:=3 \textbf{ else } x:=x-1.\ y:=7.\ A.\ y:=2\times y.\ x:=5 \textbf{ fi}$$

可以很容易地将它转换成你身边计算机上的程序设计语言。例如用 C 书写, 程序变为：

```
int x,y;
void A(void){if (x==0) {y=1; x=3;}else{x=x-1; y=7; A( ); y=2*y; x=5;}}
```

可以用不同的 x 值对此程序进行测试。例如执行

x:= 5; A( ); printf (“%i”, y);

将输出 32。但是，比起在没有理论指导下理解这个程序所有可能的执行, 证明精化式要容易得多。

----------二的指数幂结束

----------程序开发结束

4.2 时间

至此, 本书仅讨论了计算的结果, 还未讨论需要多长时间才能得到结果。要讨论时间, 仅需增加一个时间变量。原来的理论不必改变, 新的时间变量被当作与其它变量一样, 作为状态的一部分。状态 $\sigma = t; x; y; ...$现在包括了时间变量 t 以及一些存储变量（memory variable）$x, y,...$。t 作为时间的解释随着使用方式的不同而不同。在实现中, 时间变量不需要申请计算机的主存空间, 它仅表示执行发生的那一时刻。

用 t 表示初始时间, 即执行的开始时刻, 用 t' 表示终结时间, 即执行的结束时刻。为了能表示非终止的情形, 将时间的定义域扩展到包含 ∞ 的数值系统。这个数值系统可以是自然数，或非负实数，视情况而定。

时间不能减少, 因此带有时间的规范 S 是可实现的当且仅当

$$\forall \sigma \cdot \exists \sigma' \cdot S \wedge t' \geq t$$

对每一个初始状态, 必须至少存在一个符合条件的终结状态, 其时间是不减少的。

有许多方法可以度量时间。这里仅介绍两种: 真实时间和递归时间。

4.2.0 真实时间

在真实时间度量中，时间变量 t 的类型为扩展非负实数（extended nonnegative real）。真实时间有利于度量实际的执行时间。这对于某些应用, 如对化学或核反应的控制, 是最基本的。它的缺点是必须了解内部的实现知识（硬件和软件）。

要获得程序的真实时间, 对程序作如下修改：

* 将赋值句 $x := e$ 替换成

$$t := t + (\text{计算和存储 } e \text{ 的时间}).\ x := e$$

* 将每一个条件句 **if** b **then** P **else** Q **fi** 替换成

$$t := t + (\text{计算 } b \text{ 及其分支所需的时间}).\ \textbf{if } b \textbf{ then } P \textbf{ else } Q \textbf{ fi}$$

* 将每一个对 P 的调用替换成

$$t := t + (\text{调用和返回所需的时间}).\ P$$

对于“内部（in-line）”实现的调用, 调用时间为 0。对于精化解的最后一个调用, 它就是执行分支所需的时间。有时它等于将返回地址压入堆栈并执行分支, 再加上从堆栈中弹出地址及从分支返回所需的时间。

* 每一个被精化的规范都可以包括时间。例如, 令 f 为函数, 其初始状态为 σ , 那么

$$t' = t+f\,\sigma$$

表示 $f\,\sigma$ 为执行时间,

$$t' \le t+f\,\sigma$$

表示 $f\,\sigma$ 为执行时间的上界, 而

$$t' \ge t+f\,\sigma$$

表示 $f\,\sigma$ 为执行时间的下界。

也可以在每一个程序设计记号之后而不是之前, 将时间递增。不过将它放在前面, 会使置换定律更容易使用。

在 4.0.4 节里, 考虑过如下例子

$$P \Leftarrow \textbf{if } x{=}0 \textbf{ then } ok \textbf{ else } x{:=}\, x{-}1.\ P \textbf{ fi}$$

假定 **if** 句、赋值句和调用都占用一个时间单位。上面的精化式变为

$$P \Leftarrow t{:=}\, t{+}1.\ \textbf{if } x{=}0 \textbf{ then } ok \textbf{ else } t{:=}\, t{+}1.\ x{:=}\, x{-}1.\ t{:=}\, t{+}1.\ P \textbf{ fi}$$

当 P = **if** $x \ge 0$ **then** $x' = 0 \wedge t' = t+3x+1$ **else** $t' = \infty$ **fi**

时, 上述精化式就为定理。当 x 从一个非负值开始时，该程序的执行总是将 x 置成 0, 此执行需要的时间是 $3\times x+1$; 当 x 从一个负值开始时, 执行时间为无穷，并且对 x 的终止值不作说明。这是对该计算的一个合理的说明。

同样的精化式

$$P \Leftarrow t{:=}\, t{+}1.\ \textbf{if } x{=}0 \textbf{ then } ok \textbf{ else } t{:=}\, t{+}1.\ x{:=}\, x{-}1.\ t{:=}\, t{+}1.\ P \textbf{ fi}$$

对于 P 的其他定义也是一个定理，包括下面的三个定义：

P = $x'=0$

P = **if** $x\ge 0$ **then** $t'=t+3\times x+1$ **else** $t'=\infty$ **fi**

P = $x'=0 \wedge$ **if** $x\ge 0$ **then** $t'=t+3\times x+1$ **else** $t'=\infty$ **fi**

第一个定义忽略了时间，第二个忽略了结果。如果能对第一个和第二个定义的精化进行证明，根据部分精化法定律，第三个定义的精化也顺带被证明了。第三个定义说明程序的执行总是将 x 置为 0；当 x 从一个非负值开始时，将花费 $3\times x+1$ 时间；当 x 从负值开始时，将花费无限时间。为得到如 $x'=0$ 这样的结果要花费无限时间，这听上去有点奇怪。其实需要无限时间得到一个结果就意味着这个结果是永远不能得到的。采取这种奇怪方式的原因在于可以将证明划分成两个部分：结果和时间，然后不费吹灰之力就可得到二者的合取。因此如果一个规范说明变量的值在无限时间取得，那么可以忽略它。

更奇怪的事情是在无穷时间处讨论变量的值。考虑精化式

$$Q \Leftarrow t{:=}\, t{+}1.\ Q$$

它的三个使上式为定理的可实现的规范为：

Q = $t' = \infty$

Q = $x' = 2 \wedge t' = \infty$

$Q = x' = 3 \wedge t' = \infty$

第一个看上去比较合理，而后两个可看到 x 的“最终”值可以是 2, 也可以是 3。但因为 $t' = \infty$, 其实也可以说在以上两种情形下，永远得不到结果。

----------真实时间结束

4.2.1 递归时间

递归时间度量比真实时间度量更为抽象。它并不度量实际的执行时间。其优点在于不必了解实现细节。在递归时间度量中，时间变量 t 的类型为 *xnat*，且

* 每一次递归调用花费一个时间单位

* 其余执行不花时间

这种度量方法忽略了“直线”和“分支”程序的执行时间, 只考虑循环时间。

在这种递归度量方法下, 前面的例子变成

$$P \Leftarrow \textbf{if } x = 0 \textbf{ then } ok \textbf{ else } x := x - 1.\ t := t + 1.\ P \textbf{ fi}$$

对于 P 的一些不同的定义，上式为定理，例如以下两个定义：

$$P = \textbf{if } x \geq 0 \textbf{ then } x' = 0 \wedge t' = t + x \textbf{ else } t' = \infty \textbf{ fi}$$

$$P = x' = 0 \wedge \textbf{ if } x \geq 0 \textbf{ then } t' = t + x \textbf{ else } t' = \infty \textbf{ fi}$$

其执行时间在真实时间度量中对于非负的 x 为 $3 \times x + 1$，而在递归时间度量中还是 x。递归时间度量比真实时间度量告诉我们更少的信息，它只用 x 表示了执行时间的线性增长，而没有具体的倍数和增加常量。

这是一个直接递归的例子, 即问题 P 被一个包含对 P 的调用的解所精化。递归也可以是间接的。例如, 问题 A 可以被包含调用 B 的解精化, 而 B 的精化解包含了对 C 的调用，C 的精化解又包含了对 A 的调用。那么在间接递归中，哪些调用是递归的？所有的？还是只有一个？哪一个？回答是对于递归时间度量，这些并不重要；可能会有常数时间的影响，但时间表达式的形式不会改变。递归时间的一般规则是：

•在每一次调用的循环中, 至少有一个单位时间的时间增长。

----------递归时间结束

下面来证明一个带有时间的精化式((练习 146(b)) :

$$R \Leftarrow \textbf{if } x = 1 \textbf{ then } ok \textbf{ else } x := div\ x\ 2.\ t := t + 1.\ R \textbf{ fi}$$

其中 x 是整数变量, 并且

$$R = x' = 1 \wedge \textbf{if } x \geq 1 \textbf{ then } t' \leq t + log\ x \textbf{ else } t' = \infty \textbf{ fi}$$

为了更有效地使用部分精化定律，将 **if then else fi** 写成合取的形式。

$$R = x' = 1 \wedge (x \geq 1 \Rightarrow t' \leq t + log\ x) \wedge (x < 1 \Rightarrow t' = \infty)$$

练习中使用了函数 *div*（整除且余数舍去）和 *log*（以二为底的对数）。这个程序的执行总是将 x 置成 1；当 x 从正值开始时它的执行需要对数时间完成；当 x 从非正值开始时, 它需要无穷时间。利用部分精化法，足以分别验证 R 的三个合取式：

$x' = 1 \Leftarrow$ **if** $x = 1$ **then** *ok* **else** $x := div\ x\ 2.\ t := t + 1.\ x' = 1$ **fi**

$x \geq 1 \Rightarrow t' \leq t + log\ x \Leftarrow$ **if** $x = 1$ **then** *ok*

else $x := div\ x\ 2.\ t := t + 1.\ x \geq 1 \Rightarrow t' \leq t + log\ x$ **fi**

$x < 1 \Rightarrow t' = \infty \Leftarrow$ **if** $x = 1$ **then** *ok* **else** $x := div\ x\ 2.\ t := t + 1.\ x < 1 \Rightarrow t' = \infty$ **fi**

利用置换定律可重写以上三式:

$x' = 1 \Leftarrow$ **if** $x = 1$ **then** $x' = x \wedge t' = t$ **else** $x' = 1$ **fi**

$x \geq 1 \Rightarrow t' \leq t+log\ x \Leftarrow$ **if** $x = 1$ **then** $x' = x \wedge t' = t$

else $div\ x\ 2 \geq 1 \Rightarrow t' \leq t+1+log\ (div\ x\ 2)$ **fi**

$x<1 \Rightarrow t' = \infty \Leftarrow$ **if** $x=1$ **then** $x' = x \wedge t' = t$ **else** $div\ x\ 2 < 1 \Rightarrow t' = \infty$ **fi**

现在利用情况精化定律, 将上面三式每个一分为二。必须证明

$x' = 1 \Leftarrow x = 1 \wedge x' = x \wedge t' = t$

$x' = 1 \Leftarrow x \neq 1 \wedge x' = 1$

$x \geq 1 \Rightarrow t' \leq t+log\ x \Leftarrow x = 1 \wedge x' = x \wedge t' = t$

$x \geq 1 \Rightarrow t' \leq t+log\ x \Leftarrow x \neq 1 \wedge (div\ x\ 2 \geq 1 \Rightarrow t' \leq t+1+log\ (div\ x\ 2))$

$x<1 \Rightarrow t' = \infty \Leftarrow x = 1 \wedge x' = x \wedge t' = t$

$x<1 \Rightarrow t' = \infty \Leftarrow x \neq 1 \wedge (div\ x\ 2 < 1 \Rightarrow t' = \infty)$

依次证明以上六式。首先

$(x' = 1 \Leftarrow x = 1 \wedge x' = x \wedge t' = t)$ 利用传递性和特定化

$= \mathsf{T}$

其次

$(x' = 1 \Leftarrow x \neq 1 \wedge x' = 1)$ 利用特定化

$= \mathsf{T}$

再次

$(x \geq 1 \Rightarrow t' \leq t+log\ x \Leftarrow x = 1 \wedge x' = x \wedge t' = t)$

利用移动定律（Law of Portation）的第一条将初始前件移到解这一边, 使它成为一个合取式

$= t' \leq t+log\ x \Leftarrow x = 1 \wedge x' = x \wedge t' = t$ 注意 $log\ 1 = 0$

$= \mathsf{T}$

接着是六式中最难的一个,

$(x \geq 1 \Rightarrow t' \leq t+log\ x \Leftarrow x \neq 1 \wedge (div\ x\ 2 \geq 1 \Rightarrow t' \leq t+1+log\ (div\ x\ 2)))$

再次利用移动定律的第一条, 将初始前件移到解这一边, 变成一个合取式

$= t' \leq t+log\ x \Leftarrow x>1 \wedge (div\ x\ 2 \geq 1 \Rightarrow t' \leq t+1+log\ (div\ x\ 2))$

因为 x 是整数, $x>1 = div\ x\ 2 \geq 1$, 利用抛弃定律（Law of Discharge）的第一条

$= t' \leq t+log\ x \Leftarrow x>1 \wedge t' \leq t+1+log\ (div\ x\ 2)$

利用移动定律的第一条, 将 $t' \leq t+1+log\ (div\ x\ 2)$ 移到左边

$= (t' \leq t+1+log\ (div\ x\ 2) \Rightarrow t' \leq t+log\ x) \Leftarrow x>1$

根据连接定律（Connection Law)）$(t' \leq a \Rightarrow t' \leq b) \Leftarrow a \leq b$

$\Leftarrow t+1+log\ (div\ x\ 2) \leq t+log\ x \Leftarrow x>1$ 两边同时减去 1

$= t + log\ (div\ x\ 2) \leq t + log\ x - 1 \Leftarrow x>1$ 对数定律

$= t + log\ (div\ x\ 2) \leq t + log\ (x/2) \Leftarrow x>1$ log 和+对 $x>0$ 单调

$\Leftarrow div\ x\ 2 \leq x/2$ div 先做/ 运算, 然后舍去小数部分

$= \mathsf{T}$

下一个式子很简单：

$(x < 1 \Rightarrow t' = \infty \Leftarrow x = 1 \wedge x' = x \wedge t' = t)$ 移动定律

$= t' = \infty \Leftarrow x < 1 \wedge x = 1 \wedge x' = x \wedge t' = t$ 把 $x<1 \wedge x=1$ 放在一起，基定律第一条

$= t' = \infty \Leftarrow \bot$ 基定律最后一条

$= \mathsf{T}$

最后一个式子：

$(x<1 \Rightarrow t' = \infty \Leftarrow x \neq 1 \wedge (div\ x\ 2<1 \Rightarrow t' = \infty))$ 移动定律

$= t' = \infty \Leftarrow x < 1 \wedge (div\ x\ 2 < 1 \Rightarrow t' = \infty)$ 抛弃定律

$= t' = \infty \Leftarrow x < 1 \wedge t' = \infty$ 特定化

$= \mathsf{T}$

至此完成整个证明。

4.2.2 终止问题

规范是需要软件的客户和提供软件的程序员之间的契约。如果执行程序时，客户发现程序行为与规范相悖就可以抱怨程序员违反了契约。

下面是四个规范，它们都说明变量 x 有终止值 2。

(a) $x' = 2$

(b) $x' = 2 \wedge t' < \infty$

(c) $x' = 2 \wedge (t < \infty \Rightarrow t' < \infty)$

(d) $x' = 2 \wedge t' \leq t + 1$

规范(a)没有表示出什么时间得到想要的最终值。它可被精化成包含递归时间的下式：

$$x' = 2 \Leftarrow t := t+1.\ x' = 2$$

该无限循环在时间∞上提供 x 的终止值，或者说它永远不会提供 x 的终止值。这也许不是一个好的精化，但客户无法抱怨。只有当程序产生结果 $x' \neq 2$ 时客户才能抱怨，但那是永远不会发生的。

为了避免这种情况，客户也许会要求规范(b)，因为它说明终止值会在有限时间得到。程序员会拒绝规范(b), 因为它是不可实现的: (b) $\wedge\ t' \geq t$ 在 $t = \infty$ 时是不可满足的。这看上去似乎有点奇怪：仅仅因为当从 ∞ 时间开始计算时这个时间不递减的规范不可满足就拒绝它。毕竟，客户并不想从∞时间处开始进行计算。但是如果客户使用的软件是在相关（顺序）组合中跟随在一个无限循环的后面，那么计算就的确从∞时间处开始（换句话说, 它永远不会开始）, 而在计算开始之前我们不能期望它结束。 一个可实现的规范必须对任何初始状态在时间不减的条件下都可满足, 甚至初始时间为∞也不例外。

因此客户尝试使用规范(c)。这表示如果计算在一个有限时间上开始，则必须在有限时间结束。该规范是可实现的，但令人惊奇的是，它可被与(a)同样的结构精化！包含递归时间，精化如下：

$$x' = 2 \wedge (t < \infty \Rightarrow t' < \infty) \Leftarrow t := t+1.\ x' = 2 \wedge (t < \infty \Rightarrow t' < \infty)$$

客户可能还是不满意，但仍然无法抱怨。只有当程序产生结果 $x' \neq 2$ 或永远计算不停止时客户才能抱怨。但不存在客户抱怨永远计算的时间，因此(a)和(c)的情况是相同的。理论中精确地反映了这一情况，它允许(a)和(c)被同样的结构精化。

最后，客户将规范改成了(d)，以秒为单位测量时间。现在如果计算产生 $x' \neq 2$ 或计算超过一秒，客户就可以抱怨。无限循环不再可能，因为：

$$x' = 2 \wedge t' \leq t + 1 \Leftarrow t := t + 1 . x' = 2 \wedge t' \leq t + 1$$

不是一个定理。因此如下精化：

$$x' = 2 \wedge t' = t \Leftarrow x := 2$$

规范(d)给出了时间界限，于是更多情况可抱怨，也就导致了更少的精化。计算在时间限制内提供给客户预期的结果。

当且仅当程序行为与规范相悖时客户才能抱怨。因此，没有设置实际时间界限的规范是无意义的。

----------终止问题结束

4.2.3 可靠性（Soundness）与完备性 可选节

本书中的程序设计理论在以下意义下是可靠的。令 P 是一个可实现的规范。若能证明下面的精化：

$P \Leftarrow$ (一些可能包含对 P 的递归调用的东西)

则相应的计算（在有限时间内）不会与 P 相矛盾。

理论在以下意义下是不完备的。即使 P 是一个可实现的规范，且与以下精化

$P \Leftarrow$ (一些可能包含对 P 的递归调用的东西)

相应的计算（在有限时间内）不会与 P 矛盾，精化可能无法证明。但在那种情况下，存在另一可实现的规范 Q，使得下列精化：

$P \Leftarrow Q$

$Q \Leftarrow$ (一些可能包含对 Q 的递归调用的东西)

都可证明，且除了从 P 到 Q 的变化外，精化 Q 与先前不能被证明的精化 P 是等同的。在这种更弱的意义下，理论是完备的。在更强的意义下，不存在一个既可靠又完备的程序设计理论。

----------可靠性与完备性结束

4.2.4 线性查找

练习 180: 写一个程序, 寻找给定项在给定表中的首次出现。执行时间必须随表长呈线性增长。

令表为 L, 要查找的值为 x（这些不是状态变量）。如果 x 存在于表 L 中，程序将给出 x 在表 L 中首次出现的索引, 用自然数变量 h（代表“在这里”（*here*））表示。如果 x 不存在于表 L 中, 那么“首次出现”没有定义; 通过将 h 赋值为 L 的长度, 可以方便地表示 x 不在 L 中。 规范表示为:

$$\neg x : L(0,..h') \wedge (Lh' = x \vee h' = \#L) \wedge t' \leq t + \#L$$

首先考虑h'的规范部分，时间稍后考虑。想法显然是这样的：依次查看表中的每一项, 从项 0 开始, 直到找到 x 或者全部项都被查找完。从项 0 开始，可作以下精化：

$\neg x : L(0,..h') \wedge (Lh' = x \vee h' = \#L) \Leftarrow$

$h := 0.\ h \le \#L \ ® \ \neg x : L\ (h,..h2)\ э\ (Lh2 = x\ (\ h2 = \#L)$

新问题和原问题很相象，除了表示从索引 h 开始，使得 $0 \le h \le \#L$ 成立的任何 h，而不是从索引 0 开始。因为 h 是自然数变量，可以不必写 $0 \le h$，但也可以写。在描述搜索过程的剩余问题时，需要将开始的索引一般化。可以不做任何事满足 $\neg x : (h,..h')$，因为这样 $h' = h$ 和表段为空。要得到 $Lh' = x \vee h' = \#L$，需要测试 $Lh = x$ 或 $h = \#L$。为了测试 $Lh = x$ 需要知道 $h < \#L$，因此必须首先测试 $h = \#L$。

$h \le \#L \Rightarrow \neg x : L(h,..h') \wedge (Lh' = x \vee h' = \#L) \Leftarrow$

if $h = \#L$ **then** *ok* **else** $h < \#L \Rightarrow \neg x : L(h,..h') \wedge (Lh' = x \vee h' = \#L)$ **fi**

在余下的问题中就可以测试 $Lh = x$。

$h < \#L \Rightarrow \neg x : L(h,..h') \wedge (Lh' = x \vee h' = \#L) \Leftarrow$

if $Lh = x$ **then** *ok* **else** $h := h+1.\ h \le \#L \Rightarrow \neg x : L(h,..h') \wedge (Lh' = x \vee h' = \#L)$ **fi**

现在考虑时间:

$t' \le t + \#L \Leftarrow h := 0.\ h \le \#L \Rightarrow t' \le t + \#L - h$

$h \le \#L \Rightarrow t' \le t + \#L - h \Leftarrow$ **if** $h = \#L$ **then** *ok* **else** $h < \#L \Rightarrow t' \le t + \#L - h$ **fi**

$h < \#L \Rightarrow t' \le t + \#L - h \Leftarrow$ **if** $Lh = x$ **then** *ok*

else $h := h+1.\ t := t+1.\ h \le \#L \Rightarrow t' \le t + \#L - h$ **fi**

部分精化法指出如果两个规范可以使用相同的精化结构, 那么它们的合取式也可使用这个结构。如果把 $t := t+1$ 加到不考虑时间的精化式中，不会影响证明，那么规范 $\neg x : L(0,..h') \wedge (Lh' = x \vee h' = \#L)$ 和 $t' \le t + \#L$ 就使用了相同的精化结构, 所以该结构对两者的合取式也有效, 原问题因此得以解决。同样也可以将 $\neg x : L(0,..h') \wedge (Lh' = x \vee h' = \#L)$ 划分成部分。当然必须证明这些精化。

在程序设计中没有必要考虑如此细小的步骤。可以写成

$\neg x : L(0,..h') \wedge (Lh' = x \vee h' = \#L) \ \wedge t' \le t + \#L \Leftarrow$

$h := 0.\ h \le \#L \ \Rightarrow \neg x : L(h,..h') \wedge (Lh' = x \vee h' = \#L) \ \wedge t' \le t + \#L - h$

$h \le \#L \Rightarrow \neg x : L(h,..h') \wedge (Lh' = x \vee h' = \#L) \ \wedge t' \le t + \#L - h \Leftarrow$

if $h = \#L$ **then** *ok*

else if $Lh = x$ **then** *ok*

else $h := h+1.\ t := t+1.\ h \le \#L \Rightarrow \neg x : L(h,..h') \wedge (Lh' = x \vee h' = \#L) \ \wedge t' \le t + \#L - h$ **fi fi**

但是现在，假定已知给定的表 L 非空。利用这一新信息, 可将上述第一个精化式重写成：

$\neg x : L(0,..h') \wedge (Lh' = x \vee h' = \#L) \wedge t' \le t + \#L \Leftarrow$

$h := 0.\ h < \#L \Rightarrow \neg x : L(h,..h') \wedge (Lh' = x \vee h' = \#L) \ \wedge t' \le t + \#L - h$

就这样，如果求解跨步不是太大的话,这个新问题就已经被解决了。（注意: 如果使用了递归时间度量, 按这种方式重写第一个精化式就没有任何优越性。如果使用真实时间, 就稍微有优势些）。习惯上, 将有关常量的信息只写一遍, 而不是在每个规范中都写一遍。例如, 在上例中可以说明$\#L>0$一次, 以便证明精化式时使用, 而不必在每个规范中都重复说明它。

有时可以采用一种称为标记 （Sentinel）的技术来减少执行时间（真实时间）。 方法是： 将表 L 作为变量, 以便可以在其末尾连接一个值。如果这样做代价不高的话，可将 x 连到 L 的末尾。因而保证搜索一定可以找到 x, 这样每次循环中都要进行的测试 $h=\#L$ 就可以省去。原程序忽略时间变成：

$\neg x : L(0,..h') \wedge (Lh' = x \vee h' = \#L) \Leftarrow L := L+[x].\ h := 0.\ Q$

$Q \Leftarrow$ **if** $Lh = x$ **then** *ok* **else** $h := h+1.\ Q$ **fi**

其中 $Q = L\,(\#L - 1) = x \wedge h < \#L \Rightarrow L' = L \wedge \neg x : L(h,..h') \wedge Lh' = x$ 。

----------线性查找结束

4.2.5 二分查找

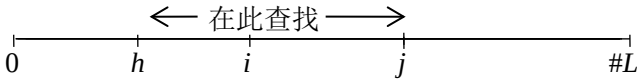
练习 181: 写一个程序, 在排好序的非空表中查找一个给定的项。执行时间必须与表的长度呈二的对数关系。采用的策略是: 如果给定项存在的话，判断给定项应出现在表的哪一半中, 然后再判断在哪个四分之一中, 然后是哪个八分之一中, 依次下去。

如同上一小节, 令表为 L, 欲查找的值为 x（这些都不是状态变量）。程序中仍用自然数变量 h 表示 x 在 L 中出现的索引, 如果 x 存在于 L 的话。不过这次还要用布尔变量 p 表示 x 是否出现在 L 中, 如果出现, p 赋值为 ⊤ , 否则赋值为 ⊥ 。暂时不考虑时间, 问题为

$$x: L(0,..\#L) = p' \Rightarrow Lh' = x$$

在查找过程中, 每次减少可能含有项 x 的表的区间。这里引入自然变量 i 和 j，用规范 R 来表示在表段 $h,..j$ 间的查找。

$$R = (x: L(h,..j) = p2 \ ® \ Lh2=x)$$



现在来求解这个问题。

$(x: L(0,..\#L) = p' \Rightarrow Lh' = x) \Leftarrow h := 0.\ j := \#L.\ h < j \Rightarrow R$

$h < j \Rightarrow R \Leftarrow$ **if** $j-h=1$ **then** $p := Lh=x$ **else** $j-h \geq 2 \Rightarrow R$ **fi**

$j-h \geq 2 \Rightarrow R \Leftarrow j-h \geq 2 \Rightarrow h' = h < i' < j = j'$.

if $Li \leq x$ **then** $h := i$ **else** $j := i$ **fi**.

$h < j \Rightarrow R$

要得到正确的结果, 如何选择 i 并不重要，只要它是在 h 和 j 之间。如果选择 $i := h+1$，就得到了线性查找。在最坏情况下，如果要得到最好的执行时间，必须选择将区间 $h;..j$ 分成两半的 i。要获得最佳平均执行时间, 应该选择 i 将区间 $h;\ ..\ j$ 分成两个具有相等查找到 x 可能性的区间。由于没有这种出现可能性的详细要求, 可以将 $h;..j$ 分成相同大小的两个部分。

$$j - h \geq 2 \Rightarrow h' = h < i' < j = j' \Leftarrow i := div\ (h+j)\ 2$$

在找到区间 $h;..j$ 的中点 i 后，可以对 $Li = x$ 进行测试，如果 Li 就是要查找的项，执行就此结束，这样可能减少执行时间。令人惊奇的是，如果加入这个测试，使用递归时间进行测量，最坏情况的时间完全没有改善，平均时间只改善了一点点，大约是$(\#L)/(\#L+1)$ 的一部分，假如在每个索引找到给定项和完全找不到它的概率相等。而如果使用真实时间进行测量，此时因为循环中不再是包含两个测试，而是三个测试，所以最坏情况和平均情况下的执行时间都要坏很多。

将 $t := t+1$ 放到最后那个递归调用之前, 就得到了递归执行时间。必须证明

$T \Leftarrow h:= 0.\ j := \#L.\ U$

$U \Leftarrow$ **if** $j-h=1$ **then** $p:=Lh=x$ **else** V **fi**

$V \Leftarrow i:= div\ (h+j)2.$
if $Li \le x$ **then** $h:= i$ **else** $j:= i$ **fi**.
$t := t+1.\ U$

这里计时表达式 T, U, V 应当适当选择。如果没有看到一个合适的选择, 也可将这一程序多执行几次, 看看各有什么结果。这样最坏的情况可能出现：被查找的项大于表中所有的项。对于这种情况有

$\#L$ = 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 ...

$t'-t$ =0 1 2 2 3 3 3 3 4 4 4 4 4 4 4 4 5 5 5 ...

由此定义

$T = t' \le t+ceil\ (log(\#L))$

$U = h<j \Rightarrow t' \le t+ceil\ (log\ (j-h))$

$V = j-h \ge 2 \Rightarrow t' \le t+ceil\ (log\ (j-h))$

其中函数 $ceil$ 的值为不小于自变量的最小整数。

因此可以将对程序设计的要求定义为三个层次。最低的层次是，没有清楚的规范和精化而写出的程序。第二个层次是，写出清楚的规范和精化，就象刚才对二分查找所做的；在实践中，可以不需形式化的证明很快地检验精化的正确性。最高的层次是，对每一步精化进行形式化的证明，这一步骤，使用自动的定理证明器会十分有帮助。

下面对本小节的七个精化进行证明。对第一个精化：

$$(x: L(0,..\#L) = p' \Rightarrow Lh' = x) \Leftarrow h:= 0.\ j:= \#L.\ h < j \Rightarrow R$$

从右边开始：

$h:= 0.\ j:= \#L.\ h < j \Rightarrow R$ 置换 R 并使用置换定律两次

$=$ $0<\#L \ ® \ (x: L\ (0,..\#L) = p2 \ Ⓡ \ Lh2 = x)$ 已知

L为非空

$=$ $(x: L(0,..\#L) = p2 \ Ⓡ \ Lh2=x)$

第二个精化：

$$h< j \Rightarrow R \Leftarrow \textbf{if}\ j-h=1\ \textbf{then}\ p:= Lh=x\ \ \textbf{else}\ \ j-h \ge 2 \Rightarrow R\ \textbf{fi}$$

可以分情况证明。第一种情况是：

$(h < j \Rightarrow R \Leftarrow j - h = 1 \wedge (p := Lh = x))$ 移动定律

$= \quad j - h = 1 \wedge (p := Lh = x) \Rightarrow R$ 扩展赋值与 R

$= \quad j - h=1 \ni p2=(Lh=x) \ni h2=h \ni i2=i \ni j2=j \circledR (x: L(h,..j) = p2 \circledR Lh2=x)$

使用前件作为上下文简化后件

$= \quad j - h = 1 \wedge p' = (Lh = x) \wedge h' = h \Rightarrow (x = Lh = Lh=x \circledR Lh=x)$

对称性，基定律和自反性

$= \quad \mathsf{T}$

第二个精化的第二种情况是：

$(h < j \Rightarrow R \Leftarrow j - h \neq 1 \wedge (j - h \geq 2 \Rightarrow R))$ 移动

$= \quad j - h \geq 2 \wedge (j - h \geq 2 \Rightarrow R) \Rightarrow R$ 抛弃

$= \quad j - h \geq 2 \wedge R \Rightarrow R$ 特定化

$= \quad \mathsf{T}$

下一个精化：

$j\text{-}h \geq 2 \Rightarrow R \Leftarrow j\text{-}h \geq 2 \Rightarrow h' = h < i' < j = j'.$

if $Li \leq x$ **then** $h := i$ **else** $j := i$ **fi**.

$h < j \Rightarrow R$

可以分情况证明。使用相关组合的分配律，其第一种情况是：

$(j - h \geq 2 \Rightarrow R \Leftarrow j - h \geq 2 \Rightarrow h' = h < i' < j = j'.Li \leq x \wedge (h := i.h < j \Rightarrow R))$ 条件

$\Leftarrow (j - h \geq 2 \Rightarrow R \Leftarrow j - h \geq 2 \Rightarrow (h' = h < i' < j = j'.Li \leq x \wedge (h := i.h < j \Rightarrow R)))$ 移动

$= \quad j–h\geq 2 \ni (j–h\geq 2 \circledR (h2=h<i2<j=j2.\ Li\leq x \ni (h:= i.\ h<j \circledR R))) \circledR R$

抛弃 $j - h \geq 2$ 和特定化

$\Leftarrow (h' = h < i' < j = j'.Li \leq x \wedge (h := i.h < i \Rightarrow R)) \Rightarrow R$ 扩展第一个 R，使用置换

$= \quad (h2=h<i2<j=j2.\ Li\leq x \ni (i<j \circledR (x: L\ (i,..j) = p2 \circledR Lh2 = x))) \circledR R$ 相关组合

$= \quad (\square h22,\ i22,\ j22,\ p22 \cdot \quad h22=h<i22<j=j22 \ni Li22\leq x$

$\ni (i22<j22 \circledR (x: L\ (i22,..j22) = p2 \circledR Lh2 = x)))$

$\Rightarrow R$ 使用单点定律消除 $p22$，$h22$和 $j22$，并使用 i 为 i'' 换名

$= \quad (\square i \cdot h<i<j \ni Li\leq x \ni (i<j \circledR (x: L(i,..j) = p2 \circledR Lh2=x))) \circledR R$

使用上下文 $i < j$ 抛弃

$= \quad (\square i \cdot h<i<j \ni Li\leq x \ni (x: L(i,..j) = p2 \circledR Lh2=x)) \circledR R$

若 $h < i$ 且 $Li \leq x$ 且 L 是有序的,则 $x : L(i,..j) = x : L(h,..j)$

$= \quad (\square i \cdot h<i<j \ni Li\leq x \ni (x: L(h,..j) = p2 \circledR Lh2=x)) \circledR R$

注意到 $x: L(h,..j) = p2 \circledR Lh2=x$ 是 R 由于它不使用到 i ,因此可将它提到量词符外面

$= \quad (\exists i \cdot h < i < j \wedge Li \leq x) \wedge R \Rightarrow R$ 特定化

$= \quad \mathsf{T}$

第二种情况：

$$j-h\geq 2\Rightarrow R \Leftarrow j-h\geq 2\Rightarrow h'=h<i'<j=j'.Li>x\wedge(j:=i.h<j\Rightarrow R)$$

的证明与第一种类似。

下一个精化是：

$(j-h\geq 2\Rightarrow h'=h<i'<j=j' \Leftarrow i:=div\ (h+j)\ 2)$ 扩展赋值

$=$ $(j-h\geq 2\Rightarrow h'=h<i'<j=j' \Leftarrow i':=div\ (h+j)\ 2\ \ni p2=p\ \ni h2=h\ \ni j2=j)$

使用前件作为上下文简化后件

$=$ $(j-h\geq 2\Rightarrow h=h<div(h+j)2<j=j \Leftarrow i'=div(h+j)2\wedge h'=h\wedge j'=j)$

简化 $h=h$ 和 $j=j$，使用 div 的性质

$=$ $(j-h\geq 2\Rightarrow \mathsf{T}\Leftarrow i'=div(h+j)2\wedge h'=h\wedge j'=j)$ 基定律两次

$=$ T

下一个精化是：

$(\mathsf{T} \Leftarrow h:=0.\ j:=\#L.\ U)$ 替换 T 和 U

$=$ $(t'\leq t+ceil(\log(\#L)) \Leftarrow h:=0.j:=\#L.h<j\Rightarrow t'\leq t+ceil(\log(j-h)))$

置换定律两次

$=$ $(t'\leq t+ceil(\log(\#L)) \Leftarrow 0<\#L\Rightarrow t'\leq t+ceil(\log(\#L-0)))$

$=$ T

下一个精化：

$U \Leftarrow$ **if** $j–h=1$ **then** $p:=Lh=x$ **else** V **fi**

可以分情况证明。第一种情况：

$(U \Leftarrow j-h=1\wedge(p:=Lh=x))$ 扩展 U 和赋值

$=$ $(h<j\Rightarrow t'\leq t+ceil\ (log\ (j–h)) \Leftarrow j–h=1\wedge p'=(Lh=x)\wedge h'=h\wedge j'=j\wedge t'=t)$

使用主前件作为上下文简化后件

$=$ $(h<j\Rightarrow t'\leq t+ceil\ (log\ 1) \Leftarrow j–h=1\wedge p'=(Lh=x)\wedge h'=h\wedge j'=j\wedge t'=t)$

使用 $\log 1=0$

$=$ $(h<j\Rightarrow \mathsf{T} \Leftarrow j–h=1\wedge p'=(Lh=x)\wedge h'=h\wedge j'=j\wedge t'=t)$ 基定律两次

$=$ T

第二种情况要证明的是：

$(U \Leftarrow j-h\neq 1\wedge V)$ 扩展 U 和 V

$=$ $(h<j\Rightarrow t'\leq t+ceil\ (log\ (j–h)) \Leftarrow j–h\neq 1\wedge(j-h\geq 2\Rightarrow t'=t+ceil(\log(j-h))))$

移动

$=$ $h<j\wedge j-h\neq 1\ \wedge(j-h\geq 2\Rightarrow t'=t+ceil(\log(j-h)))\Rightarrow t'\leq t+ceil\ (log\ (j–h))$

简化

$=$ $j-h\geq 2\ \wedge(j-h\geq 2\Rightarrow t'=t+ceil(\log(j-h)))\Rightarrow t'\leq t+ceil\ (log\ (j–h))$ 抛弃

$=$ $j-h\geq 2\ \wedge t'\leq t+ceil(\log(j-h)))\Rightarrow t'\leq t+ceil\ (log\ (j–h))$ 特定化

$=$ T

在证明下一个精化之前，先证明两个小定理。

if *even* (*h*+*j*)
then *div*(*h*+*j*)2 < *j*
= (*h*+*j*)/2 < *j*
= *j*–*h* >0
⇐ *j*–*h* ≥ 2
else *div*(*h*+*j*)2 < *j*
= (*h*+*j*–1)/2 < *j*
= *j*–*h* > –1
⇐ *j*–*h* ≥ 2 **fi**

if *even*(*h*+*j*)
then 1+*ceil* (*log* (*j*–*div*(*h*+*j*)2))
= *ceil* (1+*log* (*j*–(*h*+*j*)/2))
= *ceil* (*log*(*j*–*h*))
else 1+*ceil* (*log* (*j*–*div*(*h*+*j*)2))
= *ceil* (1+*log* (*j*–(*h*+*j*–1)/2))
= *ceil* (*log*(*j*–*h*+1)) 如果 *h*+*j* 是奇数, 那么 *j*–*h* 也是奇数因而不是 2 的幂次方
= *ceil* (*log*(*j*–*h*)) **fi**

最后一个精化：

$V \Leftarrow i := div\ (h+j)\ 2.$ **if** $Li \le x$ **then** $h := i$ **else** $j := i$ **fi**. $t := t+1.\ U$

可以分成两种情况。第一种情况：

$(V \Leftarrow i := div\ (h+j)\ 2.\ Li \le x \wedge (h := i.\ t := t+1\ .\ U))$ 去除 $Li \le x$ 并置换 U

$\Leftarrow (V \Leftarrow i := div\ (h+j)\ 2.\ h := i.\ t := t+1.\ h < j \Rightarrow t' \le t + ceil(\log(j-h)))$

使用置换定律三次

$= (V \Leftarrow div\ (h+j)\ 2 < j \Rightarrow t' \le t+1+ceil(\log(j - div(h+j)2)))$

使用上面两个小定理

$\Leftarrow (V \Leftarrow j - h \ge 2 \Rightarrow t' \le t + ceil(\log(j-h)))$ V 的定义，自反定律

$= \mathsf{T}$

第二种情况

$V \Leftarrow i := div\ (h+j)\ 2.\ Li > x \wedge (j := i.\ t := t+1.\ U)$

的证明与第一种类似。

----------二分查找结束

4.2.6 快速指数运算

练习 177: 给定有理数变量 *x* 和 *z*, 和自然数变量 *y*, 编写程序计算 $z' = x^y$，要求不使用指数运算且快速执行。

该规范没有讲明执行应当有多快, 尽可能地快就行了。可以采用累加乘积的办法 ,使用变量 *z* 作为累加器, 定义

$$P = z' = z \times x^y$$

可以求解问题如下, 虽然它并不是计算最快的。

$z' = x^y \Leftarrow z := 1.\ P$

$P \Leftarrow$ **if** $y=0$ **then** *ok* **else** $y > 0 \Rightarrow P$ **fi**

$y>0 \Rightarrow P \Leftarrow z := z \times x.\ y := y-1.\ P$

为了加快计算速度, 可以将精化式中的 $y>0 \Rightarrow P$ 改成测试 y 的奇偶性: 如果 y 是奇数则没有什么改进, 但如果 y 是偶数, 则可以将 y 减成一半。

$y>0 \Rightarrow P \Leftarrow$ **if** *even* y **then** *even* $y \wedge y>0 \Rightarrow P$ **else** *odd* $y \Rightarrow P$ **fi**

even $y \wedge y>0 \Rightarrow P \Leftarrow x := x \times x.\ y := y/2.\ P$

odd $y \Rightarrow P \Leftarrow z := z \times x.\ y := y-1.\ P$

以上的精化式都很容易证明。

作了以上重要改进之后, 还有几处小的地方可以改进。之所以在这里改进它，部分目的是作为一个练习, 让读者了解如何尽可能地加快执行速度, 而主要目的是作为程序修改的例子。首先，如果 y 是偶数且比 0 大, 那么至少是 2; 减少一半后, 就至少为 1。利用这个信息重新精化

$$\textit{even}\ y \wedge y>0 \Rightarrow P \Leftarrow x := x \times x.\ y := y/2.\ y>0 \Rightarrow P$$

如果 y 初始值为奇数, 将它减去 1 ,那么必然变成偶数。利用这一信息, 重新精化

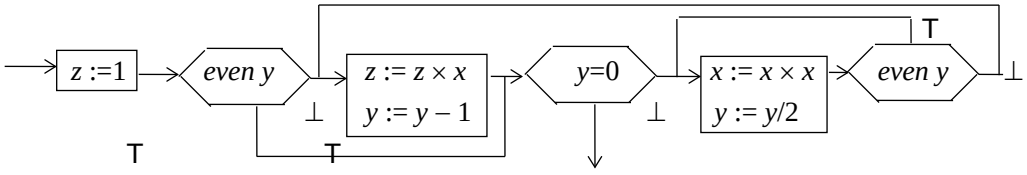
odd $y \Rightarrow P \Leftarrow z := z \times x.\ y := y-1.$ *even* $y \Rightarrow P$

even $y \Rightarrow P \Leftarrow$ **if** $y=0$ **then** *ok* **else** *even* $y \wedge y>0 \Rightarrow P$ **fi**

另一个更加次要的改进：如果程序用来计算 x^0 的机会小于计算 x 的奇次幂的机会（这是一个合理的假设）, 那么程序开始时测试奇偶性要比测试是否为零要好。因此再重精化为

$$P \Leftarrow \textbf{if}\ \textit{even}\ y\ \textbf{then}\ \textit{even}\ y \Rightarrow P\ \textbf{else}\ \textit{odd}\ y \Rightarrow P\ \textbf{fi}$$

程序修改, 无论是为了获得一点加速还是为了其它目的, 如果没有适当的理论指导, 在实际做时是较容易出错的。读者不妨将这一程序用你最熟悉的标准程序设计语言写出来, 先写前面第一个较简单的解, 然后再作几次上面所述的修改。第一次修改使得循环体中增加一个新的情况语句; 第二次修改将其中的一个情况分支变成了内层循环; 第三次修改将外层循环变成了内层循环中的一个情况语句, 带有一个中间出口; 最后一次修改将循环的入口点变成了另外两个入口点的一个选择。程序流程图如下所示:



如果没有理论依据, 光靠对程序外观施行手术必定会导致一些错误。但有了理论的指导, 每一个精化都是简单的定理, 就有更好的机会来保证程序修改的正确性。

在考虑时间因素之前, 再次列出这个快速指数运算的程序:

$z' = x^y \Leftarrow z := 1.\ P$

$P \Leftarrow$ **if** *even* y **then** *even* $y \Rightarrow P$ **else** *odd* $y \Rightarrow P$ **fi**

$even\ y \Rightarrow P \Leftarrow$ **if** y=0 **then** ok **else** $even\ y \wedge y>0 \Rightarrow P$ **fi**

$odd\ y \Rightarrow P \Leftarrow z := z \times x.\ y := y-1.\ even\ y \Rightarrow P$

$even\ y \wedge y>0 \Rightarrow P \Leftarrow x := x \times x.\ y := y/2.\ y>0 \Rightarrow P$

$y>0 \Rightarrow P \Leftarrow$ **if** $even\ y$ **then** $even\ y \wedge y>0 \Rightarrow P$ **else** $odd\ y \Rightarrow P$ **fi**

在递归时间度量中, 调用的每一次循环必须加入时间增长式。在本程序中, 只要在调用 $y>0 \Rightarrow P$ 之前加入一个时间增长式即可：

$$even\ y \wedge y>0 \Rightarrow P \Leftarrow x := x \times x.\ y := y/2.\ t := t+1.\ y>0 \Rightarrow P$$

为了帮助确定要证明的时间界限, 可以用一些测试例子来运行该程序。通过发现对每个自然数 n，y: 2^n,..$2^{n+1} \Rightarrow t' = t+n$ ，加上一个孤立情况 y=0 $\Rightarrow t' = t$ ，于是定义计时规范为:

$$\textbf{if } y=0 \textbf{ then } t' = t \textbf{ else } t' = t + floor(\log y) \textbf{ fi}$$

其中函数 *floor* 的值为不大于自变量的最大整数。可以证明这个是准确的执行时间，但证明下面的相对不精确的规范 T 要更容易：

$$T = \textbf{ if } y=0 \textbf{ then } t' = t \textbf{ else } t' \le t + \log y \textbf{ fi}$$

为了做到这一点，必须将 T 精化为和精化 $z' = x^y$ 同样的结构以便可以根据部分精化法将结果和时间的规范进行合并。可以证明：

$T \Leftarrow z := 1.\ T$

$T \Leftarrow$ **if** $even\ y$ **then** T **else** $y>0 \Rightarrow T$ **fi**

$T \Leftarrow$ **if** y=0 **then** ok **else** $y>0 \Rightarrow T$ **fi**

$y>0 \Rightarrow T \Leftarrow z := z \times x.\ y := y-1.\ T$

$y>0 \Rightarrow T \Leftarrow x := x \times x.\ y := y/2.\ t := t+1.\ y>0 \Rightarrow T$

$y>0 \Rightarrow T \Leftarrow$ **if** $even\ y$ **then** $y>0 \Rightarrow T$ **else** $y>0 \Rightarrow T$ **fi**

规范 T 和 $y>0 \Rightarrow T$ 不止一次地被精化，但没关系。当将这些规范与前面的结果规范合并时，事实上每个规范只被精化了一次。

计时可以写成下面的合取式：

$$(y=0 \Rightarrow t' = t) \wedge (y>0 \Rightarrow t' \le t + \log y)$$

很容易想试图分成两部分来证明。不幸的是不能对计时的第二部分独立地证明。可见将规范划分成部分并不总是成功的。

----------快速指数运算结束

4.2.7 斐波那契数

在本小节中, 将讨论练习 250。斐波那契数（Fibonacci Numbers）的定义是

$fib\ 0=0$

$fib\ 1=1$

$fib\ (n+2) = fib\ n + fib\ (n+1)$

于是可以使用递归函数的定义:

$fib = 0 \rightarrow 0 \mid 1 \rightarrow 1 \mid \langle n: nat+2 \rightarrow fib\ (n-2) + fib\ (n-1) \rangle$

$= \langle n: nat \rightarrow$ **if** $n<2$ **then** n **else** $fib\ (n-2) + fib\ (n-1)$ **fi** $\rangle$

不过现在的程序设计语言中没有引入函数, 所以还需要做一些工作。除此之外, 这个函数型的解需要指数级执行时间, 而事实上可以获得更快速的解。

对于 $n \geq 2$, 如果知道了前两个斐波那契数, 就能得到下一个斐波那契数。于是可以去跟踪这对斐波那契数。令 x, y 和 n 为自然数变量, 精化

$$x' = fib\ n \Leftarrow P$$

其中 P 为寻找一对斐波那契数的问题。

$$P = x' = fib\ n \wedge y' = fib\ (n+1)$$

当 $n=0$ 时, 解很简单。当 $n \geq 1$ 时, 可以先将 n 减去 1, 再利用这一实参寻找斐波那契数对, 然后将 x 和 y 分别前进一个数。

$$P \Leftarrow \textbf{if } n=0 \textbf{ then } x:= 0.\ y:= 1 \textbf{ else } n:= n-1.\ P.\ x' = y \wedge y' = x+y \textbf{ fi}$$

要使数 x 和 y 向前, 需要再添加一个变量。可以使用一个新变量, 不过这里已经有变量 n; 利用 n 作这一用途安全吗? 规范 $x' = y \wedge y' = x+y$ 允许 n 被改变, 所以可以使用 n。

$$x' = y \wedge y' = x+y \Leftarrow n:= x.\ x:= y.\ y:= n+y$$

这个解的时间是线性的。要证明它, 可以使用相同的精化式结构, 但在规范中加入时间。将 P 替换成 $t' = t+n$, 并且在使用之前加上 $t:=t+1$; 同时将 $x' = y \wedge y' = x+y$ 替换成 $t' = t$。

$$t' = t+n \Leftarrow \textbf{if } n=0 \textbf{ then } x:= 0.\ y:= 1 \textbf{ else } n:= n-1.\ t:= t+1.\ t' = t+n.\ t' = t \textbf{ fi}$$

$$t' = t \Leftarrow n:= x.\ x:= y.\ y:= n+y$$

线性的时间比指数级的时间要好得多, 但还可以进一步改进。练习 250 求一个对数级时间的解。要得到这个解, 需要利用该练习中提供的提示, 并使用下面的等式:

$$fib\ (2\times k+1) = (fib\ k)^2 + (fib\ (k+1))^2$$

$$fib\ (2\times k+2) = 2\times fib\ k \times fib\ (k+1) + (fib\ (k+1))^2$$

这些等式说明可以利用之前的数对 $fib\ k$, $fib\ (k+1)$来计算数对 $fib\ (2\times k+1)$和 $fib\ (2\times k+2)$, 而前两者的实参恰为后两者的一半。进行如下精化

$$\begin{aligned} P \Leftarrow\ & \textbf{if } n=0 \textbf{ then } x:= 0.\ y:= 1 \\ & \textbf{else if } even\ n \textbf{ then } even\ n \wedge n>0 \Rightarrow P \\ & \textbf{else } odd\ n \Rightarrow P \textbf{ fi fi} \end{aligned}$$

先处理最后一个新问题。如果 n 是奇数, 通过赋值 $n:= (n-1)/2$, 可以将 n 从 $2\times k+1$ 减成 k, 然后调用 P 得到 $fib\ k$ 和 $fib\ (k+1)$, 然后再利用上述等式计算出 $fib\ (2\times k+1)$和 $fib\ (2\times k+2)$。

$$odd\ n \Rightarrow P \Leftarrow n:= (n-1)/2.\ P.\ x' = x^2+y^2 \wedge y' = 2\times x\times y+y^2$$

情况分支 $even\ n \wedge n>0$ 要稍难一些。通过赋值语句 $n:= n/2-1$ 将 n 从 $2\times k+2$ 减成 k, 然后调用 P 获得 $fib\ k$ 和 $fib\ (k+1)$, 然后再同奇数时一样, 利用上述等式获得 $fib\ (2\times k+1)$和 $fib\ (2\times k+2)$, 不过这次要求出 $fib\ (2\times k+2)$和 $fib\ (2\times k+3)$。将 $fib\ (2\times k+1)$和 $fib\ (2\times k+2)$相加就得 $fib\ (2\times k+3)$:

$$even\ n \wedge n>0 \Rightarrow P \Leftarrow n:= n/2-1.\ P.\ x' = 2\times x\times y+y^2 \wedge y' = x^2+y^2+x'$$

剩下的问题就是根据 x 和 y 求出 x' 和 y', 这需要第三个变量, 同前面一样, 可以利用 n。

$$x' = x^2+y^2 \wedge y' = 2\times x\times y+y^2 \Leftarrow n:= x.\ x:= x^2+y^2.\ y:= 2\times n\times y+y^2$$

$$x' = 2\times x\times y+y^2 \wedge y' = x^2+y^2+x' \Leftarrow n:= x.\ x:= 2\times x\times y+y^2.\ y:= n^2+y^2+x$$

现在来证明所得程序的执行时间为对数级时间。重新定义时间规范

$$T = t' \leq t+log\ (n+1)$$

并将 $t:=t+1$ 放到 T 调用之前, 现在必须证明:

$$T \Leftarrow \textbf{if } n=0 \textbf{ then } x:= 0.\ y:= 1 \textbf{ else if } even\ n \textbf{ then } even\ n \wedge n>0 \Rightarrow T \textbf{ else } odd\ n \Rightarrow T \textbf{ fi fi } odd$$

$n \Rightarrow T \Leftarrow n := (n-1)/2.\ t := t+1.\ T.\ t' = t$

$even\ n \wedge n>0 \Rightarrow T \Leftarrow n := n/2-1.\ t := t+1.\ T.\ t' = t$

$t' = t \Leftarrow n := x.\ x := x^2+y^2.\ y := 2\times n\times y+y^2$

$t' = t \Leftarrow n := x.\ x := 2\times x\times y+y^2.\ y := n^2+y^2+x$

第一个和最后两个式子容易证明, 现在来证明其余的两个。

$(odd\ n \Rightarrow t' \le t+log\ (n+1)) \Leftarrow (n := (n-1)/2.\ t := t+1.\ t' \le t+log\ (n+1).\ t' = t)$

$= (odd\ n \Rightarrow t' \le t+log\ (n+1)) \Leftarrow t' \le t+1+log\ ((n-1)/2+1)$

注意有 $(a \Rightarrow b) \Leftarrow c = a \Rightarrow (b \Leftarrow c)$

$= odd\ n \Rightarrow (t' \le t+log\ (n+1) \Leftarrow t' \le t+1+log\ ((n-1)/2+1))$ 连接定律

$\Leftarrow odd\ n \Rightarrow 1+log\ ((n-1)/2+1) \le log\ (n+1)$ 对数定律

$= odd\ n \Rightarrow log\ (n-1+2) \le log\ (n+1)$ 算术

$= odd\ n \Rightarrow log\ (n+1) \le log\ (n+1)$ 自反与基定律

$= T$

$(even\ n \wedge n>0 \Rightarrow t' \le t+log\ (n+1)) \Leftarrow (n := n/2-1.\ t := t+1.\ t' \le t+log\ (n+1).\ t' = t)$

利用同样的步骤

$= even\ n \wedge n>0 \Rightarrow 1+log\ (n/2-1+1) \le log\ (n+1)$

$= even\ n \wedge n>0 \Rightarrow log\ n \le log\ (n+1)$

$= T$

----------斐波那契数结束

要知道一个程序的执行时间，可以将程序转换成一个表示执行时间的函数。这里通过做练习 249（过山车）加以说明，这是一个执行时间为未知的著名程序。令 n 为自然数变量，那么，包括递归时间，有：

$n'=1 \Leftarrow$ **if** $n = 1$ **then** *ok*

else if $even\ n$ **then** $n := n/2.\ t := t+1.\ n2=1$

else $n := 3\times n + 1.\ t := t+1.\ n2=1$ **fi fi**

对于所有的 $n > 0$，甚至执行时间是否为有限都是未知的。

可以使用 $f\,n$ 表示执行时间，其中 f 必须满足：

$t'=t+fn \Leftarrow$ **if** $n = 1$ **then** *ok*

else if $even\ n$ **then** $n := n/2.\ t := t+1.\ t'=t+fn$

else $n := 3\times n + 1.\ t := t+1.\ t'=t+fn$ **fi fi**

可进一步简化为：

$fn =$ **if** $n = 1$ **then** 0

else if $even\ n$ **then** $1+f(n/2)$

else $1+f(3\times n+1)$ **fi fi**

这样就有了一个执行时间的确切函数。那么为什么执行时间被认为是未知的呢？

若某个程序的执行时间是 n^2，可以认为程序的执行时间是已知的。为什么 n^2 被认为是时间界，而以上所定义的 fn 却不被接受呢？在回答之前，可以先建议几个不是的原因。原因不是 f 为递归定义的，因为平方函数是基于乘法定义，而乘法的定义也是递归的。原因也不是 n^2 具有良好的行为（有限、单调、平滑），而 f 是跳跃的， f 的值的每次跳跃和改变都很好地反映了源程序的执行时间，但不能因为它是一个完美界限而认为 f 不合格。

人们还可能将计算时间界所需花费的时间长度作为反对 f 的理由。因为计算时间界 fn 的时间与执行程序的时间是相同的。只要运行源程序，看自己的手表就能知道时间界。但是即使计算 *log log n* 的时间比 *log log n* 要长，*log log n* 还是被作为一个时间界。

原因似乎是，函数 f 是不熟悉的；它没有被仔细研究过，因此不被了解。如果它像平方函数一样被研究和熟悉，人们就会接受它作为时间界。

前面讨论了线性查找，在一个给定的表中查找一个给定项的第一次出现。现在假设表是无限长的，且在表中至少有项 *x* 的一次出现。所要求的结果可以被简化为：

$$\neg x : L(0,..h') \wedge Lh' = x$$

且程序可被简化为：

$\neg x : L(0,..h') \wedge Lh' = x \Leftarrow h := 0.\ \neg x : L(h,..h2)\ э\ Lh2 = x$

$\neg x : L(h,..h') \wedge Lh' = x \Leftarrow$ **if** $Lh = x$ **then** *ok* **else** $h := h+1.\ \neg x : L(h,..h2)\ э\ Lh2 = x$ **fi**

加入递归时间，可以证明：

$t' = t + h' \Leftarrow h := 0.\ t' = t + h' - h$

$t' = t + h' - h \Leftarrow$ **if** $Lh = x$ **then** *ok* **else** $h := h+1.\ t := t+1.\ t' = t + h' - h$ **fi**

执行时间是 h'。它是否可以作为时间界呢？它并没有给出为了得到结果需要等待多久的信息。另一方面，对于执行时间也没有给出更多的信息。其过错在一个给定的信息内：*x* 出现在某个地方，但不知道是什么地方。

----------时间结束

4.3 空间

这里使用练习 286：汉诺塔问题来说明空间的计算。有三个塔和 n 个盘子。盘子有大小，盘子 *0* 为最小而盘子 $n-1$ 为最大。初始时，*n* 个盘子都在 A 塔上，按最小盘子在最上端，最大盘子在最下端的顺序排列。任务是将 A 塔的所有盘子移至 B 塔，每次只能移动一个盘子，且大的盘子永远不能放在小的盘子上面。在这过程中，可以使用 C 塔作为中间存储设施。

问题的一个解是 *MovePile"A""B""C"*，可以将 *MovePile* 精化为：

MovePile from to using $\Leftarrow$ **if** *n=0* **then** *ok*
else *n*:= *n*-1.
MovePile from using to.
MoveDisk from to.
MovePile using to from.
n:= *n*+1 **fi**

过程 *MovePile* 移动所有的 *n* 个盘子，每次移动一个盘子，大的盘子不放在小的盘子上。它的第一个参数 *from* 表示 *n* 个盘子初始所在的塔，第二个参数 *to* 表示 *n* 个盘子最终所在的塔，第三个参数 *using* 表示作为中间存储的塔。它的任务是这样完成的。如果还有盘子需要移动，一开始忽略最底部的盘子 $(n := n-1)$。然后递归调用从 *from* 塔移动剩余的盘子（除了最底部盘子的所有盘子，每次移动一个，大的不放在小的上面）到 *using* 塔（使用 *to* 塔作为中间存储）。然后 *MoveDisk* 将移动底部的盘子。如果没有真正的机器人手臂移动盘子，也可以只打印应该做的事：

“Move Disk”; *nat2text n*; “from tower”; *from*; “to tower”; *to*

然后递归调用从 *using* 塔移动剩余的盘子（除了最底部盘子的所有盘子，每次移动一个，大的不放在小的上面）到 *to* 塔（使用 *from* 塔作为中间存储）。最后 *n* 恢复原来的值。

为了形式化 *MovePile* 和 *MoveDisk* 以证明遵守了规则且盘子最终到达了正确的目的地，这里需要形式化地表示盘子在塔上的位置。但这不是本节的重点。这里所关心的是时间和

空间需求，因此可以忽略盘子的位置和参数 *from*，*to* 和 *using*。现在能证明的只是如果 *MoveDisk* 满足 *n'=n*，那么 *MovePile* 也满足。

为了测量时间，可以加入一个时间变量 *t*，使用它计算盘子的移动。假定 *MoveDisk* 花费时间 1，这是目前唯一需要关心的，那么将它替换为 *t:=t*+1。现在将 *MovePile* 替换为 $t := t + 2^n - 1$来证明执行时间为$2^n - 1$。分情况证明下式：

$t := t + 2^n - 1 \Leftarrow$ **if** *n=0* **then** *ok*
else *n*:= *n*-1.
$t := t + 2^n - 1$.
t:= *t*+1.
$t := t + 2^n - 1$.
n:= *n*+1 **fi**

第一种情况，从右边开始：

$n = 0 \wedge ok$ 扩展 *ok*

$= \quad n = 0 \wedge n' = n \wedge t' = t$ 算术

$\Rightarrow \quad t := t + 2^n - 1$

第二种情况，从右边开始：

$n > 0 \wedge$ (*n:=n*-1. $t := t + 2^n - 1$. *t:=t*+1. $t := t + 2^n - 1$. *n:=n*+1)

抛弃合取因子$n > 0$，扩展最后的赋值式

$\Rightarrow \quad$ *n*:= *n*-1. $t := t + 2^n - 1$. *t:=t*+1. $t := t + 2^n - 1$. $n := n+1 \wedge t' = t$

从右向左重复使用置换定律

$= \quad n' = n - 1 + 1 \wedge t' = t + 2^{n-1} - 1 + 1 + 2^{n-1} - 1$ 简化

$= \quad n' = n \wedge t' = t + 2^n - 1$

$= \quad t := t + 2^n - 1$

要讨论一个计算所使用的存储空间，只需一个空间变量 *s*。正如时间变量 *t* 一样，*s* 也不是实现的一部分，只用于说明和计算空间需求。使用 *s* 表示执行初始时所占用的空间，而 *s′*表示在执行结束时最终占用的空间。任何一个程序都可能是一个更大的程序的一部分，而且可能不是第一个部分，因此不能假定程序初始时的空间占用为 0，正如不能假定一个计算从时间 0 开始。在例子中，程序递归调用其自身，每次递归调用都从不同的时间和占用不同的空间开始。

为了表示无限地消耗空间的执行的可能，可以将空间变量的定义域从自然数扩展到∞。只要空间是增加的，适当地插入 *s*:=*s*+(增加量)，如果空间是减少的，插入 *s*:=*s*-（减少量）。在例子中，递归调用不是精化的最后一个动作，递归调用要求在调用开始时将一个返回地址压入栈中，结束时弹出。如果忽略时间和盘子的移动，只考虑空间，可以证明：

s'=*s* $\Leftarrow$ **if** *n=0* **then** *ok*
else *n*:= *n*-1.
s:= *s*+1. *s′*=*s*. *s*:= *s*-1.
ok.
s:= *s*+1. *s′*=*s*. *s*:= *s*-1.
n:= *n*+1 **fi**

它表示终止时的空间占用和初始时相等。

没有“空间泄漏”是令人满意的，但它并不表示空间的使用情况。对空间的使用有两种有趣的测量方法：占用的最大空间和占用的平均空间。

4.3.0 最大空间

令 m 为执行开始时所占用的最大空间（注意：任何程序可能是一个更早开始执行的更大的程序的一部分）， m' 为执行结束时所占用的最大空间。每当空间增加时，对当前的 m 插入 $m:=\max\ m\ s$ 。当空间减少时没有必要改变 m 。在例子中，想证明的是所占用的最大空间为 n 。然而，在一个更大的环境中，开始空间不为 0 是可能的，因此有 $m'=s+n$ 。假设在开始时 $m\geq s$ ，因为 m 被假定为 s 的最大值，但也有可能 m 的初始值已经大于 $s+n$ ，因此规范变成 $m\geq s\Rightarrow(m:=\max m(s+n))$ 。

$m\geq s\Rightarrow(m:=\max m(s+n))\Leftarrow$

if $n=0$ **then** ok

else $n:=n-1.$

$s:=s+1.\ m:=\max m\ s.\ m\geq s\Rightarrow(m:=\max m(s+n)).\ s:=s-1.$

$ok.$

$s:=s+1.\ m:=\max m\ s.\ m\geq s\Rightarrow(m:=\max m(s+n)).\ s:=s-1.$

$n:=n+1$ **fi**

在证明之前，先对出现了两次的长的行进行简化。

$s:=s+1.\ m:=\max m\ s.\ m\geq s\Rightarrow(m:=\max m(s+n)).\ s:=s-1.$

使用条件定律，扩展最后的赋值式

$\Rightarrow$ $s:=s+1.\ m:=\max m\ s.$

$m\geq s\Rightarrow(m:=\max m(s+n).\ s':=s-1\wedge m'=m\wedge n'=n)$

使用置换定律

= $s:=s+1.\ m:=\max m\ s.\ m\geq s\Rightarrow s'=s-1\wedge m'=\max m(s+n)\wedge n'=n$

使用置换定律

= $s:=s+1.\ (\max m\ s)\geq s\Rightarrow s'=s-1\wedge m'=\max(\max ms)(s+n)\wedge n'=n$

前件简化成 T，且 max 是可结合的

= $s:=s+1.\ s2=s-1\ э\ m2=\max\ m\ (s+n)\ э\ n2=n$ 使用置换定律

= $s'=s\wedge m'=\max m(s+1+n)\wedge n'=n$

= $m:=\max m(s+1+n)$

精化的证明仍然分成两种情况。第一种情况：

$n=0\wedge ok$

= $n'=n=0\wedge s'=s\wedge m'=m$

$\Rightarrow$ $m\geq s\Rightarrow(m:=\max m(s+n))$

第二种情况：

$n>0\wedge(n:=n-1.$

$s:=s+1.\ m:=\max m\ s.\ m\geq s\Rightarrow(m:=\max m(s+n)).\ s:=s-1.$

$ok.$

$s:=s+1.\ m:=\max m\ s.\ m\geq s\Rightarrow(m:=\max m(s+n)).\ s:=s-1.$

$n:=n+1)$ 抛弃 $n>0$ 和 ok ，简化长的行，扩展最后的赋值式

$\Rightarrow$ $n:=n-1.\ m:=\max\ m\ (s+1+n).\ m:=\max\ m\ (s+1+n).\ n2=n+1\ э\ s2=s\ э\ m2=m$

使用置换定律三次

$=$ $n' = n \wedge s' = s \wedge m' = \max(\max m(s+n))(s+n)$ 结合律与幂等律

$=$ $n' = n \wedge s' = s \wedge m' = \max m(s+n)$

$\Rightarrow$ $m \geq s \Rightarrow (m := \max m(s+n))$

----------最大空间结束

4.3.1 平均空间

为了寻找一个计算所占用的平均空间，可以找到累计的空间-时间的乘积，然后除以执行时间。令 p 为执行开始时的空间-时间乘积， p' 为执行结束时的空间-时间乘积。另外还需要变量 s，像前面一样可以改变它。这里不需要变量 t；然而， p 的增加在 t 增加时发生，增加的量就是 s 的增量乘以 t 的增量。在例子中， t 每次增加 1， p 则增加 $s\times 1$ 。证明：

p:=p+s×(2^n –1)+(n–2)×2^n +2 ∠

if $n = 0$ **then** ok

else $n := n - 1.$

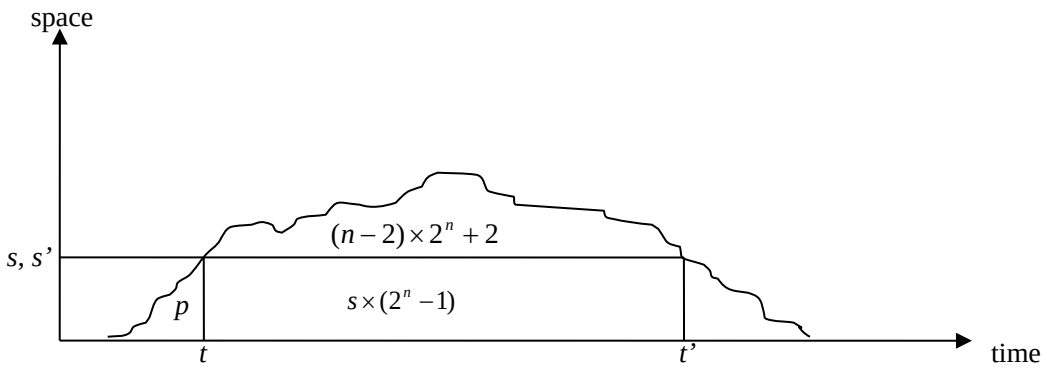
s:= s+1. p:= p + s×(2^n – 1) + (n–2)×2^n + 2. s:= s–1.

$p := p + s.$

s:= s+1. p:= p + s×(2^n – 1) + (n–2)×2^n + 2. s:= s–1.

$n := n + 1$ **fi**

在规范 $p := p + s\times(2^n - 1) + (n-2)\times 2^n + 2$ 中,项 $s\times(2^n - 1)$ 是初始空间 s 和总时间 $2^n - 1$ 的乘积；它是根据周围的计算（若 s 为 0 则为 0）决定的空间-时间乘积的增量。根据计算，其增加的量为 $(n-2)\times 2^n + 2$ 。而计算的平均空间就是这个增加的量除以执行时间。这样，计算占用的平均空间为 $n + n/(2^n - 1) - 2$ 。



证明同样分两部分，第一部分：

$n = 0 \wedge ok$ 扩展 ok

$=$ $n = o \wedge n' = n \wedge s' = s \wedge p' = p$ 算术

$\Rightarrow$ $n' = n \wedge s' = s \wedge p' = p + s\times(2^n - 1) + (n-2)\times 2^n + 2$

$=$ $p := p + s\times(2^n - 1) + (n-2)\times 2^n + 2$

最后一部分：

$n > 0 \wedge (n := n - 1.\ s$:=$s$+1. p:=p+s×(2^n –1)+(n–2)×2^n +2. s:=s–1. $n := n + 1.$

$p := p + s.$

$n := n - 1.\ s$:=s+1. p:=p+s×(2^n –1)+(n–2)×2^n +2. s:=s–1. $n := n + 1)$

抛弃 $n>0$，扩展最后的赋值式

$\Rightarrow$ *n*:=*n*–1. *s*:=*s*+1. *p*:=*p*+*s*×(2^n –1)+(*n*–2)×2^n +2. *s*:=*s*–1. *n*:=*n*+1. *p*:=*p*+*s*.

n:= *n*–1. *s*:= *s*+1. *p*:= *p* + *s*×(2^n–1) + (*n*–2)×2^n + 2. *s*:= *s*–1.

*n*2=*n*+1ə*s*2=*s*ə*p*2=*p*

从右向左使用置换定律 10 次

$= \quad n' = n \wedge s' = s$

$\wedge\ p' = p + (s+1)\times(2^{n-1}-1) + (n-3)\times 2^{n-1} + 2 + s + (s+1)\times(2^{n-1}-1) + (n-3)\times 2^{n-1} + 2$

简化

$= \quad n' = n \wedge s' = s \wedge p' = p + s\times(2^n-1) + (n-2)\times 2^n + 2$

$= \quad p := p + s\times(2^n-1) + (n-2)\times 2^n + 2$

证明平均空间的界为 n 比证明平均空间正好为 $n + n/(2^n-1) - 2$ 要容易。因此与其证明空间-时间乘积为 $s\times(2^n-1) + (n-2)\times 2^n + 2$，不如证明其最大为 $(s+n)(2^n-1)$。这里把它留作练习 286(f)。

----------平均空间结束

现在将关于汉诺塔问题的所有证明放在一起，得到：

MovePile $\Leftarrow$ **if** *n*=0 **then** *ok*

else *n*:= *n*-1.

s:= *s*+1. *m*:=*max m s. MovePile. s*:= *s*-1.

t:= *t*+1. *p*:= *p*+*s*. *ok.*

s:= *s*+1. *m*:= *max m s. MovePile. s*:= *s*-1.

n:= *n*+1 **fi**

其中规范 *MovePile* 是:

$n' = n$

$\wedge\ t' = t + 2^n - 1$

$\wedge\ s' = s$

$\wedge\ (m \geq s \Rightarrow m' = \max m(s+n))$

$\wedge\ p' = p + s\times(2^n-1) + (n-2)\times 2^n + 2$

----------空间结束

----------程序理论结束

第五章　程序设计语言

我们前面使用的是非常简单的程序设计语言, 组成部分只有 *ok*, 赋值句, **if then else fi**, 相关(顺序)组合以及被精化的规范。本章将进一步考虑一些流行语言中采用的标记, 充实我们的计算机指令系统。目前尚不考虑并行(独立组合)和通信(输入和输出)； 之后的章节会详细介绍它们。

5.0 作用域

5.0.0 变量说明(Variable Declaration)

在局部作用域中说明新状态变量的能力非常有用, 以至于每一个得体的程序设计语言都提供了此能力。变量说明的形式如下：

$$\textbf{var}\ x : T$$

其中 x 是被说明的变量, T 称作类型, 它指明 x 可以被赋值成什么值。参考本书最后一页的*优先次序表*，变量说明可以作用到其随后的部分。 在程序理论中, 每一个标记都应能作用到所有规范上而不仅仅是程序上, 这一点必不可少的。如此, 可以在精化一个局部变量的作用域之前，先提出一个本地变量作为程序设计过程的一部分。

我们可以将变量说明连同它将作用到的规范一起, 表示成一个带有初始和终结状态的双元表达式：

$$\textbf{var}\ x : T \cdot P \quad = \quad \exists\, x, x' : T \cdot P$$

规范 P 是一个表达式, 它包含所有非局部的(已经说明了的)变量的初始值和终结值加上新说明的局部变量。 规范 $\textbf{var}\ x{:}T \cdot P$ 是一个仅仅含有非局部变量的表达式。如果一个变量说明是可实现的, 它的类型一定非空。如下是一个简单的例子, 假设非局部变量 y 和 z 都是整数变量, 那么

$$\begin{aligned} & \textbf{var}\ x : int \cdot x{:=}2\ .\ y := x{+}z \\ = \ & \exists x, x' : int \cdot x' = 2 \wedge y' = 2{+}z \ \wedge z' = z \\ = \ & y' = 2{+}z \ \wedge \ z' = z \end{aligned}$$

根据变量说明的定义, 局部变量的初始值可以是它所属类型中的任何一个值。

$$\begin{aligned} & \textbf{var}\ x : int \cdot y{:=}x \\ = \ & \exists\, x, x' : int \cdot x' = x \ \wedge \ y' = x \ \wedge \ z' = z \\ = \ & z' = z \end{aligned}$$

该式表明 z 保持不变, 其中变量 x 没有出现是因为它是一个局部变量, 而 y 也没有出现是因为它的终结值未知。不过

$$\begin{aligned} & \textbf{var}\ x : int \cdot y := x{-}x \\ = \ & y' = 0 \ \wedge \ z' = z \end{aligned}$$

在某些语言中, 新说明的变量具有一个特殊值, 称为“未定义值”, 它不能出现在任何表达式中。为了将这类说明写成双元表达式, 这里引入一个“*undefined* ”（未定义）表达式, 但是不提供有关它的任何公理, 因此也就不能证明有关它的任何结论。

于是有

$$\textbf{var}\ x : T \cdot P = \exists x : undefined \cdot \exists x' : T, undefined \cdot P$$

对于这种类型的变量说明, 不必要求它的类型为非空。

用相同的方法很容易定义初始化赋值运算：

$$\textbf{var}\ x : T := e \cdot P = \exists x : e \cdot \exists x' : T \cdot P$$

假定 e 的类型为 T。

如果要计算空间的使用情况，一个变量说明必须在作用域开始处增加其空间变量 s 的值，而在作用域结束处将相应增值从 s 减去。

如同许多其它的程序设计语言, 这里可以在一个变量说明中同时说明几个变量。例如,

$$\textbf{var}\ x, y, z : T \cdot P = \exists x, x', y, y', z, z' : T \cdot P$$

----------变量说明结束

尽可能地把变量说明为局部变量是一条全球通用原则。如此，局部作用域之外的状态空间不会被不必要的变量干扰。局部作用域之内, 存在所有的非局部变量和相关的局部变量, 还存在许多用以记录局部性操作的其它变量。

5.0.1 变量悬挂

有时我们希望暂时把目光聚焦于状态子空间。如果状态子空间仅由变量 x 和 y 组成，这可用下面记号表示：

$$\textbf{frame}\ x, y$$

它可根据书本最后一页的*优先次序表*作用到其随后的表达式上, 就像 **var**。这个 **frame** 记号是“所有其它变量（甚至包括那些无法表明的，因为它们属于局部说明）不改变”的一种形式化说法。它类似于某些语言中的“输入” (import) 语句, 尽管并不完全一致。如果状态变量 w、z 不包括在框架 (frame) 里，那么形式化可表示为，

$$\textbf{frame}\ x, y \cdot P = P \wedge w' = w \wedge z' = z$$

P 内的状态变量是 x 和 y。它允许 P 使用 w 和 z，不过只作为局部常量（数学变量, 而非状态变量；没有 w'*或* z' ）。时间和空间变量隐含地假定为在所有的框架 (frame) 中，尽管可能没有明确地列出。

----------变量悬挂结束

ok 的定义和赋值句（使用状态变量）

$$ok = x' = x \wedge y' = y \wedge ...$$

$$x := e = x' = e \wedge y' = y \wedge ...$$

一定程度上非正式，三个点（省略号）代表“还有另外一些状态变量合取式”。如果我们已经定义了 **frame**，那么我们可以给出其形式化的写法：

$$ok = \textbf{frame} \cdot \top$$

$$x := e = \textbf{frame}\ x \cdot x' = e$$

前一章定义过表求和问题为 $s' = \sum L$。将 s 作为状态变量, L 作为常量。或许这里更倾

向于使用规范 $s{:=}\sum L$，以表明 s 保存正确的最终值, 并且所有其它变量不改变，但是我们的解中包括了一个变量 n, 它的值从 0 开始到$\#L$ 结束。我们现在给出其形式化的写法：

$$s := \sum L \;=\; \textbf{frame}\ s \cdot \textbf{var}\ n : nat \cdot s' = \sum L$$

首先将状态空间减小成 s, 如果之前 L 是状态变量, 现在它是常量； 接着再引入局部变量 n, 其后的处理同前。

----------作用域结束

5.1 数据结构

5.1.0 数组 (Array)

在大多数流行的程序设计语言中都有一种下标变量记号, 或索引变量记号, 通常叫做数组(Array)。数组的每一个元素都是一个变量。数组 A 的元素 2 被赋值成 3 可以表示为：

$$A(2) := 3$$

圆括号也可以用方括号替代； 这里暂且不考虑括号的问题。一个数组元素的赋值可以写成如下的初始状态和终结状态的双元表达式。令 A 为数组名, i 为索引类型的任意表达式, e 为元素类型的任意表达式, 那么

$$A\,i := e \;=\; A'\,i{=}e \;\wedge\; (\forall j \cdot j \neq i \Rightarrow A'\ j = A\,j\,) \;\wedge\; x'{=}x \;\wedge\; y'{=}y \;\wedge\; \ldots$$

上式表明， 赋值之后, A 中的元素 i 等于 e, 而 A 中所有其它元素保持不变, 所有其它变量也保持不变。如果对“撇号”（′）的位置没有把握, 参考下面的例子：

$$\begin{aligned} & A(A2) := 3 \\ = \;& A'\,(A2){=}3 \wedge\; (\forall j \cdot j \neq A2 \Rightarrow A'\,j{=}Aj\,) \;\wedge\; x'{=}x \;\wedge\; y'{=}y \;\wedge\; \ldots \end{aligned}$$

置换定律：

$$x := e\,.\,P = (\text{将 } P \text{ 中所有 } x \text{ 用 } e \text{ 置换})$$

是非常有用的，但不幸的是, 它不适用于数组元素。例如,

$$A2 := 3\,.\ i{:=}2\,.\ A\,i := 4\,.\ A\,i = A2$$

应当等于 ⊤, 因为 $i{=}2$ 刚好出现在最后的双元表达式之前, 而 $A2{=}A2$ 肯定等于 ⊤。如果我们尝试运用置换定律, 可得到

$$\begin{aligned} & A2{:=}3\,.\ i{:=}2\,.\ A\,i := 4\,.\ A\,i = A2 && \text{置换定律的无效使用} \\ = \;& A2{:=}3\,.\ i{:=}2\,.\ 4{=}A2 && \text{置换定律的有效使用} \\ = \;& A2{:=}3\,.\ 4{=}A2 && \text{置换定律的无效使用} \\ = \;& 4{=}3 \\ = \;& \bot \end{aligned}$$

下面是第二个对数组元素运用置换定律失败的例子：

$$A2{:=}2\,.\ A(A2){:=}3\,.\ A2{=}2$$

它应当等于 ⊥, 因为 $A2{=}3$ 出现在最后那个双元表达式之前。但置换定律表明

$$\begin{aligned} & A2{:=}2\,.\ A(A2){:=}3\,.\ A2{=}2 && \text{置换定律的无效使用} \\ = \;& A2{:=}2\,.\ A2{=}2 && \text{置换定律的无效使用} \\ = \;& 2{=}2 \\ = \;& \top \end{aligned}$$

当赋值句 := 左边是一个简单名时, 置换定律才有效。幸运的是我们总是可以将数组元素的赋值按这一要求重写：

$$
\begin{aligned}
& A\ i := e \\
= \ & A'\ i = e \ \wedge\ (\forall\ j \cdot j \neq i \Rightarrow A'\ j = A j) \wedge\ x' = x\ \ \wedge\ y' = y\ \wedge\ \ldots \\
= \ & A' = i \to e \mid A\ \wedge\ x' = x\ \wedge\ y' = y\ \wedge\ \ldots \\
= \ & A := i \to e \mid A
\end{aligned}
$$

现在重新看看置换定律不适用的那两个例子, 这次使用记号 $A := i \to e \mid A$。

$$
\begin{aligned}
& A := 2 \to 3 \mid A\ .\ \ i := 2\ .\ \ A := i \to 4 \mid A\ .\ \ A\ i = A 2 \\
= \ & A := 2 \to 3 \mid A\ .\ \ i := 2\ .\ \ (i \to 4 \mid A)\ i = (i \to 4 \mid A) 2 \\
= \ & A := 2 \to 3 \mid A\ .\ \ (2 \to 4 \mid A) 2 = (2 \to 4 \mid A) 2 \\
= \ & A := 2 \to 3 \mid A\ .\ \mathsf{T} \\
= \ & \mathsf{T}
\end{aligned}
$$

$$
\begin{aligned}
& A := 2 \to 2 \mid A\ .\ \ A := A 2 \to 3 \mid A\ .\ \ A 2 = 2 \\
= \ & A := 2 \to 2 \mid A\ .\ \ (A 2 \to 3 \mid A) 2 = 2 \\
= \ & ((2 \to 2 \mid A) 2 \to 3 \mid (2 \to 2 \mid A) 2 = 2 \\
= \ & (2 \to 3 \mid 2 \to 2 \mid A) 2 = 2 \\
= \ & 3 = 2 \\
= \ & \bot
\end{aligned}
$$

关于数组元素赋值唯一要记住的是： 在运用任何程序设计理论之前, 将 $A\ i := e$ 改为 $A := i \to e \mid A$ 。 二维数组元素赋值 $A\ ij := e$ 必须改成 $A := (\ i\ ;\ j\ ) \to e \mid A$， 多维数组元素赋值与其类似。

----------数组结束

5.1.1 记录 (Record)

不必引入任何新概念, 就可以建立记录，或称结构, 它与某些语言里的记录或结构相似。定义 *person* 如下：

person = “name” → *text*
| “age” → *nat*

说明（变量 *p* 为该记录类型）：

var *p* : *person*

并对 *p* 赋值如下：

p := “name” → “Josh” | “age” → 17

在包含记录（或结构）的语言中，一个成员(component)或域(field)的赋值类似于数组元素的赋值。例如：

p “age” := 18

正如置换定律不适用于数组元素赋值一样，它也不适用于记录的成员。解决的办法相同，可以重写成这样：

p := “age” → 18 | *p*

关于记录没有什么新理论。

----------记录结束

----------数据结构结束

5.2 控制结构

5.2.0 While 循环

在几种程序设计语言中, **while** 循环(While-loop)的语法类似于：

while *b* **do** *P* **od**

其中 *b* 是一个双元表达式, *P* 是规范。执行时, 先评估 *b* , 如果它的值是 ⊥ , 那么 **while** 执行完毕 ； 但如果它的值是 ⊤ , 那么执行 *P*, 然后再重头开始 **while** 循环。这里不把 **while** 循环用定义前面程序记号的方式定义为规范。而是，如果 W 是一个可执行的规范，我们把精化式

$$W \Leftarrow \textbf{while } b \textbf{ do } P \textbf{ od}$$

看成如下精化式的替代

$$W \Leftarrow \textbf{if } b \textbf{ then } P.W \textbf{ else } ok \textbf{ fi}$$

例如, 要证明

$s' = s + \sum L[n;..\#L] \ \wedge \ t' = t+\#L - n \Leftarrow$

while $n \neq \#L$ **do** $s := s+Ln \ . \ n := n+1 \ . \ t := t+1$ **od**

只需证明：

$s' = s+\sum L[n;..\#L] \ \wedge \ t' = t+\#L - n \Leftarrow$

if $n \neq \#L$ **then** $(s := s+Ln \ . \ n := n+1 \ . \ t := t+1 \ . \ s' = s+\sum L[n;..\#L] \ \wedge \ t' = t+\#L - n)$

else *ok* **fi**

在程序设计中, 我们可能刚好要精化一个规范 *W* 为 **if** *b* **then** (*P*.W) **else** *ok* **fi**。 这时, 可以用 **while** 循环来替代精化式。当调用的实现比较困难, 并且在这种情况下不使用分支指令时, 这种作法是相当有价值的。

while 循环的这种用途非常适合实际运用：它说明了在程序设计中如何使用它们。但它并不能证明所有我们所希望的内容，例如，不能证明

while *b* **do** *P* **od** = **if** *b* **then** *P* . **while** *b* **do** *P* **od else** *ok* **fi**

while 循环的另一个用途在第六章中给出。

练习 300（无限制的范围）：考虑下面包含自然数 x 和 y 的程序：

while $\neg x = y = 0$

do if $y > 0$ **then** $y := y - 1$

else $x := x - 1$. **var** $n : nat \cdot y := n$ **fi od**

该循环减小 y 直至为 0；然后将 x 减 1 并将任意自然数赋值给 y ； 然后又减小 y 直至为 0；然后又将 x 减 1 并将任意自然数赋值给 y ； 依此类推直到 x 和 y 都为 0 为止。下面的问题是寻找时间界。因此引入时间变量 t ，将循环重写为精化的形式：

$P \Leftarrow$ **if** $x=y=0$ **then** *ok*

else if $y>0$ **then** $y := y-1$. $t := t+1$. *P*

else $x := x-1$. $(\exists n \cdot y := n)$. $t := t+1$. *P* **fi fi**

执行时间依赖于 x， y 以及赋给 y 的任意自然数。这意味着 n 必须是非局部的，这样才能在规范 P 中表示它。但是一个非局部的 n 只能有一个任意的初始值，每次 x 减少时赋给 y，而在这里的计算中，每次 x 减少时 y 会被赋予一个不同的任意值。因此可以将 n 改成 x 的函数 f。（x 的值不重复，若重复，必须令 f 为一个时间的函数。）

令 $f : nat \to nat$，这里没有提到更多相关函数 f 的信息，因此它是一个从 nat 到 nat 的完全任意的函数。 f 的引入介绍了一种表示任意值的函数的方法，但不能表示何时，以及如何来选择那些任意值。令 $s = \sum f[0;..x]$，它表示 s 是 f 的前 x 个值的和。证明：

$t' = t + x + y + s \Leftarrow$ **if** x=y=0 **then** *ok*

else if y>0 **then** y:= y-1. t:= t+1. t' = t+x+y+s

else x:= x-1. y:= fx. t:= t+1. t' = t+x+y+s **fi fi**

证明分成三种情况：

$x = y = 0 \wedge ok$

$\Rightarrow x = y = s = 0 \wedge t' = t$

$\Rightarrow t' = t + x + y + s$

$y>0 \wedge (y := y-1.\ t := t+1.\ t' = t+x+y+s)$ 置换定律两次

$= y > 0 \wedge t' = t + 1 + x + y - 1 + s$

$\Rightarrow t' = t + x + y + s$

$x>0 \wedge (x := x-1.\ y := fx.\ t := t+1.\ t' = t+x+y+s)$ 置换定律三次

$= x > 0 \wedge y = 0 \wedge t' = t + 1 + x - 1 + f(x-1) + \sum f[0;..x-1]$

$\Rightarrow t' = t + x + y + s$

此程序的执行时间为 $x + y +$（x 个任意自然数的和）。

----------While 循环结束

5.2.1 包含退出 (Exit) 的循环

某些语言提供了一种从循环中跳出的命令。假设循环的语法是

do *P* **od**

而在 *P* 中允许有额外的语法

exit when *b*

其中 *b* 是双元表达式。有时使用 “break” 而不是 “exit”。与 5.2.0 小节一样, 可以将带有退出语句的循环精化式当作一个可选择使用的记号。例如, 如果 *L* 是个可实现的规范， 那么

L ⇐ **do** *A*.

exit when *b*.

C **od**

另外一种表达为

L ⇐ *A*. **if** *b* **then** *ok* **else** *C*. *L* **fi**

程序员在使用循环结构编程时, 有时会发现经过若干层嵌套循环之后就可以达到想要的目的。 问题是如何从循环中出来。可以引入一个二元变量, 用以记录目标是否达到，在每

次循环迭代时进行测试，以决定循环是继续还是退出执行。 也可以用 **go to** 语句直接跳出所有的循环, 免去所有的测试。 或者程序设计语言为这种情况提供一种专门的 **go to** 操作：**exit** *n* **when** *b* 表示当 *b* 满足时退出 *n* 层循环。例如, 也许有以下情况：

P ⇐ **do** *A*.
  do *B*.
    exit 2 **when** *c*.
    D **od**.
  E **od**

对某个适当定义的 *Q*，与该循环相对应的精化式结构如下：

P ⇐ *A*. *Q*
Q ⇐ *B*. **if** *c* **then** *ok* **else** *D*. *Q* **fi**

一般建议每个循环有一个规范，但循环结构不对此进行要求，而精化结构要求这一点。

前面的那个例子有一个深层退出, 但没有浅层退出, 它将 *E* 丢在一个死区中永远不会执行。下面是一个同时具有深层和浅层退出的例子：

P ⇐ **do** *A*.
  exit 1 **when** *b*.
  C.
  do *D*.
    exit 2 **when** *e*.
    F.
    exit 1 **when** *g*.
    H **od**.
  I **od**

对某个适当定义的 *Q*，与此对应的精化式结构为：

P ⇐ *A*. **if** *b* **then** *ok* **else** *C*. *Q* **fi**
Q ⇐ *D*. **if** *e* **then** *ok* **else** *F*. **if** *g* **then** *I*. *P* **else** *H*. *Q* **fi fi**

带有退出的循环总可以容易地转换成一个精化式结构, 不过, 反过来就不成立了； 在将精化式结构转换成带有退出的循环代码时, 有些精化式结构需要引入一些新变量, 甚至需要引入整个数据结构来对其进行编码。

----------退出循环结束

5.2.2 二维查找

为了说明前一小节所述内容，这里来做 练习 185： 写一个程序，在一个二维数组中查找一个指定的项。执行时间必须与维数值的乘积呈线性关系。

令数组为 *A*, 令它的维数值分别为 *n* 和 *m*, 并且令查找的项为 *x*。自然数变量 *i* 和 *j* 的最终值将指出 *x* 在 *A* 中的位置。如果 *x* 出现不止一次，任何一处位置都可被指出。如果 *x* 没有出现，将 *i* 和 *j* 分别赋值为 *n* 和 *m*。不考虑时间，令原问题为 *P*，有

P = **if** *x*: *A*(0,..*n*)(0,..*m*) **then** *x* = *Ai*2*j*2 **else** *i*2=*n* ∋ *j*2=*m* **fi**

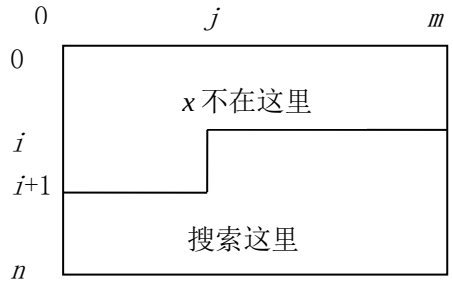
可以先搜索行 0，然后搜索行 1，依次下去。因此，定义 *Q* 表示从行 *i* 开始搜索：

$$Q \;=\; \textbf{if } x\colon A(i,..n)(0,..m) \textbf{ then } x = Ai2j2 \textbf{ else } i2{=}n \ni j2{=}m \textbf{ fi}$$

在每一行中，依次查找每一列元素，所以定义 R 表示从行 i 之中的列 j 开始搜索：

$$R \;=\; \textbf{if } x\colon Ai(j,..m),\ A(i{+}1,..n)(0,..m) \textbf{ then } x = Ai2j2 \textbf{ else } i2{=}n \ni j2{=}m \textbf{ fi}$$

表达式 $Ai(j,..m),\ A(i{+}1,..n)(0,..m)$ 表示下图中的底部区域：



现在就可以用 5 个简单的步骤解决这个问题：

$P \;\angle\; i{:=}\,0.\ \ i{\le}n \;®\; Q$

$i{\le}n \;®\; Q \;\angle\; \textbf{if } i{=}n \textbf{ then } j{:=}\,m \textbf{ else } i{<}n \;®\; Q \textbf{ fi}$

$i{<}n \;®\; Q \;\angle\; j{:=}\,0.\ \ i{<}n \ni j{\le}m \;®\; R$

$i{<}n \ni j{\le}m \;®\; R \;\angle\; \textbf{if } j{=}m \textbf{ then } i{:=}\,i{+}1.\ \ i{\le}n \;®\; Q \textbf{ else } i{<}n \ni j{<}m \;®\; R \textbf{ fi}$

$i{<}n \ni j{<}m \;®\; R \;\angle\; \textbf{if } A\ i\ j = x \textbf{then } ok \textbf{ else } j{:=}\,j{+}1.\ \ i{<}n \ni j{\le}m \;®\; R \textbf{ fi}$

当仅保留执行所需的足够信息时，这个程序的执行模式就很容易看出来了。上面的这个非程序规范可用于理解题目意图，和证明，但不能用于执行。对一个编译器来说，该程序应如下所示：

$P \;\angle\; i{:=}\,0.\ \ L0$

$L0 \;\angle\; \textbf{if } i{=}n \textbf{ then } j{:=}\,m \textbf{ else } j{:=}\,0.\ \ L1 \textbf{ fi}$

$L1 \;\angle\; \textbf{if } j{=}m \textbf{ then } i{:=}\,i{+}1.\ \ L0$

$\textbf{else if } A\ i\ j = x \textbf{ then } ok$

$\textbf{else } j{:=}\ j{+}1.\ \ L1 \textbf{ fi fi}$

用 C 语言，程序应如下：

```
        i = 0 ;
    L0:  if (i==n) j=m;
         else {      j=0 ;
                L1: if (j==m) { i=i+1; goto L0;}
                    else if (A [i][j]==x);
```

else { j =j +1 ; goto L1;} }

如果想要加上递归时间，可以在 $i := i+1$ 和 $j := j+1$ 之后加上 $t := t+1$。或者更聪明的做法是在测试 $j = m$ 前进行一次时间的增加就可以了。这里也修改了那 5 个要求精化的规范，加上了时间。剩余的时间最多为剩余的将被搜索的面积。

$t' \le t + n \times m \Leftarrow i{:=}0.\ i \le n \Rightarrow t' \le t + (n-i) \times m$

$i \le n \Rightarrow t' \le t + (n-i) \times m \Leftarrow$ **if** $i=n$ **then** $j{:=}\ m$ **else** $i<n \Rightarrow t' \le t + (n-i) \times m$ **fi**

$i<n \Rightarrow t' \le t + (n-i) \times m \Leftarrow j{:=}0.\ i<n \wedge j \le m \Rightarrow t' \le t + (n-i) \times m - j$

$i<n \wedge j \le m \Rightarrow t' \le t + (n-i) \times m - j \Leftarrow$

$t{:=}\ t+1.$

if $j=m$ **then** $i{:=}\ i+1.\ i \le n \Rightarrow t' \le t + (n-i) \times m$

else $i<n \wedge j<m \Rightarrow t' \le t + (n-i) \times m - j$ **fi**

$i<n \wedge j<m \Rightarrow t' \le t + (n-i) \times m - j \Leftarrow$

if $A\ i\ j = x$ **then** *ok*

else $j{:=}\ j+1.\ i<n \wedge j \le m \Rightarrow t' \le t + (n-i) \times m - j$ **fi**

----------二维查找结束

5.2.3 For 循环

这里用以下语法定义 for 循环（for Loop）：

for $i := m;..n$ **do** P **od**

其中 i 是新变量，m 和 n 是整数表达式，满足 $m \le n$，P 是一个规范。这是受控循环的最典型写法，它与流行语言中对应成分的差别是：循环继续执行一直到 $i=n$，但不包括 $i=n$。为避免一些麻烦，不把 i 作为状态变量(所以它不能在 P 中赋值)。并且 m 和 n 的初始值控制着循环（所以共有 $n - m$ 个循环）。

与前面介绍的循环结构一样，这里也不把 for 循环定义成一个规范，而只说明如何在精化式中使用它。令 F 为两个整数变量的函数，其结果为一个可实现的规范，那么

$Fmn \Leftarrow$ **for** $i := m;..n$ **do** P **od**

是以下三个精化式的替代式

$Fii \ \angle\ m \le i \le n \wedge ok^{[\,]}_{[\,]}$

$Fi(i+1) \ \angle\ m \le i < n \wedge P^{[\,]}_{[\,]}$

$Fik \ \angle\ m \le i < j < k \le n \wedge (Fij.\ Fjk)$

若 $m = n$ 则没有迭代，那么不须做任何事($ok^{[\,]}_{[\,]}$) 就可满足 Fmn。循环的主体必须运行一次迭代 $Fi(i+1)$。最终，Fmn 也必须被首先从 m 到一个中间索引 j 的迭代，再从 j 到 n 的迭代所满足。

例如，令状态包含整数变量 x，且 F 定义为：

$$F = \langle i, j: nat \rightarrow x' = x \times 2^{j-i} \rangle$$

那么可以用以下两个精化式来解指数问题 $x' = 2^n$ ：

$x'=2^n \ \angle \ x:= 1. F0n$

$F0n \ \angle$ **for** i:= 0;..n **do** x:= 2×x **od**

第一个精化式可用置换定律证明。要证明第二个精化式，必须先证以下三个定理：

$Fii \ \angle \ 0 \le i \le n$ ə ok

$Fi(i+1) \ \angle \ 0 \le i < n$ ə (x:= 2×x)

$Fik \ \angle \ 0 \le i < j < k \le n$ ə (Fij. Fjk)

证明这三个定理很简单。

递归时间的测量要求在每一个循环中包含一个时间增长，每次至少增加一个时间单位。一般而言，for 循环体占用的时间可能是迭代 i 的一个函数 f。使用 $t' = t + \sum i : m,..n \cdot f\, i$ 作为 for 循环规范 Fmn，根据 for 循环的规则：

$$t' = t + \Sigma i{:}m,..n{\cdot}\ fi \ \angle \ \textbf{for}\ i{:=}m;..n\ \textbf{do}\ t' = t{+}fi\ \textbf{od}$$

如果循环体需要的时间为常量 c，那么上式可简化成：

$$t' = t + (n-m)\times c \ \angle \ \textbf{for}\ i{:=}m;..n\ \textbf{do}\ t' = t{+}c\ \textbf{od}$$

对 for 循环规则的一个典型使用是对表中的每个项进行处理。例如，练习 305 要求对表中的每个项加 1。规范为：

$$\#L' = \#L \wedge \forall i : 0,..\#L \cdot L'i = Li + 1$$

现在需要一个规范 Fik 来表示迭代的任意子段：从索引 i 到索引 k 的每一项进行加 1。

$$Fik = \#L' = \#L \wedge (\forall j : i,..k \cdot L'\, j = Lj + 1) \wedge (\forall j : (0,..i),(k,..\#L) \cdot L'\, j = Lj)$$

为证明

$$F0(\#L) \ \angle \ \textbf{for}\ i{:=}0;..\#L\ \textbf{do}\ L{:=}\ i{\square}Li{+}1 \mid L\ \textbf{od}$$

我们必须证明三个定理：

$Fii \ \angle \ 0 \le i \le \#L \ \wedge\square ok$

$Fi(i+1) \ \angle \ 0 \le i < \#L \ \wedge\square (L:= i \rightarrow Li+1 \mid L)$

$Fik \ \angle \ 0 \le i < j < k \le \#L \ \wedge\square (Fij.\ Fjk)$

有时for循环的规范 Fmn 具有如 $I\ m \Rightarrow I'n$ 的形式，其中 I 是一个变量的函数，其结果是前置条件，而 I' 也是一个函数，其结果是相应的后置条件。当 I 应用于for循环的索引上时，条件 Ii 称为不变式(invariant)。这种规范形式的一个好处在于 $Fii \angle ok$ 和 $Fik \Leftarrow (Fij.Fjk)$ 都能自动满足。并非所有的for循环规范都能表示成这种形式；不论是时间还是前面的例子（把每个项加1）都不能表示成这种形式。但前面的指数例子可以，定义：

$$I = \langle i: nat \square x = 2^i \rangle$$

那么求解结果为：

$$x' = 2^n \Leftarrow x := 1.I0 \Rightarrow I'n$$

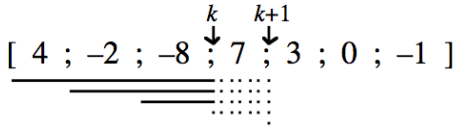
$$I0 \Rightarrow I'n \Leftarrow \textbf{for } i := 0;..n \textbf{ do } Ii \Rightarrow I'(i+1) \textbf{ od}$$

$$Ii \Rightarrow I'(i+1) \Leftarrow x := 2 \times x$$

另一个使用 for 循环规则的不变式形式的例子是练习 220(a)：给定一个整数组成的表，可能含有负数，编写一个程序寻找某个段(连续项组成的子表)，其项之和为最小。令 L 为给定的整数表，问题 P 可形式化表示为：

$$P = s' = MIN\ i,j \cdot \sum L\ [i;..j]$$

其中 $0 \le i \le j \le \#L$。条件 Ik 表示 s 是直到索引 k 处为止的所有段的最小和。对 k=0，只有一个段，空段，它的和是 0。当 k=#L 时，说明已检查了所有的段，因而就得到所需要的解。从 Ik 到 $I(k+1)$，必须检查结束于索引 k+1 处的那些段。可以找出这些新段的项之和的最小值，与原 s 比较，将较小者作为 s 的新值。不过，还有更好的方法。其实每个以索引 k+1 结束的段仅比以索引 k 结束的段多一个项。其中有一个例外： 以索引 k+1 结束的空段 。如图所示：



因此，如果知道了以索引 k 结束的任意段的最小和 c，那么以索引 k+1 结束的任意段的最小和就是 $min\ (c+Lk)\ 0$。因此，定义如下：对 $0 \le k \le \#L$，

$$I\ k = s = (MIN\ i : 0,..k+1 \cdot MIN\ j : i,..k+1 \cdot \sum L[i;..j])$$

$$\wedge\ c = (MIN\ i : 0,..k+1 \cdot \sum L\ [i;..k])$$

现在，程序就容易写出来了：

$$P \Leftarrow s:=0.\ c:=0.\ I\ 0 \Rightarrow I'\ (\#L)$$

$$I\ 0 \Rightarrow I'\ (\#L) \Leftarrow \textbf{for } k:=0;..\#L \textbf{ do } I\ k \Rightarrow I'(k+1) \textbf{ od}$$

$$I\ k \Rightarrow I'\ (k+1) \Leftarrow c:= min\ (c + L\ k)0.\ s:= min\ c\ s$$

----------For 循环结束

5.2.4 转向 (Go To)

假设4.2.6小节的快速求幂程序$z2=x^y$ 被重写为以下形式（用冒号进行标注）。

A:(*z*:=1. **If** *even y* **then go to** *C*

else *B*: (*z*:= *z*×*x*. *y*:= *y*–1.

C: **if** *y*=0 **then go to** *E*

else *D*: (*x*:= *x*×*x*. *y*:= *y*/2.

if *even y* **then go to** *D* **else go to** *B* **fi) fi) fi**).

E: *ok*

在这个程序里，需要证明以下几项：

$A \angle z{:=}1.$ **if** *even y* **then** *C* **else** *B* **fi**

$B \angle z{:=}\ z{\times}x.\ y{:=}\ y{-}1.\ C$

$C \angle$ **if** $y{=}0$ **then** *E* **else** *D* **fi**

$D \angle x{:=}x{\times}x.\ y{:=}y/2.$ **if** *even y* **then** *D* **else** *B* **fi**

$E \angle ok$

go to语句带来的麻烦在于，和循环结构类似，引入一些在程序建构过程中没有记录的规范。在这个例子里，这些规范是： $A = z2{=}x^y$, $B = odd\ y \mathbin{\text{®}} z2{=}z{\times}x^y$, $C = even\ y \mathbin{\text{®}} z2{=}z{\times}x^y$, $D = even\ y \ni y{>}0 \mathbin{\text{®}} z2{=}z{\times}x^y$, 以及 $E{=}ok$ 。

----------转向结束

----------控制结构结束

5.3 时间与空间依赖

某些程序设计语言提供了时钟, 或延迟, 或其他依赖时间的特征。在前面的例子中就使用过时间变量作为辅助变量, 而不影响计算过程。它被用作理论的一部分，用来证明有关执行时间的一些结论。如果仅用于这个目的, 就无需在计算机中表示时间。但是如果在计算中有一个可读时钟作为时间来源, 就可用它来影响计算。赋值 $deadline := t + 5$ 是允许的，等同于 **if** $t \le deadline$ **then** …. **else** …，但赋值 $t := 5$ 不被允许。我们可以查看时钟，但不能以一个任意值重置它；所有的时钟变化必须伴随时间的推移（根据某种测量方法）。（一个计算机操作可能有时需要设置时钟，但它不属于程序设计理论的范畴。）

有时我们也许想要说明时间推移。例如, 如果想要计算“等待, 直至时间为 *w*”。引入一个记号, 将这一问题形式化定义为：

wait until *w* = $t := max\ t\ w$

由于时间不能被重置, 因此 $t := max\ t\ w$ 在精化之前不能作为一个可接受的程序。令时间是一个扩展自然数，并使用递归时间, 有

wait until *w* ⇐ **if** $t \ge w$ **then** *ok* **else** $t{:=}t{+}1.$ **wait until** *w* **fi**

这就获得了一个“忙式等待”循环。可以分情况证明这个精化式。首先,

$t \ge w \ \wedge\ ok$

$= \quad t \ge w \ \wedge\ (t := t)$

$\Rightarrow\ t := max\ t\ w$

第二步,

$t<w \ \wedge \ (t:=t+1 \ . \ t:=max \ t \ w)$

在左侧合取式中, 利用 t: $xnat$；在右侧合取式中, 利用置换定律。

$= \quad t+1 \le w \ \wedge \ (\ t := max \ (t+1) \ w \ )$

$= \quad t+1 \le w \ \wedge \ (\ t := w \ )$

$= \quad t< w \ \wedge (\ t := max \ t \ w \ )$

$\Longrightarrow \ t:= max \ t \ w$

在依赖于时间的程序中，我们必须使用真实时间度量，而不是递归时间度量。对于在什么位置进行时间的增加也需更加地小心。并且，需要对 **wait until** w 进行一个略微不同的定义，将它留作练习 312(b)。

如时间变量 t 一样，空间变量 s ，到目前为止也只是用于证明空间使用的情况，而不影响计算。但如果一个程序包含可用的空间使用信息，那么使用该信息也没什么坏处。如 t 一样， s 也可以读但不能任意地写。所有对 s 的改变必须与空间使用情况相对应。

----------时间与空间依赖结束

5.4 断言（Assertions)　　可选节

5.4.0 检查

作为安全检查, 一些程序设计语言含有语句

assert b

其中 b 是二元值，大意类似于“相信 b 是真的”。如果它出现在一个程序或方法的开头，它可以用一个词**前置条件（precondition）**来表示；如果出现在最后，它可用一个词**后置条件(postcondition)**来表示；如果它出现在一个循环的开头或结尾，它可用一个词**不变式（invariant)**来表示；它们都有一样的结构。这个语句的执行是通过检查 b 是否为真；如果为真, 执行正常进行下去；如果为假, 打印一条错误信息, 执行被中断。使用这种断言语句的目的为：在一个正确程序中, 断言表达式将总为真, 因而所有断言语句是多余的。不过所有的错误检查都免不了多余，而断言可以帮我们发现错误，并且防止以后损害状态变量。但这个不是无偿的；它需要消耗执行时间。

断言的定义如下：

assert b = **if** b **then** ok **else** $print$ "error". **wait until** ∞ **fi**

如果 b 为真, **assert** b 与 ok 作用一样；如果 b 为假, 执行在有限时间内不能进行后续动作。因此断言是使程序更加强健的一种简单方法。

----------检查结束

5.4.1 回溯

如果 P 和 Q 都是可实现的规范, 那么 $P \vee Q$ 也是。要实现它, 只要选择满足 P 和 Q 中的任何一个就可以了。通常情况下以一个精化的形式进行选择， $P \vee Q \Leftarrow P$ 或者 $P \vee Q \Leftarrow Q$ 。在实现中把析取作为一个程序连接, 可以节省程序设计步骤，可能用记号 **or**

表示 。例如,

$$x{:=}0 \ \ \textbf{or} \ \ x{:=}1$$

是一个程序, 它的执行将 x 赋值为 0 或 1。它将析取的选择留给了程序设计语言的实施者。

下一个构造从根本上改变了编程方式。引入记号：

$$\textbf{ensure}\ b$$

其中 b 为二元值, 意味着“使 b 为真”。定义如下：

$$\textbf{ensure}\ b \ = \ \textbf{if}\ b\ \textbf{then}\ ok\ \ \textbf{else}\ b' \wedge ok\ \textbf{fi}$$
$$= \ b' \wedge ok$$

与 **assert** b 一样, **ensure** b 当 b 为真时等于 ok。但当 b 为假时, 就出现了问题：这个计算必须在不改变任何内容的前提下使 b 为真。这个构造是不可实现的(除非 b 恒等于真)。不过, 当它与其他结构合起来作用时, 整个式子也许就变成可实现的。不妨看看下面的例子, x 和 y 都是变量：

$$x{:=}0 \ \textbf{or}\ x{:=}1.\ \textbf{ensure}\ x{=}1$$
$$= \ \exists x'', y'' \cdot (x''{=}0 \wedge y''{=}y \ \vee \ x''{=}1 \wedge y''{=}y) \ \wedge \ x''{=}1 \ \wedge \ x' {=}x'' \ \wedge \ y' {=}y''$$
$$= \ x'{=}1 \ \wedge \ y'{=}y$$
$$= \ x{:=}\ 1$$

虽然实现可以选择 $x{:=}0$ 和 $x{:=}1$ 中任何一个, 但这个选择必须满足其后的条件。为此, 采用的方法是：首先任意选择一个(与平常一样), 然后检查其后的 **ensure** 条件, 若发现条件为假, 就必须回溯, 再选择另外一个。因为选择项可以嵌套, 所以必须做许多簿记。

一些流行的程序设计语言, 比如 Prolog, 支持回溯。它们可能规定选择必须按某种特定顺序进行(这里省略了这些复杂的规定)。对这类语言有两点警告：第一, 程序员必须保证程序是可实现的；这一点语言本身无法保证。换句话说，实现不能保证所作的计算满足程序要求, 因为有时这个程序要求是不可满足的。第二, 时间和空间的计算无效。

----------回溯结束

----------断言结束

5.5 子程序

5.5.0 Result 表达式

令 P 为一个规范, e 是不含撇号变量组成的表达式。那么,

$$P\ \textbf{result}\ e$$

是初始状态的一个表达式。它表示执行 P 之后再计算 e 所得的结果。下式为自然对数基的一个表达方式：

var *term*, *sum*: *rat*:=1·
for *i*:=1;..15 **do** *term*:= *term*/*i*. *sum*:= *sum*+*term* **od**
result *sum*

局部变量 *term* 和 *sum* 的作用域延伸到 **result** 表达式的结束。

有关 **result** 表达式的公理为：

$$P.\ (P\ \textbf{result}\ e)=e$$

只不过，当运用置换定律时，(*P* **result** *e*) 不适用于相关组合中的加双撇，也不适用于置换定律中的置换。例如：

T　　运用**result**表达式的公理

= $x:= x+1.\ (x:= x+1\ \textbf{result}\ x)=x$　　运用置换定律, **result**表达式保持不变

= $(x:= x+1\ \textbf{result}\ x) = x+1$

其结果好像是先执行赋值句 $x:=x+1$, 所得的 *x* 值就是结果，但是变量 *x* 的值并没有改变。

$y:= (x:= x+1\ \textbf{result}\ x)$　　根据之前的计算

= $y:= x+1$

表达式*P* **result** *e*可用以下途径实现。把*P*内每个非局部变量以及*P*内被赋值的*e*替换成全新的局部变量，这个局部变量的初始值是相应非局部变量的值。然后执行修改过的*P*并计算修改过的*e* 。

在一些编程语言的实现过程中，对于有关这个目的而进行的全新局部变量的引入并没有完全成熟。因此表达式的估算可能导致状态变化。这个估算表达式所造成的状态变化被称为“副作用”。由于副作用的存在，数学推理变得不再可行。例如，我们不能说 $x+x = 2\times x$, 甚至不能说 $x=x$, 因为 *x* 可能定义为 $(y:=y+1\ \textbf{result}\ y)$，每一个估算都会得出一个比前面的估算大 1 的整数。副作用不难避免；如果语言的实现无法引入必要的局部变量，程序员可以引入必要的局部变量。一些编程语言禁止在表达式中对非局部变量赋值，因此程序员有义务引入必要的局部变量。如果一个编程语言允许副作用，必须在运用任何定理之前，除掉副作用。比如，

$x:= (P\ \textbf{result}\ e)$ 变成 $(P.\ x:=e)$

其中先将 *P* 中局部变量作必要的换名, 避免与非局部变量冲突, 然后将 *P* 中变量说明的作用域延伸到 $x:= e$ 上。另有一例,

$x:= (P\ \textbf{result}\ e) + y$ 变成 $(\textbf{var}\ z:= y\cdot\ P.\ x:= e+z)$

具有类似的附带条件。

计算表达式的时间在使用递归时间测量中是忽略不计的。这样做对某些应用来说很合理，其表达式只包含少数已用计算机硬件实现的操作。但对包含了非硬件实现的操作 (例如表联接) 的表达式，这样做值得怀疑。而对包含循环的 **result** 表达式, 这样做就不合理了。而允许 **result** 表达式对时间变量进行增加是有副作用的，因此我们可以这样处理：为了计算时间界，首先在 **result** 表达式中包含时间。然后从结果表达式中除去时间变量（为了去除副作用），然后在包含了 **result** 表达式的程序中进行时间的增加。

----------Result 表达式结束

5.5.1 函数（Function）

在许多流行的程序设计语言中, 函数是关于有关结果的断言，函数名字，参数, 作用域控制以及 **result** 表达式的一个组合。它是一堆“包装起来的东西”。例如，若用 C 写一个二进制指数函数, 是这样的：

```
int bexp(int n)
{   int r=1;
    int i;
    for (i=0; i<n; i++) r = r*2;
     return r; }
```

而使用本书的标记，是这样的：

bexp = ⟨ *n*: *int* →

var *r*: *int* := 1 ⨾

for *i*:= 0;..*n* **do** *r*:= *r*×2 **od**.

assert *r*: *int* ⨾

result *r* ⟩

将这些程序设计的特性分开表示是为了分别地理解它们。它们可以以任何理想的方式进行组合，如在例子中一样。为这种组合提供一种构造的坏处是它太复杂了。使用这些计算机语言的程序员可能无法分离这类问题， 从而无法意识到命名, 参数, 断言, 局部作用域以及 **result** 表达式这些成分独立使用的实用性。

甚至本书所使用的函数形式也可以被简化和通用化。一个参数定义域的说明是公理引用的一种特殊情况, 它可以从名字引用中分离出来(参见练习 109)。

----------函数结束

5.5.2 过程（Procedure）

过程(或称空函数、或方法), 与许多语言中定义的一样, 类似于函数, 也是“包装起来的东西”。它组合了名称说明, 参数与局部作用域。前一小节中的有关评述这里仍然适用, 不过也存在一些新问题。

为了将理论用于程序开发, 而不只用于程序验证, 必须能够讨论那些过程体还是尚未精化的规范的过程，而不是一个程序。例如, 我们可能希望带有参数 *x* 的过程 *P* 定义如下：

$$P = \langle x\text{: } int \rightarrow a' < x < b' \rangle$$

它对变量 *a* 和 *b* 赋值, 所赋的值分别小于和大于所提供的实参。在对其体精化前, 可以使用过程 *P*。例如,

$$P3 = a' < 3 < b'$$

$$P(a{+}1) = a' < a{+}1 < b'$$

过程体可以简单地精化成

$$a' < x < b' \Leftarrow a{:=}\, x - 1.\ \ b{:=}\, x{+}1$$

过程体怎样精化都不改变 *P* 的定义；当使用 *P* 时，并不使用其精化式。使用者不需要知道

实现过程，实现者也不需要知道如何使用。

过程和实参可以转化成一个局部变量和其初始值。

$$\langle p{:}\ D\to B\rangle\ a \ =\ (\textbf{var}\ p{:}\ D{:=}\ a\cdot B) \qquad \text{如果} B \text{ 不使用} p' \text{ 或} p{:=}$$

这种转换表明一个参数只是一个局部变量, 其初始值将由实参提供。在许多流行的程序设计语言中, 情况正是这样。这是规范和实现的一个不幸混淆。创建参数的决定, 和其定义域的选择是过程规范的一部分, 并且是过程使用者所感兴趣的。创建局部变量的决定，和其定义域的选择通常是精化式的一部分, 是过程实现的一部分, 并且不应被过程的使用者关心。当一参数在过程体中被赋了一个值时, 它只起了一个局部变量的作用, 已不再与其先前的参数角色有任何联系。

另一种类型的参数, 通常称之为引用参数或 **var** 参数，代表一个作为实参的未知非局部变量。这里有一个例子，使用⟨* ⟩表示引用参数的引入：

$$\langle *x{:}\ int\to a{:=}\ 3.\ \ b{:=}\ 4.\ \ x{:=}\ 5\rangle\ a$$

$$=\quad a{:=}3.\ \ b{:=}4.\ \ a{:=}5$$

$$=\quad a'{=}5\wedge b'{=}4$$

当过程是纯程序时才可运用引用参数，而不可运用任何其他规范记号。根据上面的例子，如果我们把其写为

$$\langle *x{:}\ int\to a'{=}\ 3\wedge b'{=}\ 4\wedge x'{=}\ 5\rangle\ a$$

我们便不能仅仅把 x 替换成 a ，也不能仅仅把 x'替换成 a' 。并且，直到过程被运用于实参之前，我们不能进行有关过程体的任何推理。下面的例子和前面的例子有一样的过程体，

$$\langle *x{:}\ int\to x{:=}\ 5.\ \ b{:=}\ 4.\ \ a{:=}\ 3\rangle\ a$$

$$=\quad a{:=}5.\ \ b{:=}4.\ \ a{:=}3$$

$$=\quad a'{=}3\wedge b'{=}4$$

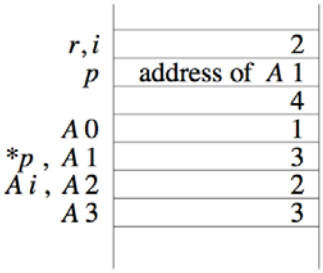
但是结果不同。引用参数阻碍了规范的运用，并且它们阻碍了有关过程本身的任何推理。我们必须对每个调用（call）分别应用编程理论。这一点与过程的目的相矛盾。

----------过程结束

----------子程序结束

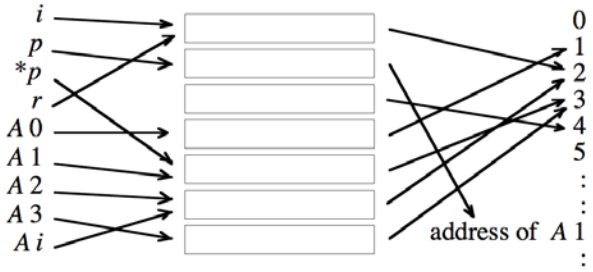
5.6 别名（Alias） 可选节

许多流行的程序设计语言所介绍的计算模式，都包含一块由许多单个存贮单元组成的内存。每一单元有一个值。通过程序设计语言, 存贮单元都有了名称。下面是一种标准图样：

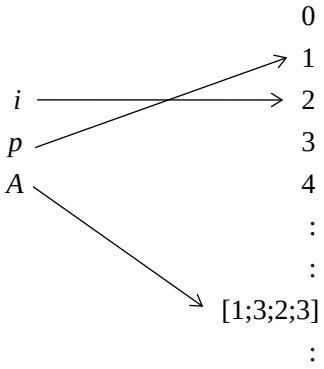


在这个图中, *p* 是一个指针变量, 它当前指向数组元素 *A* 1, **p* 是 *p* 解除引用的结果；所以**p* 和 *A* 1 指向相同的内存单元。 因为变量 *i* 的当前值是 2, 因此 *A i* 和 *A* 2 也指向相同的内存单元。 *r* 是一个引用参数, 而变量 *i* 是它的实参, 所以 *r* 和 *i* 也就指向了相同的内存单元。一个内存单元可以有零个，一个，两个或多个名。当一个内存单元在同一时刻具有两个或多个可见的名时, 这些名被称为“别名(Alias)”。

显然，对于数组和引用参数, 别名的使用会防碍程序设计理论的应用。因而一些程序设计语言禁止使用别名。不幸的是很难检测到别名的使用, 尤其是在规范被完全精化成程序之前的程序构造期间。对大多数人来说, 禁止和限制都是令人讨厌的。为了避免禁止, 可以有一个选择： 将程序设计理论复杂化, 使之可以处理使用别名的情况；或者简化计算模型, 消除出现别名的可能性。如果将上面的图稍微重画一下, 可以看见存在两个映射， 一个从名到内存单元, 一个从内存单元到值：



每一个赋值句诸如 *A* 3 内的 *p* := address 或 *i* := 4 可立即同时改变上述两个映射。对一个名的赋值可以间接地改变另一个名指向的值。为了简化这个映射图并且消除使用别名的可能性, 去掉内存单元且允许使用一个更为广泛的值空间。下面是改变 后的新图：



指针变量可以被一个结构的相关索引变量替换, 这样指针变量就可以实现成地址。如果函数可以返回一个结构值，那么引用参数就不必要了。这个简单图已经足够用, 别名的问题也消失了。

5.7 概率程序设计 可选节

根据传统，概率理论中的概率(probability)是一个 0 到 1 之间（包括 0 和 1）的实数：

$$prob = \S\ r : real \cdot 0 \le r \le 1$$

其中 1 表示“肯定为真”，0 表示“肯定为假”，1/2 表示“相等的可能性为真或为假”，等等。相应地，仅在本节中，增加公理：

$$\mathsf{T} = 1$$

$$\bot = 0$$

有了这些公理，可以用算术方法表达二元运算符。例如，$\neg x = 1 - x$, $x \wedge y = x \times y$, 以及 $x \vee y = x - x \times y + y$ 。

一个分布（distribution）是一个表达式，它的值（对其变量的所有取值）是一个概率，它们的和（在其变量所有取值下）为 1。（为简化起见，只考虑二元变量和整数变量的分布；对于有理数变量和实数变量，相加会变成整数。）例如，若 $n : nat + 1$，则 2^{-n} 是一个分布，因为：

$$(\forall n : nat + 1 \cdot 2^{-n} : prob) \wedge (\sum n : nat + 1 \cdot 2^{-n}) = 1$$

如果有两个变量 $n, m : nat + 1$，那么 2^{-n-m} 是分布，因为：

$$(\forall n, m : nat + 1 \cdot 2^{-n-m} : prob) \wedge (\sum n, m : nat + 1 \cdot 2^{-n-m}) = 1$$

一个分布可用于表示其中的变量的值出现的频率。例如，2^{-n} 表示 n 的取值为 3 的频率是八分之一。分布 2^{-n-m} 表示 n 的值为 3 且 m 的值为 1 的状态出现的频率是十六分之一。分布也可被用于描述所期许的值，或者预测其变量。2^{-n} 的分布表示 n 的值为 1 的几率和其值不为 1 的几率相等；其值为 1 的几率是其值为 2 的几率的两倍。分布也可被用于指定每个变量的值具有我们想要的几率。

假设有自然数变量 n 。规范 $n' = n + 1$ 说明说明对 n 的任意给定的初值，n' 的值总为 $n + 1$。换句话说，说明了最终值 n' 等于初始值 $n + 1$ 的几率为 1，等于任意其他值几率为 0 。对于任意值的 n 和 n'，$n' = n + 1$ 的值是 T（1）或者⊥（0），所以它是一个概率。规范 $n' = n + 1$ 不是 n 和 n' 的分布，因为有无限多对值使得 $n' = n + 1$ 的值为 T 或者 1，那么

$$\Sigma n, n' \cdot n' = n+1 \ = \ \infty$$

但是对 n 的任何固定值，都有一个 n' 的值使得 $n' = n + 1$ 的值为T或者1，于是

$$\Sigma n' \cdot n' = n+1 \ = \ 1$$

对 n 的任何固定值，$n' = n + 1$ 是 n' 的单点分布。类似地，任意一个可实现的确定的规范是终结状态的单点分布。

对编程记号总结如下，使得允许概率运算对象。

$$ok \qquad = \ (x'=x) \times (y'=y) \times \ldots$$

$x := e$ = $(x'=e) \times (y'=y) \times \ldots$

if b **then** P **else** Q **fi** = $b \times P + (1-b) \times Q$

$P.Q$ = $\Sigma x'', y'', \ldots \cdot$ (对于P内的$x', y', \ldots$ 替换成$x'', y'', \ldots$)

$\times$ (对于Q内的$x, y, \ldots$ 替换成$x'', y'', \ldots$)

因为$\perp=0$ 和 $\top=1$，ok 的定义和赋值保持不变；它们只是以算术上的形式表达出来。如果 b，P 和 Q 是二元的，那么 **if** b **then** P **else** Q **fi** 和 $P.Q$ 的定义保持不变。但现在它们不只应用于$\perp$和 $\top$，也就是 0 和 1，而且还应用于 0 和 1 之间的值。换句话说，它们应用于概率。若 b 是一个初始状态的概率，P 和 Q 是终结状态的分布，那么 **if** b **then** P **else** Q **fi 是终极状态的一个分布。若** P 和 Q 是终结状态的分布，那么 $P.Q$ **是终极状态的一个分布。例如，**

if 1/3 **then** $x := 0$ **else** $x := 1$ **fi**

意味着在概率为 1/3 的情况下，把 x 赋值为 0，在剩下的概率为 2/3 的情况下，把 x 赋值为 1 。对于变量 x，

if 1/3 **then** $x := 0$ **else** $x := 1$ **fi**

= $1/3 \times (x'=0) + (1 - 1/3) \times (x'=1)$

用0作为x'的值，计算这个表达式。

$1/3 \times (0=0) + (1-1/3) \times (0=1)$

= $1/3 \times 1 + 2/3 \times 0$

= $1/3$

这个结果就是x最终值为0的概率。用1作为x'的值，计算这个表达式。

$1/3 \times (1=0) + (1-1/3) \times (1=1)$

= $1/3 \times 0 + 2/3 \times 1$

= $2/3$

这个结果就是x最终值为1的概率。用2作为x'的值，计算这个表达式。

$1/3 \times (2=0) + (1-1/3) \times (2=1)$

= $1/3 \times 0 + 2/3 \times 0$

= 0

这个结果就是x最终值为2的概率。

这里有一个稍微更详细说明的有关变量x的例子。

if 1/3 **then** x:= 0 **else** x:= 1 **fi**.

if x=0 **then if** 1/2 **then** x:= x+2 **else** x:= x+3 **fi**

else if 1/4 **then** x:= x+4 **else** x:= x+5 **fi fi**

$= \quad \Sigma x'' \cdot \ ((x''=0)/3 + (x''=1)\cdot 2/3)$

$\times \ ((x''=0) \times ((x' = x''+2)/2 + (x' = x''+3)/2)$

$+ (1 - (x''=0)) \times ((x' = x''+4)/4 + (x' = x''+5) \times 3/4))$

$= \quad (x'=2)/6 + (x'=3)/6 + (x'=5)/6 + (x'=6)/2$

第一行之后，x可能为0或1 。如果它是0，有1/2的概率加2，有剩下1/2的概率加3；否则（如果x不为0）有1/4的概率加4，有剩下3/4的概率加5 。所得总和比它看起来更简单，因为所有除了0和1之外的x''的值再总和中都得出了0 。最后一行说明变量x的最终值为2的概率是1/6，是3的概率是1/6，是5的概率是1/6，是6的概率是1/2，任何其他值的概率是0 。

令P为最终状态的任意分布，令e为初始状态的任意数字表达。执行P之后，e的平均值为$(P.e)$ 。例如，根据分布2^{-n}, 当n在$nat+1$上变化时，n^2的平均值为

$2^{-n'} . n^2$

$= \Sigma n'' : nat+1 \cdot 2^{-n''} \times n''^2$

$=6$

执行前面的例子之后，x的平均值为

if 1/3 **then** x:= 0 **else** x:= 1 **fi**.

if x=0 **then if** 1/2 **then** x:= x+2 **else** x:= x+3 **fi**

else if 1/4 **then** x:= x+4 **else** x:= x+5 **fi fi**.

x

$= \quad (x'=2)/6 + (x'=3)/6 + (x'=5)/6 + (x'=6)/2 . x$

$= \quad \Sigma x'' \cdot \ ((x''=2)/6 + (x''=3)/6 + (x''=5)/6 + (x''=6)/2 \times x''$

$= \quad 1/6 \times 2 + 1/6 \times 3 + 1/6 \times 5 + 1/2 \times 6$

$= \quad 4 + 2/3$

令P为最终状态的任意分布，令b为初始状态的任意二元表达。执行P之后，b为真的概率为$(P.b)$ 。概率只是一个二元表达式的平均值。例如，前面的例子执行之后，x大于3的概率是

if 1/3 **then** x:= 0 **else** x:= 1 **fi**.

if x=0 **then if** 1/2 **then** x:= x+2 **else** x:= x+3 **fi**

else if 1/4 **then** x:= x+4 **else** x:= x+5 **fi fi**.

x>3

= (x'=2)/6 + (x'=3)/6 + (x'=5)/6 + (x'=6)/2. x>3

= $\Sigma x''\cdot$ ((x''=2)/6 + (x''=3)/6 + (x''=5)/6 + (x''=6)/2 × (x''>3)

= 1/6 × (2>3) + 1/6 × (3>3)+ 1/6 × (5>3) +1/2 × (6>3)

= 2/3

大部分的定律，包括所有的分布定律和置换定律，应用的时候不改变概率的规范和程序。例如，前面两个计算可以这样开始，先把最后一行（第一例中的x，第二例中的x>3)分配回到**then**-和**else**-的增长x的部分，然后把**if** x=0 …分配回到**then**-和**else**-的赋予x初始值的部分，再用六次置换定律，避免求和的需要。

5.7.0 随机数产生器

许多程序设计语言都提供了随机数产生器(Random Number Generators)（有时称为“伪随机数产生器”）。通常的表示是函数式的，通常的结果是一个值，其分布是在一个非空有限范围上的常量。若 $n : nat+1$，使用 $rand\ \ n$ 表示平均分布在范围 $0,..n$ 上产生自然数的产生器。因此 $rand\ \ n$ 以 $(r : 0,..n)/n$ 的概率取值 r 。

随机数产生器的函数表示是不一致的。因为 $x = x$ 是定律，按理来说可以将 *rand n*=*rand n* 简化成 T，但事实不能，因为 *rand n* 的两次出现可能产生不同的数。因为 $x + x = 2 \times x$ 是定律，按理来说能将 *rand n* + *rand n* 简化成 $2\times$*rand n*，但事实也不能。为了恢复一致性，在做任何操作之前可以将 *rand n* 的每次出现用一个新的变量代替。可以将 *rand n* 用整数变量 r 代替，它的取值的概率是 $(r : 0,..n)/n$ 。或者，如果你喜欢，也可以将 *rand n* 用一个取值概率为 1/n 的变量 $r : 0,..n$ 代替。（这是一个数学变量，换句话说是状态常量，没有 r'。）例如，使用一个状态变量 x，

x := *rand* 2. x := x + *rand* 3　　用 r 和 s 分别置换两个 *rand*

= Σr: 0,..2· Σs: 0,..3· (x:= r)/2. (x:= x + s)/3　　因式分解两次

= (Σr: 0,..2· Σs: 0,..3· (x:= r. x:= x + s))/6　　替换最后的赋值，置换定律

= (Σr: 0,..2· Σs: 0,..3· (x':= r + s)) / 6　　求和

= $((x'=0+0)+(x'=0+1)+(x'=0+2)+(x'=1+0)+(x'=1+1)+(x'=1+2))/6$

= $(x'=0)/6+(x'=1)/3+(x'=2)/3+(x'=3)/6$

表示 x' 以 1/6 的概率为 0，以 1/3 的概率为 1，以 1/3 的概率为 2，以 1/6 的概率为 3，任何其他值的概率都为 0。

当 *rand* 出现在一个简单等式的上下文中时，例如 *r* = *rand n*，不需要为它引入变量，因为已经提供了一个。只要将这个具有欺骗性的等式置换成 $(r:0,..n)/n$。例如，使用变量 x，和之前一样，有

x:= *rand* 2. *x* := *x* + *rand* 3　　置换赋值式

= $(x':0,..2)/2.(x':x+(0,..3))/3$　　相关组合

= $\sum x''\cdot(x'':0,..2)/2\times(x':x''+(0,..3))/3$　　求和

= $1/2\times(x':0,..3)/3+1/2\times(x':1,..4)/3$

= $(x'=0)/6+(x'=1)/3+(x'=2)/3+(x'=3)/6$

并且 **if** *rand* 2 **then** *A* **else** *B* **fi** 可被替换成 **if** 1/2 **then** *A* **else** *B* **fi**。

尽管 *rand* 在自然数上产生平均分布的数值，它可被转换成各种不同的分布。通过前面可以知道 $rand2+rand3$ 以分布 $(n=0\vee n=3)/6+(n=1\vee n=2)/3$ 取值 *n*。作为另一个例子，$rand8<3$ 以下述分布取二元值 *b*：

$\sum r:0,..8\cdot(b=(r<3))/8$

= $(b=\mathsf{T})\times 3/8+(b=\bot)\times 5/8$

= $5/8-b/4$

表示 b 以 3/8 的概率取 T，以 5/8 的概率取 ⊥。

练习 232 是纸牌游戏的简化版本。现在假设有台面上的一张纸牌，它的值是 1 至 13 的其中之一。可以只要这一张牌，也可以要第二张。目标是要总点数最接近 14，但不能超过 14。策略是当第一张纸牌的值小于 7 时要第二张。假设每张纸牌以相等的概率取值（事实上，第二张纸牌取和第一张纸牌相等的值的概率会变小，但避免复杂性可以忽略这一点），用 (*rand* 13)+1 表示一张纸牌。使用一个变量 x，游戏为：

x:= (*rand* 13) + 1. **if** *x*<7 **then** *x*:= *x* + (*rand* 13) + 1 **else** *ok* **fi**

替换 *rand* 和 *ok*

= $(x':(0,..13)+1)/13.$ **if** *x*<7 **then** $(x':x+(0,..13)+1)/13$ **else** $x'=x$ **fi**　　替换. 和 **if**

= $\sum x''\cdot(x'':1,..14)/13\times((x''<7)\times(x':x''+1,..x''+14)/13+(x''\geq 7)\times(x'=x''))$

几个省略的步骤

= $((2\leq x'<7)\times(x'-1)+(7\leq x'<14)\times 19+(14\leq x'<20)\times(20-x'))/169$

这是使用“小于 7”策略情况下 x' 的分布。类似地，也可以得到“小于 8”策略或任何别的策略下 x' 的分布。但是哪个策略才是最好的？如果要比较两种策略，可以一次同时采用。使用完全相同的纸牌 c 和 d，游戏者 x 采用“小于 n”策略，游戏者 y 采用“小于 $n+1$”策略（如果使用不同的纸牌，结果也是一样，但是需要更多变量）。下面是新的游戏，跟随的是 x 赢的条件：

$c:=(rand13)+1.d:=(rand13)+1.$

if $c<n$ **then** $x:=c+d$ **else** *x:*= *c* **fi**. **if** $c<n+1$ **then** $y:=c+d$ **else** *y:*= *c* **fi**.

$y<x\leq 14\vee x\leq 14<y$　　替换 *rand* 并使用函数−命令定律两次

= $(c':(0,..13)+1\wedge d':(0,..13)+1\wedge x'=x\wedge y'=y)/13/13.$

$x:=$ **if** $c<n$ **then** $c+d$ **else** *c* **fi**. $y:=$ **if** $c<n+1$ **then** $c+d$ **else** *c* **fi**.

$y < x \le 14 \vee x \le 14 < y$ 使用置换定律两次

$=$ $(c': (0,..13)+1 \wedge d': (0,..13)+1 \wedge x' = x \wedge y' = y)/169.$

if $c < n+1$ **then** $c+d$ **else** c **fi** $<$ **if** $c < n$ **then** $c+d$ **else** c **fi** ≤ 14

$\vee$ **if** $c < n$ **then** $c+d$ **else** c **fi** $\le 14 <$ **if** $c < n+1$ **then** $c+d$ **else** c **fi**

$=$ $(c': (0,..13)+1 \wedge d': (0,..13)+1 \wedge x' = x \wedge y' = y)/169.\ c = n \wedge d > 14-n$

$=$ $\sum c'', d'', x'', y'' \cdot$

$(c'': (0,..13)+1 \wedge d'': (0,..13)+1 \wedge x'' = x \wedge y'' = y)/169 \times (c'' = n \wedge d'' > 14-n)$

$=$ $\sum d'': 1,..14 \cdot (d'' > 14-n)/169$

$=$ $(n-1)/169$

x 赢的概率是 $(n-1)/169$。通过类似的演算可以得到 y 赢的概率是 $(14-n)/169$，平局的概率是 12/13。对于 $n < 8$，“小于 $n+1$”比“小于 n”要好。对于 $n \ge 8$，“小于 n”比“小于 $n+1$”好。因此“小于 8”比“小于 7”和“小于 9”都好。

练习 327 问：如果不断掷两个六面的色子，直到它们相等，需要多长的时间？程序是：

$u'=v' \Leftarrow u := (rand\ 6) + 1.\ v := (rand\ 6) + 1.$ **if** $u=v$ **then** *ok* **else** $t := t+1.\ u'=v'$ **fi**

每步迭代以 5/6 的概率要继续，以 1/6 的概率停止。因此提供一个假设，（对有限时间 t）执行时间有分布：

$$(t' \ge t) \times (5/6)^{t'-t} \times 1/6$$

为了证明它，可以从实现开始：

$u := (rand\ 6)+1. v := (rand\ 6)+1.$ 代替 *rand* 及

if $u = v$ **then** $t' = t$ **else** $t := t+1.\ (t' \ge t) \times (5/6)^{t'-t} \times 1/6$ **fi** 置换定律

$=$ $((u': 1,..7) \wedge v' = v \wedge t' = t)/6.(u' = u \wedge (v': 1,..7) \wedge t' = t)/6.$ 先替换.

if $u = v$ **then** $t' = t$ **else** $(t' \ge t+1) \times (5/6)^{t'-t-1}/6$ **fi** 并简化

$=$ $(u', v': 1,..7 \wedge t'=t)/36.$ 替换剩余的.

if $u = v$ **then** $t' = t$ **else** $(t' \ge t+1) \times (5/6)^{t'-t-1}/6$ **fi** 并替换**if**

$=$ $\sum u'', v'': 1,..7 \cdot \sum t'' \cdot (t'' = t)/36 \times ((u'' = v'') \times (t' = t'')$

$+ (u'' \ne v'') \times (t' \ge t''+1) \times (5/6)^{t'-t''-1}/6)$ 求和

$=$ $(6 \times (t'=t) + 30 \times (t' \ge t+1) \times (5/6)^{t'-t-1}/6) / 36$ 组合

$=$ $(t' \ge t) \times (5/6)^{t'-t} \times 1/6$

得到的是我们假设的分布，完成证明。

t' 的平均值是：

$$(t' \ge t) \times (5/6)^{t'-t} \times 1/6.\ t = t+5$$

因此平均而言要得到相等的一对结果还需再投掷 5 次（第一次之后）。

----------随机数产生器结束

概率问题由于容易给人误导而臭名昭著，甚至对职业的数学家也是如此。在标准的概率研究中，为达到一个概率分布所进行的非形式推理对形成合理的假设是很重要的。但假

设有时是错误的。可以将假设写成概率规范，将它精化为程序，并如同二元规范一样去证明它。有时错误的假设会导致对问题的错误理解。形式化可以使理解清晰。通过证明可以说明一个程序的假设的概率分布是正确的。非形式化的争论被形式化证明所取代。

概率规范有时被解释为“模糊”(fuzzy)规范。例如$(x'=0)/1+(x'=1)\times 2/3$可以表示若 *x'* 为 0 则满意程度是 1/3，若为 1 则满意程度为 2/3，若是别的则完全不满意。

5.7.1 信息　　可选节

在信息和概率之间存在紧密的联系。若一个布尔表达式为真的概率是 p，计算它，得出它的确为真，那么刚才所得到的信息（information）的量以二进制位计为 *info p*，定义为

info p = - *log p*

其中 *log* 是 2 的对数。例如，*even (rand 8)*有 1/2 的概率为真。如果计算它发现它为真，那么得到信息的量为 1 位，如下：

info (1/2) = - *log* (1/2) = *log* 2 = 1

我们知道给定的随机数的最右位为 0。若发现 *even (rand 8)*为假，那么 ¬ *even (rand 8)*为真，而且因为它的概率也是 1/2，因此也得到 1 位信息。于是知道了该随机数的最右位为 1。若测试 *rand* 8 = 5，它为真的概率是 1/8，如果发现它确为真，于是就知道了 3 位信息，因为：

info (1/8) = - *log* (1/8) = *log* 8 = 3

这 3 位信息是整个随机数的二进制表达。如果发现 *rand* 8 = 5 为假，于是知道

info (7/8) = - *log* (7/8) = *log* 8 – *log* 7 = 3 – 2.80736 = 大约 0.19264

位信息。于是知道该随机数不是 5，但可能是任何其他 7 个数。假设测试 *rand* 8 < 8。因为它肯定为真，所以其实没必要测试，于是知道：

info 1 = - *log* 1 = - 0 = 0

位信息。

当循环中出现 **if** *b* **then** *P* **else** *Q* **fi**，*b* 需要被重复测试。假设 *b* 为真的概率是 *p*。当它确为真时，可以知道 *info p* 位信息，它发生的概率是 *p*。当它为假时，可以知道 *info* (1-*p*) 位信息，而它发生的概率是(1-*p*)。所获得的平均信息，称为熵(entropy)，为：

entro p = *p* × *info p* + (1-*p*) × *info* (1-*p*)

例如，*entro*(1/2)=1，且 *entro*(1/8)=*entro*(7/8)=大约 0.54356。由于在 *p*=1/2 时 *entro p* 获得最大值，于是知道若测试的概率约为 1/2 时，平均情况下，测试的效率最高。例如第 4 章的二分搜索问题，可以对剩余的搜素表做任何划分，但为了得到最好的平均执行时间，可以将表按照想搜索的项进行等概率的划分为二。并且在快速求幂的问题中，如果我们可以选择，最好测 *even y* 的平均值，而不是 *y* =0 的平均值。

----------信息结束

----------概率程序设计结束

5.8 函数式程序设计　　可选节

本书大部分讨论的是一类称为“命令式”(imperative)的程序设计, 它表示程序描述了状态的一个变化(或“命令”计算机以某种方式改变状态)。本节介绍另外一种程序设计：程序是其输入到输出的一个函数。更普遍地, 规范是它的可能输入到期望输出的一个函数, 而

程序(一如既往)是实现了的规范。现在从已介绍的程序设计记号中去掉赋值和相关组合, 并加入函数。

为了进一步说明, 再次看一下列表求和问题(练习 169)。这次, 规范为$\langle L : [* \; rat] \rightarrow \sum L \rangle$。假定 $\sum$ 不是一个已实现了的操作符, 则还有一些程序设计要做。引入变量 n 来指明表中有多少项已经累加过了，初始化 n 为 0。

$$\sum L = \langle n : 0,..\#L+1 \rightarrow \sum L[n;..\#L] \rangle 0$$

这里将“$\langle L : [* \; rat] \rightarrow \sum L \rangle$ = ...” 略写成 “$\sum L$ = ... ”, 不过必须记住 L 的定义域。第一眼看上去, n 的定义域很令人讨厌, 看上去似乎采用包括两个边界点的区间记号要好一些。不过再看第二眼, 就可发现一些有用的东西：n 的定义域实际上由两个必须分别对待的部分组成。

$$0,..\#L+1 = (0,..\#L), \#L$$

将该函数分成两个函数的选择合并：

$$\langle n : 0,..\#L+1 \rightarrow \sum L \; [n;..\#L] \rangle = \langle n : 0,..\#L \rightarrow \sum L \; [n;..\#L] \rangle \mid \langle n : \#L \rightarrow \sum L \; [n;..\#L] \rangle$$

并分别处理每个部分。在左边部分, 有 $n < \#L$ ； 在右边部分有 $n = \#L$ 。

$$\langle n : 0,..\#L \rightarrow \sum L[n;..\#L] \rangle = \langle n : 0,..\#L \rightarrow L \; n + \sum L[n+1;..\#L] \rangle$$

$$\langle n : \#L \rightarrow \sum L[n;..\#L] \rangle = \langle n : \#L \rightarrow 0 \rangle$$

这一次复制 n 的定义域来表明选择合并式的哪一部分将被考虑, 而剩余的问题可由递归方式解决。

$$\sum L[n+1;..\#L] = \langle n : 0,..\#L+1 \rightarrow \sum L[n;..\#L] \rangle (n+1)$$

也可以使用 **if then else fi** 结构来代替选择合并式, 它们的关系由下面的定律表示：

$$\langle v: A \rightarrow x \rangle \mid \langle v: B \rightarrow y \rangle = \langle v: A \; , B \rightarrow \textbf{if } v: A \textbf{ then } x \textbf{ else } y \textbf{ fi} \rangle$$

如果兴趣在于执行时间而不是结果, 可以将每个函数的结果根据某种度量用执行时间代替。例如, 在列表求和问题中, 可以定义每一次加法运算耗时为 1, 而其他运算耗时为 0。这样, 原规范变成 $\langle L : [*rat] \rightarrow \#L \rangle$, 它表示对任何一个表, 表求和的执行时间是它的长度。现在必须按照与以前一样的程序设计步骤执行。第一步是引入变量 n ； 现在引入变量 n, 但为这个新函数选用一个新函数结果来表示它的执行时间：

$$\#L = \langle n: 0,..\#L+1 \rightarrow \#L - n \rangle 0$$

第二步应将此函数分解成一个选择合并式，这里这样继续照此执行：

$$\langle n: 0,..\#L+1 \rightarrow \#L - n \rangle = \langle n: 0,..\#L \rightarrow \#L - n \rangle \mid \langle n: \#L \rightarrow \#L - n \rangle$$

合并式的左边变成带有一个加法运算的函数, 所以计时函数必须变成带有耗时为 1 的函数。为使等式正确, 必须调整剩余的求和时间：

$$\langle n: 0,..\#L \rightarrow \#L - n \rangle = \langle n: 0,..\#L \rightarrow 1 + \#L - n - 1 \rangle$$

合并式的右边变成一个带有常量结果的函数；根据所用的计时度量, 它的时间必须为 0。

$$\langle n: \#L \rightarrow \#L - n \rangle = \langle n: \#L \rightarrow 0 \rangle$$

剩余问题可通过递归调用解决；下面就利用相应的递归调用求解剩余的时间问题。

$$\#L - n - 1 = \langle n: 0,..\#L+1 \rightarrow \#L - n \rangle (n+1)$$

至此完成了执行时间(根据上述度量)为列表长度的证明。

在递归时间度量中, 只有递归调用才耗费时间, 定义 1 为一次递归调用的耗时。现在用这个度量重新完成计时证明。 同样, 时间的规范仍然是 $\langle L: [*rat] \rightarrow \#L \rangle$。

$$\#L \;=\; \langle n\colon 0,..\#L{+}1 \longrightarrow \#L - n\rangle\, 0$$

$$\langle n\colon 0,..\#L{+}1 \longrightarrow \#L - n\rangle \;=\; \langle n\colon 0,..\#L \longrightarrow \#L - n\rangle \mid \langle n\colon \#L \longrightarrow \#L - n\rangle$$

$$\langle n\colon 0,..\#L \longrightarrow \#L - n\rangle \;=\; \langle n\colon 0,..\#L \longrightarrow \#L - n\rangle$$

$$\langle n\colon \#L \longrightarrow \#L - n\rangle \;=\; \langle n\colon \#L \longrightarrow 0\rangle$$

$$\#L - n \;=\; 1{+}\,\langle n\colon 0,..\#L{+}1 \longrightarrow \#L - n\rangle\,(n{+}1)$$

5.8.0 函数精化

在命令式程序设计中, 可以写一个非确定的规范，例如 x' : 2 , 3 , 4 , 它允许最终结果是几种可能中的任意一个。在函数式程序设计中, 一个非确定的规范是由一个以上的元素组成的束。规范 2 , 3 , 4 允许结果可以是这三个数值中的任何一个。

函数式规范可以用与命令式规范一样的方式分类, 基于对每一个输入有多少满足的输出进行分类。

- 函数式规范 S 对定义域中的元素 x 是不可满足的, 如果： $\quad \mathcal{C}\,Sx < 1$
- 函数式规范 S 对定义域中的元素 x 是可满足的, 如果： $\quad \mathcal{C}\,Sx \geq 1$
- 函数式规范 S 对定义域中的元素 x 是确定的, 如果： $\quad \mathcal{C}\,Sx \leq 1$
- 函数式规范 S 对定义域中的元素 x 是非确定的, 如果： $\quad \mathcal{C}\,Sx > 1$

- 函数式规范 S 对定义域中的元素 x 是可满足的, 如果： $\quad \exists\, y\cdot\, y : Sx$
- 函数式规范 S 是可实现的, 如果： $\quad \forall\, x\cdot\, \exists\, y\cdot\, y : Sx$

(x 限定在 S 的定义域内, y 限定在 S 的值域内。)可实现性可以重新表述成 $\forall\, x\cdot\, Sx \neq null$ 。

下面讨论在一个整数表中寻找某个特定项的问题。首先想出的规范可能是

$$\langle L\colon [*int] \rightarrow \langle x\colon int\cdot\ \S\, n\colon 0,..\#L\cdot\, Ln = x\rangle\rangle$$

它表示对任意表 L 和项 x, 求 L 中 x 出现处的索引。如果 x 在 L 中出现了多次, 任意一次索引都可作为解。不幸的是, 如果 x 不出现在 L 中, 就无法给出任何可能的结果, 所以该规范是不可实现的。我们必须决定当 x 不出现在 L 中时希望的是怎样的结果；可以定义任意一个不为 L 索引的自然数都符合要求, 这时规范应为：

$$\langle L\colon [*int] \rightarrow \langle x\colon int\cdot\ \textbf{if}\ x\colon L(0,..\#L)\ \textbf{then}\ \S\, n\colon 0,..\#L\cdot\, Ln = x\ \ \textbf{else}\ \#L,..\infty\ \textbf{fi}\rangle\rangle$$

该规范是可实现的, 且经常是非确定的。

函数式精化式与命令式精化式类似。命令式规范是一个二元表达式，而命令式精华是其反向含义。函数式规范是一个函数，而函数式精华把函数的排序反过来。函数式规范 P(问题)被函数式规范 S(解)精化, 当且仅当 S: P。为了精化，可以减少结果的选择可能性，或者扩大定义域。现在出现了一个非常令人讨厌的记号问题。一般的习惯是将问题写在左边, 接着是精化式符号, 然后是解写在右边； 现在却要写成另一种形式 S: P。包含式是不对称的, 所以它的符号也不应该对称, 但不幸的是它的符号恰是对称的。为解决这一问题，可以用 :: 表示“反向冒号”, 则“P 被 S 精化”就可以写成 P:: S。

为了精化上面的查找规范, 可以创建一个线性查找程序：查找从索引 0 开始, 不断增加索引值, 直到 x 被找到或者 L 被查找完。首先要引入索引。

if x: $L(0,..\#L)$ **then** § n: $0,..\#L \cdot Ln = x$ **else** $\#L,.. \infty$ **fi**::

$\langle i$: $nat \rightarrow$ **if** x: $L\ (i,..\#L)$ **then** § n: $i,..\#L \cdot L\ n = x$ **else** $\#L\ ,..\infty$ **fi**$\rangle\ 0$

这个精化式的两边是相等的, 所以可用 = 代替 :: 。其实可以使 i 的定义域更为精确，然后再将函数分解成一个选择性合并，如同在前面的问题中所做的一样。但这一次采用 **if then else fi** 结构。

if x: $L\ (i,..\#L)$ **then** § n: $i,..\#L \cdot Ln = x$ **else** $\#L,.. \infty$ **fi**::

if $i = \#L$ **then** $\#L$

else if $x = Li$ **then** i

else $\langle i$: $nat \rightarrow$ **if** x: $L\ (i,..\#L)$ **then** § n: $i,..\#L \cdot Ln = x$ **else** $\#L,..\infty$ **fi**$\rangle\ (i+1)$ **fi fi**

利用递归时间度量, 计时规范为$\langle L \rightarrow \langle x \rightarrow 0,..\#L+1\rangle\rangle$, 这意味着时间少于$\#L+1$。为了证明这是执行时间，必须证明：

$0,..\#L+1$:: $\langle i$: $nat \rightarrow 0,..\#L-i+1\rangle\ 0$

以及

$0,..\#L-i+1$:: **if** $i = \#L$ **then** 0

else if $x = Li$ **then** 0

else $1+ \langle i$: $nat \rightarrow 0,..\#L-i+1\rangle\ (i+1)$ **fi fi**

通过此例可以看出，对同一个问题，函数式精化和命令式精化的步骤是一样的，包括非确定的解和时间。但标记方法有所不同。

----------函数精化结束

函数式和命令式程序设计其实并不是互不相容的, 它们可以合在一起使用。如果有时希望暂停，中断计算一会儿, 然后再从相同状态继续执行, 就无法忽视命令式程序设计。命令式程序设计语言都包含一种函数式 (表达式) 子语言, 所以同样也不能忽视函数式程序设计。

函数式程序设计的核心是应用公理：

$\langle v$: $D \rightarrow b\rangle\ x$ = (将 b 中的 v 置换为 x)

命令式程序设计的核心是置换定律：

x:= $e.P$ = (将 P 中的 x 置换为 e)

函数式和命令式程序设计主要在置换所运用的标记方法上有所不同。

----------函数式程序设计结束

----------程序设计语言结束

第六章 递归定义

6.0 递归数据定义

本节将讨论无限束 (infinite bunches) 的定义。首先要讨论的例子是 *nat*, 自然数。它在第二章已使用构造(construction)和归纳(induction)公理进行了定义，这里将进一步讨论这些公理。

6.0.0 构造和归纳

要定义 *nat*，需要说明它的元素是什么。首先定义 0 是它的一个元素：

$$0: nat$$

然后定义对于 *nat* 中的每一个元素, 将其加上 1 就是 *nat* 的另一个元素：

$$nat+1: nat$$

这些公理称为 *nat* 的构造公理, 0 和 *nat*+1 称为 *nat* 的构造式(constructors)。利用这些公理, 可以“构造” *nat* 的元素如下：

	T	根据公理, $0: nat$
$\Rightarrow$	$0: nat$	两边加 1
$\Rightarrow$	$0+1: nat+1$	根据算术运算, $0+1 = 1$；根据公理 $nat+1: nat$
$\Rightarrow$	$1: nat$	两边加 1
$\Rightarrow$	$1+1: nat+1$	根据算术运算, $1+1 = 2$；根据公理 $nat+1: nat$
$\Rightarrow$	$2: nat$	

依次下去。

利用上述构造公理可以证明 $2: nat$，但不能证明 $\neg -2: nat$, 因此还需归纳公理。构造公理说明自然数属于 *nat*, 而归纳公理说明再没有别的数属于 *nat*。下面是 *nat* 的归纳公理。

$$0: B \wedge B+1: B \;\Rightarrow\; nat: B$$

这里将 *nat* 作为一个常量, 就象 *null* 和 0 一样, 它不是一个变量, 并且不能被实例化。不过 *B* 是一个变量, 可以按意愿进行实例化。

上面两个构造公理可以合并成一个, 并且归纳公理可以重写成如下：

$0, nat+1: nat$ *nat* 构造公理

$0, B+1: B \;\Rightarrow\; nat: B$ *nat* 归纳公理

有许多束满足蕴含式 $0, B+1: B,$ 例如：自然数, 整数, 整数与半整数(half-integers), 有理数。归纳公理指出在所有这些束中, *nat* 是最小的。

前面已经介绍了使用束记号的 *nat* 构造公理以及 *nat* 归纳公理。现在介绍采用谓词记号的等价公理。从归纳开始。

在谓词记号中, *nat* 归纳公理可写成 ： 如果 $P: nat \to bin,$

$$P0 \wedge \forall\, n: nat \cdot Pn \;\Rightarrow P(n+1) \;\Longrightarrow\; \forall\, n: nat \cdot Pn$$

首先证明束形式隐含谓词形式：

$0{:}\ B \wedge B{+}1{:}\ B \ \Rightarrow nat{:}\ B$　　令 $B = \S n{:}\ nat \cdot Pn$, 那么 $B{:}\ nat$,

$\Rightarrow\ 0{:}\ B \wedge (\ \forall\ n{:}\ nat\cdot\ n{:}\ B\ \Rightarrow n{+}1{:}\ B)\ \Rightarrow \forall\ n{:}\ nat\cdot n{:}\ B$　　并且有 $\forall\ n{:}\ nat\cdot\ (n{:}\ B) = Pn$ 。

$=\ \ P0 \wedge (\ \forall\ n{:}\ nat\cdot\ Pn\ \Rightarrow P(n{+}1)\ )\ \Rightarrow\ \forall\ n{:}\ nat\cdot\ Pn$

反向证明类似。

$P0 \wedge (\ \forall\ n{:}\ nat\cdot\ Pn\ \Rightarrow P(n{+}1)\ )\ \Rightarrow\ \forall\ n{:}\ nat\cdot\ Pn$

对于任意束 B, 令 $P = \langle n : nat \rightarrow n{:}\ B\rangle$, 那么再次有 $\forall\ n{:}\ nat\cdot\ Pn = (n{:}\ B)$。

$\Rightarrow\ 0{:}\ B \wedge (\ \forall\ n{:}\ nat\cdot\ n{:}\ B \Rightarrow n{+}1{:}\ B)\ \Rightarrow\ \forall\ n{:}\ nat\cdot\ n{:}\ B$

$=\ \ 0{:}\ B \wedge (\ \forall\ n{:}\ nat\ `B\cdot\ n{+}1{:}\ B\ )\ \Rightarrow\ \forall\ n{:}\ nat\cdot\ n{:}\ B$

$=\ \ 0{:}\ B \wedge (nat\ `B){+}1{:}\ B\ \Rightarrow nat{:}\ B$

$\Rightarrow\ 0{:}\ B \wedge B{+}1{:}\ B\ \Rightarrow\ nat{:}\ B$

因此, *nat* 归纳公理的束形式和谓词形式是等价的。

nat 构造公理的谓词形式可表示如下：如果 $P{:}\ nat\ \rightarrow\ bin$,

$$P0 \wedge\ \forall\ n{:}\ nat\cdot\ Pn\ \Rightarrow P(n{+}1) \Leftarrow \forall\ n{:}\ nat\cdot\ Pn$$

它与归纳公理相同, 只不过主蕴含式是反向的。先证明束形式隐含谓词形式。

$\forall n{:}\ nat\cdot\ Pn$　　运用 *nat* 构造公理改变定义域, 以束的形式

$\Rightarrow\ \forall n{:}\ 0\ ,\ nat{+}1\cdot\ Pn$　　关于 $\forall$ 的公理

$=\ \ (\ \forall n{:}\ 0\cdot\ Pn) \wedge (\ \forall n{:}\ nat{+}1\cdot\ Pn)$　　单点定律和变量改变

$=\ \ P0 \wedge \forall\ n{:}\ nat\cdot\ P(n{+}1)$

$\Rightarrow\ \ P0 \wedge \forall n{:}\ nat\cdot\ Pn\ \Rightarrow P(n{+}1)$

现在再证明谓词形式隐含束形式。

$P0 \wedge (\forall n{:}\ nat\cdot\ Pn\ \Rightarrow P(n{+}1)\ ) \Leftarrow \forall\ n{:}\ nat\cdot\ Pn$　　令 $P = \langle n{:}\ nat \rightarrow n{:}\ nat\rangle$

$\Rightarrow\ 0{:}\ nat \wedge (\forall n{:}\ nat\cdot\ n{:}\ nat \Rightarrow n{+}1{:}\ nat\ )\ \Leftarrow\ \forall n{:}\ nat\cdot\ n{:}\ nat$

$=\ \ 0{:}\ nat \wedge (\forall n{:}\ nat\cdot\ n{+}1{:}\ nat\ )\ \Leftarrow \mathsf{T}$

$=\ \ 0{:}\ nat \wedge nat{+}1{:}\ nat$

因此有一个推论，*nat* 可由这条公理定义：

$$P0 \wedge \forall n{:}\ nat\cdot\ Pn\ \Rightarrow P(n{+}1)\ =\ \forall n{:}\ nat\cdot\ Pn$$

归纳公理还有其他的谓词形式, 下面是常用的一种加上另外三种。

$P0 \wedge \forall n{:}\ nat\cdot\ Pn \Rightarrow P(n{+}1)\ \Rightarrow\ \forall n{:}\ nat\cdot\ Pn$

$P0 \vee \exists n{:}\ nat\cdot\ \neg Pn \wedge P(n{+}1)\ \Leftarrow\ \exists n{:}\ nat\cdot\ Pn$

$\forall n{:}\ nat\cdot\ Pn \Rightarrow P(n{+}1)\ \Rightarrow\ \forall n{:}\ nat\cdot\ P0 \Rightarrow Pn$

$\exists n{:}\ nat\cdot\ \neg Pn \wedge P(n{+}1)\ \Leftarrow\ \exists n{:}\ nat\cdot\ \neg P0 \wedge Pn$

第一种形式表明，要证明对所有自然数都有 P 成立，先证明对于 0 有 P 成立, 然后假定对于自然数 n 有 P 成立, 再证明对于 $n{+}1$ 也有 P 成立。换句话说, 通过从 0 开始, 重复加 1, 就可以得到所有的自然数。第二种形式可通过对第一种形式运用对偶定律和换名得到。第三种形式是最漂亮的一个；它表明如果可以从任何自然数得到它下一个, 那么就可以从 0 得到任何自然数。

下面两个定律从归纳公理导出：

$\forall n\colon nat\cdot\ (\forall m : nat\cdot\ m<n \Rightarrow Pm) \Rightarrow Pn\ \Rightarrow \forall n\colon nat\cdot\ Pn$

$\exists n\colon nat\cdot\ (\forall m : nat\cdot\ m<n\ \Rightarrow\ \neg Pm) \wedge Pn\ \Leftarrow\ \exists n\colon nat\cdot\ Pn$

第一式类似上面归纳公理谓词形式的第一种形式，只是归纳的基础情况 $P0$ 没有明确写出，并且步骤中使用了如下假设，前面的所有自然数都满足 P，而不只是前面的一个自然数。后一个式子表明，如果存在一个自然数具有属性 P, 那么存在第一个自然数具有属性 P(所有前面的自然数都没有此属性)。

使用归纳公理的证明不需任何特殊的记号或格式。例如，练习 336 要求证明一个奇数的平方是对于某个自然数 m 的 $8\times m+1$。对 nat 进行量化，

$\forall n\cdot\exists m\cdot(2\times n+1)^2 = 8\times m+1$ 数的定律

$= \quad \forall n\cdot\exists m\cdot 4\times n\times(n+1)+1 = 8\times m+1$ 数的定律

$= \quad \forall n\cdot\exists m\cdot n\times(n+1) = 2\times m$ 归纳公理的谓词形式

$\Leftarrow \quad (\exists m\cdot\ 0\times(0+1) = 2\times m)$ 普遍化和

$\wedge\ (\forall n\cdot\ (\exists m\cdot\ n\times(n+1) = 2\times m) \Rightarrow (\exists l\cdot\ (n+1)\times(n+2) = 2\times l))$ 分配律

$\Leftarrow \quad 0\times(0+1) = 2\times 0$ 算术和

$\wedge\ (\forall m, n\cdot\ n\times(n+1) = 2\times m \Rightarrow (n+1)\times(n+2) = 2\times(m+n+1)$ 普遍化

$\Leftarrow\ \forall n, m\cdot n\times(n+1) = 2\times m \Rightarrow (n+1)\times(n+2) = 2\times(m+n+1)$ 数的定律

$= \quad \mathsf{T}$

现在有了一个无限束的定义, 很容易定义其他束了。例如, 可以定义 pow 为 2 的幂, 可运用等式：

$$pow = 2^{nat}$$

或者运用解量词：

$$pow = \S p\colon nat\cdot\ \exists\ m\colon nat\cdot\ p = 2^m$$

不过这里用定义 nat 的方法来定义它。pow 的构造公理为：

$$1,\ 2\times pow\colon pow$$

pow 的归纳公理为：

$$1,\ 2\times B\colon B \Rightarrow pow\colon B$$

归纳公理不只是为了定义 nat。用谓词形式, 可以用下面公理来定义 pow ：

$$P1 \wedge \forall p\colon pow\cdot\ Pp \Rightarrow P(2\times p)\ =\ \forall p\colon pow\cdot\ Pp$$

可以定义整数束为：

$$int\ =\ nat, -nat$$

或者等价地利用构造和归纳公理定义为：

$0,\ int+1,\ int-1\colon int$

$0,\ B+1,\ B-1\colon B\ \Rightarrow\ int\colon B$

或者利用公理定义为：

$$P0 \wedge (\forall i\colon int\cdot\ Pi \Rightarrow P(i+1)\ ) \wedge (\ \forall\ i\colon int\cdot\ Pi\ \Rightarrow P(i-1)\ )\ =\ \forall i\colon int\cdot\ Pi$$

无论选择哪一个作为公理, 另外的就都作为定理。

类似地可以定义有理数束为:

$$rat = int/(nat+1)$$

或者等价地用构造和归纳公理定义为：

1, $rat+rat$, $rat–rat$, $rat×rat$, $rat/(§r: rat· r \neq 0): rat$

1, $B+B$, $B–B$, $B×B$, $B/(§b: B· b \neq 0): B \Rightarrow rat: B$

或者用下面的公理(当然限定在 *rat* 内)定义为：

$$\begin{aligned} & P1 \\ \wedge\ & (\forall r, s· Pr \wedge Ps \Rightarrow P(r+s)) \\ \wedge\ & (\forall r, s· Pr \wedge Ps \Rightarrow P(r–s)) \\ \wedge\ & (\forall r, s· Pr \wedge Ps \Rightarrow P(r×s)) \\ \wedge\ & (\forall r, s· Pr \wedge Ps \wedge s\neq 0 \Rightarrow P(r/s)) \\ = \quad & \forall r· Pr \end{aligned}$$

从上述例子看出, 可以通过构造公理和归纳公理，利用任意数量的构造式来定义一个束。在结束本节之前, 这里用零个构造式定义了一个束。通常, 有一个构造式就有一个构造公理, 所以这里就没有构造公理了。如果没有构造式, 前件就变得无关紧要, 可以不出现, 只剩下归纳公理：

$$null: B$$

其中 *null* 是要定义的束。像往常一样, 归纳公理表明, 除了由构造公理得到的元素之外, 再没有另外的元素在所定义的束之中。这正是在第二章中定义 *null* 的方法。*null* 的归纳法的谓词形式为：

$$\forall x: null· P\ x$$

----------构造和归纳结束

6.0.1 最小不动点

前面已经利用一个构造公理和一个归纳公理定义了 *nat* ：

0, $nat+1: nat$　　*nat* 构造

0, $B+1: B \Rightarrow nat: B$　　*nat* 归纳

现在来证明两个看起来类似的定理：

$nat = 0, nat+1$　　*nat* 不动点构造

$B = 0, B+1 \Rightarrow nat: B$　　*nat* 不动点归纳

函数 *f* 的一个不动点(fixed-point)是其定义域中的一个元素 *x*, 满足 *f* 将 *x* 映射到它本身： $x = fx$。函数 *f* 的 最小不动点(least fixed-point)是这样的不动点 *x* 中最小的一个。不动点构造式具有如下形式：

name = (含有 *name* 的表达式)

它表明 *name* 是其右边表达式的一个不动点。不动点归纳说明：*name* 是满足不动点构造的最小束, 在这种意义下, 它就是构造式的最小不动点。

先证明 *nat* 不动点构造。它比 *nat* 构造要强, 所以证明也要使用到 *nat* 归纳, 就从它开始：

T　　*nat* 的归纳公理

= 0, *B*+1: *B* ⇒ *nat*: *B*　　将 *B* 用 0, *nat*+1 替换

⇒ 0, (0, *nat*+1) +1: 0, *nat*+1 ⇒ *nat*: 0, *nat*+1

将前件第一个“: ”号两边都去掉 “+1”和 “0”, 可得到增强的前件

⇒ 0, *nat*+1: *nat* ⇒ *nat*: 0, *nat*+1

前件是 *nat* 构造公理, 所以可以删去它, 并利用它再次增强后件

= *nat* = 0, *nat*+1

通过增强 *nat* 归纳的前件，就可以证明 *nat* 不动点归纳。

用类似的方法, 同样可以证明 *pow, int, rat* 都是它们各自构造式中的最小不动点。实际上, 可以定义 *nat* 和上述的每一个束作为它们各自构造式的最小不动点。经常用不动点构造公理，称为语法，来定义一个串束。例如,

exp = "*x*", *exp*; "+"; *exp*

在这个语境中, 合并通常用 | 表示, 而联接通常没有符号表示。其他公理, 例如 *exp* 是最小不动点，通常被非正式地说成：仅仅包括构造出来的元素。

----------最小不动点结束

6.0.2 递归数据构造

递归构造是从构造式中构造最小不动点的过程。它通常是可行的, 但并不总是。如果要寻找下式的最小解：

name = (包含 *name* 的表达式)

可有如下过程：

0. 构造一个束序列 $name_0$ $name_1$ $name_2$..., 开头为

$name_0 = null$

接着有

$name_{n+1}$ = (包含 $name_n$ 的表达式)

因此可以对任意自然数 *n* 构造一个束 $name_n$ 。

1. 下一步, 试图为 $name_n$ 寻找一个表达式, 该表达式应使用了 *n* 但不使用 *name*。

$name_n$ = (包含 *n* 但不包含 *name* 的表达式)

2. 现在将 *n* 用 ∞ 替换, 构成一个束 $name_\infty$:

$name_\infty$ = (既不包含 *n* 也不包含 *name* 的表达式)

3. 束 $name_\infty$ 通常是构造式的最小不动点, 但并不总是, 所以必须测试它。首先测试它是否为一个不动点：

$$name_\infty = (\text{包含 } name_\infty \text{的表达式})$$

4. 接着测试 $name_\infty$是否为最小不动点：

$$B = (\text{包含 } B \text{ 的表达式}) \implies name_\infty : B$$

以 *pow* 构造式为例来说明递归构造。 *pow* 的构造式为 1, 2×*pow*。

0. 构造序列：

$pow_0 = null$

$pow_1 = 1, 2\times pow_0$

$= 1, 2\times null$

$= 1, null$

$= 1$

$pow_2 = 1, 2\times pow_1$

$= 1, 2\times 1$

$= 1, 2$

$pow_3 = 1, 2\times pow_2$

$= 1, 2\times(1,2)$

$= 1, 2, 4$

第一个束 pow_0 说明不使用 *pow* 构造式可得到的束 *pow* 的所有元素。一般而言, pow_n 表示经过 *n* 次使用 *pow* 构造式所得到 *pow* 的所有元素。

1. 也许现在就可以猜测该序列的常规成员：

$$pow_n = 2^{0,..n}$$

可以用 *nat* 归纳来证明上式, 但其实不必要。因为证明将只会说明当 *n*: *nat* 时 pow_n 的情况, 而实际需要的是 pow_∞ 。另外，还将测试最终结果。

2. 既然有了 pow_n 表达式, 可以定义 pow_∞ 如下：

$pow_\infty = 2^{0,..\infty}$

$= 2^{nat}$

这样也就找到了 *pow* 构造式最小不动点的一个可能候选者。

3. 必须测试 pow_∞, 看它是否为一个不动点。

$2^{nat} = 1, 2\times 2^{nat}$

$= \quad 2^{nat} = 2^0, 2^1\times 2^{nat}$

$= \quad 2^{nat} = 2^0, 2^{1+nat}$

$= \quad 2^{nat} = 2^{0, 1+nat}$

$\Leftarrow \quad nat = 0, nat+1$ — *nat* 的不动点构造

$= \quad \mathsf{T}$

4. 必须测试 pow_∞ 是否为最小不动点。

$2^{nat}: B$

$=$ $\forall\, n: nat \cdot 2^n: B$ 利用 *nat* 归纳的谓词形式

$\Leftarrow$ $2^0: B \wedge \forall n: nat \cdot 2^n: B \Rightarrow 2^{n+1}: B$ 改变变量

$=$ $1: B \wedge \forall m: 2^{nat} \cdot m: B \Rightarrow 2\times m: B$ 扩大定义域

$\Leftarrow$ $1: B \wedge \forall\, m: nat \cdot m: B \Rightarrow 2\times m: B$ 定义域改变定律

$=$ $1: B \wedge \forall\, m: nat`B \cdot 2\times m: B$ 扩大定义域

$\Leftarrow$ $1: B \wedge \forall m: B \cdot 2\times m: B$

$=$ $1: B \wedge 2\times B: B$

$\Leftarrow$ $B = 1,\ 2\times B$

因为 2^{nat} 是 *pow* 构造式的最小不动点, 所以有结论 $pow = 2^{nat}$。

在第 0 步，从 $name_0 = null$ 开始，这通常是找到最小解的一个最好的开始束。但偶尔这个开始束也会失败，而某个其他开始束对给定的不动点等式会成功产生解。

在第 2 步，通过将 n 替换为 ∞，对构造式的不动点从 $name_n$ 得到一侯选式 $name_\infty$。该替换是容易实施的，且侯选式结果通常是令人满意的。但结果对于 $name_n$ 的表示方式是敏感的。对于两个对所有有限的 n 都相等的表达式 $name_n$，获得的 $name_\infty$ 可能是不相等的。另一个对于 $name_n$ 的表示方式不敏感的例子是：

$$\S\, x \cdot LIMn \cdot x : name_n$$

但该束对于 x 的定义域的选择是敏感的（n 的定义域必须为 nat）。寻找极限比替换更难，并且该结果仍不能保证产生一个不动点。这里可以和单调性一起定义一个称为“连续性”的性质，这样就充分保证极限是一个最小不动点，但还是将此讨论留给其他的书本。

----------递归数据构造结束

每当增加公理时, 都必须注意保持它们与已有的理论一致。选错公理可能引起不一致。下面就是一个例子。假设有如下公理：

$$bad \;=\; \S n: nat \cdot \neg n: bad$$

因此, *bad* 就定义成所有不在 *bad* 中的自然数组成的束。从这个公理可得

$0: bad$

$=$ $0: \S n: nat \cdot \neg n: bad$

$=$ $\neg 0: bad$

是一个定理。根据完备性规则, $0: bad = \neg 0: bad$ 也是一个反定理。为了避免这种不一致, 必须撤消这个公理。

有时递归构造不能产生任何答案。例如, 前一段中的不动点等式可导致如下束序列：

$bad_0 = null$

$bad_1 = nat$

$bad_2 = null$

依此下去, 轮流出现 *null* 和 *nat*。我们无法说 bad_∞是什么, 因为不知道 ∞ 是偶数还是奇数。就连极限公理也不能告诉我们什么。当不存在不动点时, 不能归咎于利用递归构造式方法找不到不动点。不过, 有时即使存在不动点，也会出现利用递归构造式方法找不到这个不动点的情形(参见练习 361) 。

----------递归数据定义结束

6.1 递归程序定义

程序, 更普遍地, 规范, 可以用公理来定义, 就象定义数据一样。在下面的第一个例子中, x 和 y 是整数变量, 引入名 *zap*, 并且给出以下不动点等式作为公理：

$$zap \;=\; \textbf{if } x{=}0 \textbf{ then } y{:=}0 \textbf{ else } x := x{-}1.\;\; t{:=}\, t{+}1.\; zap \textbf{ fi}$$

该等式的右式为构造式, 以下是它的六个解：

(a) $x \geq 0 \;\Rightarrow x' = y' = 0 \;\wedge t' = t{+}x$

(b) **if** $x \geq 0$ **then** $x' = y' = 0 \wedge t' = t + x$ **else** $t' = \infty$ **fi**

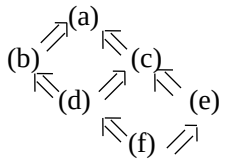
(c) $x' = y' = 0 \;\wedge (x \geq 0 \;\Rightarrow t' = t{+}x)$

(d) $x' = y' = 0 \;\wedge$ **if** $x \geq 0$ **then** $t' = t{+}x$ **else** $t' = \infty$ **fi**

(e) $x' = y' = 0 \;\wedge t' = t{+}x$

(f) $x \geq 0 \wedge x' = y' = 0 \;\wedge t' = t{+}x$

虽然这些解不是完全排好序的，但解(a)是最弱解, 解(f)是最强解。这些解之间的次序可由下图表示：



解(e)和(f)太强了, 以致于它们不可实现。解(d)是可实现的，因为它也是确定的, 因此它是可实现解中最强的一个。

从定义 *zap* 的不动点等式，并不能说它等同于某一个特定解。不过可以这样说： *zap* 精化了最弱解：

$$x \geq 0 \;\Rightarrow x' = y' = 0 \wedge t' = t{+}x \;\Leftarrow\; zap$$

所以可以用它来求解问题。并且 *zap* 可被它的构造式精化：

$$zap \;\Leftarrow\; \textbf{if } x{=}0 \textbf{ then } y{:=}0 \textbf{ else } x{:=}x{-}1.\; t{:=}t{+}1.\; zap \textbf{ fi}$$

这样就可以执行 *zap* 了。从实用目的考虑, 做到这样也就足够了。但如果希望定义 *zap* 为解(a)，可以通过增加一个定点归纳公理来实现：

$$\forall \sigma, \sigma' \cdot\; (Z \;=\; \textbf{if } x{=}0 \textbf{ then } y{:=}\,0 \textbf{ else } x{:=}\,x{-}1.\;\; t{:=}\,t{+}1.\;\; Z \textbf{ fi})$$
$$\Rightarrow\;\; \forall \sigma, \sigma' \cdot\; zap \;\Leftarrow\; Z$$

这个归纳公理表示：如果任意规范 *Z* 满足构造公理，那么 *zap* 比 *Z* 弱或和 *Z* 一样。所以 *zap* 是构造公理的最弱解。

尽管我们用定点构造和归纳来定义 *zap*，我们同样也可以用普通构造和归纳来定义它。

if x=0 **then** y:= 0 **else** x:= x–1. t:= t+1. *zap* **fi** $\Leftarrow$ *zap*

$\forall\sigma, \sigma' \cdot$ (**if** x=0 **then** y:= 0 **else** x:= x–1. t:= t+1. Z **fi** $\Leftarrow$ Z)

$\Rightarrow$ $\forall\sigma, \sigma' \cdot$ *zap* $\Leftarrow$ Z

6.1.0 递归程序构造

递归程序构造与递归数据构造类似, 且有相似的用途。这里用 *zap* 为例来说明这一过程。从 zap_0 开始尽最大努力描述计算过程, 先不参考 *zap* 的定义。当然, 如果不看定义, 就不知道 *zap* 描述了什么样的计算, 所以可以从能够满足每一个计算的规范开始：

$$zap_0 = \mathsf{T}$$

zap 的下一个描述可以通过用 zap_0 去替换构造式中的 *zap* 而获得, 并依次下去：

zap_1 = **if** x=0 **then** y:=0 **else** x := x–1. t := t+1. zap_0 **fi**

= x=0 $\Rightarrow x' = y' = 0 \ \wedge t' = t$

zap_2 = **if** x=0 **then** y:=0 **else** x := x–1. t := t+1. zap_1 **fi**

= $0 \le x < 2 \Rightarrow x' = y' = 0 \ \wedge t' = t+x$

一般地, zap_n 描述了 n 次使用其构造式之后的计算。现在可以猜测 (并且如果愿意，可以用 *nat* 归纳去证明)：

$$zap_n = 0 \le x < n \Rightarrow x' = y' = 0 \ \wedge t' = t+x$$

下一步用 ∞ 来代替 n ：

$$zap_\infty = 0 \le x < \infty \Rightarrow x' = y' = 0 \ \wedge t' = t+x$$

最后, 必须测试这一结果是否满足 *zap* 的不动点公理。

(等式右边用 zap_∞代替 *zap*)

= **if** x=0 **then** y:=0 **else** x:=x–1. t:=t+1. $0 \le x \Rightarrow x' = y' = 0 \wedge t' = t+x$ **fi**

= **if** x=0 **then** $x' = y' = 0 \ \wedge t' = t$ **else** $0 \le x-1 \Rightarrow x' = y' = 0 \ \wedge t' = t+x$ **fi**

= $0 \le x \Rightarrow x' = y' = 0 \wedge t' = t+x$

= (等式左边用 zap_∞代替 *zap*)

它满足不动点等式, 实际上它是最弱不动点。

如果不考虑时间, 那么对一个未知计算, 我们所知的只能是 T, 并且从它开始进行递归构造。如果考虑时间, 仅说 T 就不够了；还应加上时间不会减少。从 $t' \ge t$ 开始可以构造一个较强的不动点。

zap_0 = $t' \ge t$

zap_1 = **if** x=0 **then** y:=0 **else** x:=x–1. t:=t+1. zap_0 **fi**

= **if** x=0 **then** $x' = y' = 0 \ \wedge t' = t$ **else** $t' \ge t+1$ **fi**

zap_2 = **if** x=0 **then** y:=0 **else** x:=x–1. t := t+1. zap_1 **fi**

= **if** x=0 **then** $x' = y' = 0 \wedge t' = t$ **else if** x=1 **then** $x' = y' = 0 \ \wedge t' = t+1$ **else** $t' \ge t+2$ **fi fi**

$=$ **if** $0 \le x < 2$ **then** $x' = y' = 0 \wedge t' = t+x$ **else** $t' \ge t+2$ **fi**

一般地, zap_n 描述了到时间 n 时所知道的情况。现在可以猜测 (并且如果愿意，可以用 *nat* 归纳证明)：

$$zap_n = \textbf{if } 0 \le x < n \textbf{ then } x' = y' = 0 \wedge t' = t+x \textbf{ else } t' \ge t+n \textbf{ fi}$$

用 ∞ 代替 n, 有

$$zap_\infty = \textbf{if } 0 \le x \textbf{ then } x' = y' = 0 \wedge t' = t+x \textbf{ else } t' = \infty \textbf{ fi}$$

下面测试这个结果：

(等式右边用 zap_∞ 代替 *zap*)

$=$ **if** $x=0$ **then** $y:=0$ **else** $x:=x-1.\ t:=t+1.$ **if** $0 \le x$ **then** $x' = y' = 0 \wedge t' = t+x$ **else** $t' = \infty$ **fi fi**

$=$ **if** $x=0$ **then** $x' = y' = 0 \wedge t' = t$ **else if** $0 \le x-1$ **then** $x' = y' = 0 \wedge t' = t+x$ **else** $t' = \infty$ **fi fi**

$=$ **if** $0 \le x$ **then** $x' = y' = 0 \wedge t' = t+x$ **else** $t' = \infty$ **fi**

$=$ (等式左边用 zap_∞ 代替 *zap*)

从 $t' \ge t$ 开始进行递归构造, 就构造了一个较强但仍可实现的不动点。在本例中，如果从 $\perp$ 开始进行递归构造, 就可以获得一个最强不动点, 但它是不可实现的。

通过将 n 替换为 ∞ 我们得到一个不动点的侯选式 zap_∞。另一个侯选式可以是 $LIMn \cdot zap_n$。在本例中，两个侯选式是等价的，但在其他的例子中这两种形成侯选式的方法可能导致不同的结果。

----------递归程序构造结束

6.1.1 循环定义

循环可以用构造和归纳来定义。**while**-循环的公理是：

$t' \ge t \Leftarrow$ **while** b **do** P **od**

if b **then** $P.\ t:=t+1.$ **while** b **do** P **od else** *ok* **fi** $\Leftarrow$ **while** b **do** P **od**

$\forall \sigma, \sigma' \cdot\ t' \ge t \wedge$ **if** b **then** $P.\ t:=t+1.\ W$ **else** *ok* **fi** $\Leftarrow W$

$\Rightarrow\ \forall \sigma, \sigma' \cdot$ **while** b **do** P **od** $\Leftarrow W$

其中包含了递归时间，但这个时间可替换成任何其他时间度量规则。以下三条定义 *nat* 的公理是以上公理极其相似的类比：

$0 : nat$

$nat + 1 : nat$

$0, B+1 : B \Rightarrow nat : B$

第一条 **while**-循环公理是基本情况，说明至少时间是不会减少的。第二条构造公理只用了一步，表明 **while** b **do** P **-od** 精化(实现)了它的第一次展开式；根据逐步精化定律, 它也就精化了其任意一次展开。最后一条归纳公理，表明它是可以满足前两个公理的最弱规范。

利用这些公理, 可以证明一些称为不动点构造和不动点归纳的定理。对于 **while**-循环, 这些定理是：

while b **do** P **od** $=\ t' \ge t \wedge$ **if** b **then** $P.\ t:=t+1.$ **while** b **do** P **od else** *ok* **fi**

$\forall\sigma, \sigma' \cdot (W = t' \geq t \wedge \textbf{if } b \textbf{ then } P.\ t := t+1.\ W \textbf{ else } ok \textbf{ fi})$

$\Rightarrow \quad \forall\sigma, \sigma' \cdot \textbf{while } b \textbf{ do } P \textbf{ od} \Leftarrow W$

这里的定理不同于第五章中提出的；因此在获得一些定理的同时也失去了一些定理。例如, 根据这个定义，不再能够证明下式：

$$x' \geq x \Leftarrow \textbf{while } b \textbf{ do } x' \geq x \textbf{ od}$$

----------循环定义结束

----------递归程序定义结束

---------- 递归定义结束

第七章 理论设计与实现

程序员可以使用程序设计语言的设计者和实现者提供的形式、抽象、理论、结构。并且每写一个程序, 每定义一个名, 程序员就创建了新的形式、抽象、理论、结构。为了使他们的创造尽可能优美和有用, 程序员应当意识到他们不仅是理论的设计者和实现者, 也是理论的使用者。

在程序设计中, 堆栈(stack)、队列(queue)、树(tree)都是经常用到的标准数据结构。本章的目的不是介绍它们在应用中如何有用, 这些都留待专门介绍数据结构的书去讨论。它们在这里是作为理论设计与实现的研究实例加以介绍, 它们都包含了某种类型的项。例如, 整数堆栈、 二进制值表堆栈, 甚至堆栈的堆栈。在本章中, X 代表一个数据结构的项束(或类型)。

7.0 数据理论

7.0.0 数据－堆栈 理论

堆栈是程序设计语言实现系统的一种很有用的数据结构。其显著特征是：任何时候, 下次被检查或被删除的项总是最新进入堆栈的那个项。其格言为：后进先出。

引入语法 *stack* , *empty* , *push* , *pop* 以及 *top*。非形式地，它们的含义可描述如下：

stack 项为类型 X 的所有堆栈组成的束

empty 不包含任何项（堆栈束的一个元素）的堆栈

push 一个函数： 给定一个堆栈和项, 返回一个新堆栈，其项为原来堆栈的项加上那个新项

pop 一个函数： 给定一个堆栈, 返回一个新堆栈，其项为原来堆栈的项去掉最新进入的项

top 一个函数： 给定一个堆栈, 返回最新进入堆栈的那个项

以下是有关它们的首先四个公理：

$empty: stack$

$push : stack \to X \to stack$

$pop : stack \to stack$

$top : stack \to X$

我们希望 *empty* 和 *push* 成为 *stack* 的构造式, 希望由 *pop* 获得的堆栈也可由 *empty* 和 *push* 构造出来, 这样 *pop* 就不必作为一个构造式。构造公理可以用下面任何一种方式写出：

$empty, push\ stack\ X: stack$

$P\ empty \wedge \forall s: stack \cdot \forall x: X \cdot Ps \Rightarrow P(push\ s\ x) \Leftarrow \forall s: stack \cdot Ps$

其中 $P: stack \to bin$， *push* 可以分布于束的合并。为了防止任何其他东西成为堆栈, 需要一个归纳公理, 它可以用许多方式写出来, 下面给出两种：

$empty, push\ B\ X: B \Rightarrow stack: B$

$P\ empty \wedge \forall s: stack \cdot \forall x: X \cdot Ps \Rightarrow P(push\ s\ x) \Longrightarrow \forall s: stack \cdot Ps$

根据目前的公理, 有可能得出所有堆栈都相等的结论。为了能够说明此构造式总是构造不同的堆栈还需要另外两个公理。令 s, t 为 *stack* 的元素，且令 x, y 为 X 的元素, 那么有

$push\ s\ x \neq empty$

$push\ s\ x = push\ t\ y \ = \ s=t \ \wedge x=y$

最后, 还需要两个公理来表明堆栈具有“后进先出”的特性：

$pop\ (push\ s\ x) = s$

$top\ (push\ s\ x) = x$

至此完成了所有数据－堆栈公理。

----------数据－堆栈理论结束

使用数据－堆栈理论可以根据需要说明许多堆栈变量, 并且在表达式中根据公理应用它们。可以说明变量 a 和 b 为类型 *stack*, 然后写出赋值句 $a := empty$ 和 $b := push\ a\ 2$。

7.0.1 数据－堆栈实现

如果需要用一个堆栈, 而在程序语言中又没有提供堆栈功能, 就必须用所提供的数据结构来建造一个堆栈。假设表和函数已实现, 那么可用如下定义实现整数堆栈：

$stack = [*int]$

$empty = [nil]$

$push = \langle s: stack \rightarrow \langle x: int \rightarrow s+[x] \rangle \rangle$

$pop = \langle s: stack \rightarrow \textbf{if}\ s=empty\ \textbf{then}\ empty\ \textbf{else}\ s[0;..\#s-1]\ \textbf{fi} \rangle$

$top = \langle s: stack \rightarrow \textbf{if}\ s=empty\ \textbf{then}\ 0\ \textbf{else}\ s\ (\#s\ -1)\ \textbf{fi} \rangle$

为证明一个理论是实现了的，需要证明

（该理论的公理）⇐（实现的定义）

换句话说，定义必须满足公理。根据分配律，可以一次证明一个公理，例如最后一条公理的证明：

$top\ (push\ s\ x) = x$	替换 *push*
$= top\ (\langle s: stack \rightarrow \langle x: int \rightarrow s+[x] \rangle \rangle\ s\ x) = x$	函数应用
$= top\ (s+[x]) = x$	替换 *top*
$= \langle s: stack \rightarrow \textbf{if}\ s=empty\ \textbf{then}\ 0\ \textbf{else}\ s(\#s-1)\ \textbf{fi} \rangle\ (s+[x]) = x$	函数应用并替换 *empty*
$= \textbf{if}\ s+[x]=[nil]\ \textbf{then}\ 0\ \textbf{else}\ (s+[x])\ (\#(s+[x]-1)\ \textbf{fi} = x$	简化 **if** 和索引
$= (s+[x])\ (\#s) = x$	做表索引
$= x = x$	自反定律
$= \mathsf{T}$	

----------数据-堆栈实现结束

堆栈理论是一致的吗? 由于使用表理论实现了它, 因而如果表理论一致，那么堆栈理论也就一致。堆栈理论完备吗? 要说明一个关于堆栈的二元表达式是不可分类的, 就必须实现堆栈两次, 一次使得该表达式成为定理, 另一次成为反定理。表达式

$pop\ empty = empty$

$top\ empty = 0$

在前面的实现中是定理, 不过可将实现改变成如下：

pop = ⟨*s*: *stack*→**if** *s*=*empty* **then** *push empty* 0 **else** *s* [0;..#*s* −1] **fi**⟩

top = ⟨*s*: *stack*→**if** *s*=*empty* **then** 1 **else** *s* (#*s* −1) **fi**⟩

在这个新实现中, 上面两个表达式成为反定理，所以堆栈理论是不完备的。

堆栈理论指出了堆栈的特性。若要实现堆栈, 必须保证所有这些特性都要提供。若要使用堆栈, 必须保证仅仅这些特性才可靠。有一点理应强调： 理论是实现者和使用者(他们可能为同一个人的两个面貌, 或者两个公司)之间的契约。它清楚地表明了双方的义务和权利。如果出现了错误, 它指出了该错误应由谁负责。理论可以让任何一方修改程序中自己的部分而不必知道其他部分是如何书写的。这是构造大规模软件的一条基本原则。在这个小例子中, 尽管堆栈实现者提供了特性 *pop empty* = *empty*, 但堆栈使用者不可以使用它；如果使用者需要它, 应当把它加进理论中。

7.0.2 简单 数据－堆栈 理论

在前面介绍的数据－堆栈理论中, 有公理 *empty*: *stack* 和 *pop*: *stack*→*stack*；利用它们可以证明 *pop empty*: *stack*。换句话说, 弹出空堆栈得到一个堆栈, 尽管不知道是什么堆栈。实现者有义务给 *pop empty* 提供一个堆栈, 尽管它是什么无关紧要。如果根本不想对一个空堆栈进行弹出操作, 那么这个理论就太强了。应当弱化公理 *pop*: *stack*→*stack*, 并且从实现者的义务中，去掉提供那些不想要的东西的要求。更弱一点的公理为：

$$s \neq empty \Longrightarrow pop\ s\text{: } stack$$

表示弹出一个非空堆栈产生一个堆栈，不过它可由剩下的其他一些公理推出, 并非是必要的。类似地, 由 *empty*: *stack* 和 *top*: *stack* → *X* 可以证明 *top empty*: *X*；去掉公理 *top*: *stack* → *X* 就为实现者去掉了为 *top empty* 提供不想要的结果的义务。

如果确信无需对堆栈证明任何东西, 并且没有 *stack* 归纳公理也没有关系。但经过思考，可以认识到永远不需要空堆栈，也不必去测试一个堆栈是否为空。因为总可作用于一个给定(可能非空)堆栈的顶部, 而且在大多数应用中，必须这样做, 不对堆栈的其他部分作任何修改。可以删除公理 *empty*: *stack* 和所有提及 *empty* 的地方。这样必须用弱化的公理 *stack* ≠ *null* 来代替原来的公理, 以便仍旧可以用类型 *stack* 说明变量。如果要测试一个堆栈是否为空, 应先将一个特殊的值推入堆栈, 这个特殊值今后不会被再次推入堆栈；这样空堆栈测试就变成测试顶部是否为那个特殊值了。

就大多数用途而言, 能够向堆栈推入项, 从堆栈弹出项, 查看堆栈顶部项的内容就已足够了。这样需要的理论比前面介绍的理论简单得多。这个简单的数据━堆栈理论引入了名 *stack*, *push*, *pop* 和 *top*, 并使用了如下四个公理： 令 *s* 为 *stack* 的一个元素，令 *x* 为 *X* 的一个元素, 有

stack ≠ *null*

push s x: *stack*

pop(*push s x*) = *s*

top(*push s x*) = *x*

----------简单数据-堆栈理论结束

为了研究堆栈, 若作为一项数学活动, 要求尽可能最强的公理, 以便能够尽可能多地进行证明。若作为一项工程活动, 理论设计是排除所有不需要的实现, 同时又允许所有其他实现的一门艺术。设计一个比实际需要更强的理论是不利于生产的；它使得实现更加困难, 并且也使理论的扩充更为困难。

7.0.3 数据－队列理论

队列数据结构, 也被看作一个缓冲区, 在模拟与调度应用中非常有用。它的显著特征是： 在任何时候, 下次被检查或被删除的项总是最早进入的那个项。它的格言是：先进先出。

引入语法 *queue*, *emptyq*, *join*, *leave* 以及 *front*, 它们的非形式化含义如下：

queue	项为类型 *X* 的所有队列组成的束
emptyq	不包含任何项的队列（束 *queue* 的一个元素）
join	一个函数： 给定一个队列和项，返回一个新队列，其项包含原队列的项外加上所给的新项
leave	一个函数： 给定一个队列，返回一个新队列，其项包含原队列的项去掉最早进入的项
front	一个函数： 给定一个队列，返回最早进入的项

有关设计堆栈理论的各种考虑同样指导着队列理论的设计。令 *q*, *r* 为 *queue* 的元素，*x*, *y* 为 *X* 的元素。当然希望有如下的构造公理：

emptyq: *queue*

join q x: *queue*

如果要证明有关 *join* 定义域的东西，那么必须将第二个构造公理用下列更强的公理代替：

$$join: queue \rightarrow X \rightarrow queue$$

为了说明这些构造公理可构造不同的队列，没有重复，需要有：

$join\ q\ x \neq emptyq$

$join\ q\ x = join\ r\ y \ \Leftrightarrow\ q=r \wedge x=y$

希望由 *leave* 获得的队列可以通过 *emptyq* 和 *join* 来构造, 因此不需要

$$leave\ q: queue$$

用来构造，并且也不希望迫使实现者为 *leave emptyq* 提供一种表示, 所以将忽略这一公理。希望

$$q \neq emptyq \Rightarrow leave\ q: queue$$

并且类似地, 希望

$$q \neq emptyq \Rightarrow front\ q: X$$

如果希望证明有关所有队列的某些属性, 就需要 *queue* 的归纳公理：

$$emptyq,\ join\ B\ X: B \Rightarrow queue: B$$

最后, 给出队列“先进先出”的特征：

$leave\ (join\ emptyq\ x) = emptyq$

$q \neq emptyq \Rightarrow leave(join\ q\ x) = join\ (leave\ q)\ x$

$front\ (join\ emptyq\ x) = x$

$q \neq emptyq \Rightarrow front(join\ q\ x) = front\ q$

如果决定保留 *queue* 的归纳公理，那么下面两个早先的公理就可以丢掉了：

$q \neq emptyq \Rightarrow leave\ q: queue$

$q \neq emptyq \Rightarrow front\ q: X$

因为现在可以证明它们了。

----------数据—队列理论结束

数据－堆栈实现之后，数据－队列的实现没有引起新问题，所以将它留作练习 389。

7.0.4 数据－树理论

引入语法：

tree	项为类型 *X* 的所有有限二叉树组成的束
emptree	不包含任何项的树
graft	一个函数：给定两个树和一个项，返回一个新树，它以给定项作为根，两个给定的树分别作为左右两个子树
left	一个函数：给定一个树，返回它的左子树
right	一个函数：给定一个树，返回它的右子树
root	一个函数：给定一个树，返回其根的项

从研究树的目的出发，需要一个强理论。令 t, u, v, w 为 *tree* 的元素，且令 x, y 为 X 的元素。

$emptree: tree$

$graft: tree \rightarrow X \rightarrow tree \rightarrow tree$

$emptree,\ graft\ B\ X\ B: B \Rightarrow tree: B$

$graft\ t\ x\ u \neq emptree$

$graft\ t\ x\ u = graft\ v\ y\ w = t=v \wedge x=y \wedge u=w$

$left\ (graft\ t\ x\ u) = t$

$root\ (graft\ t\ x\ u) = x$

$right\ (graft\ t\ x\ u) = u$

其中在构造公理中，*graft* 对束的并可分配。

从大多数程序设计目的出发，以下简单的、较弱的理论已经足够了：

$tree \neq null$

$graft\ t\ x\ u: tree$

$left\ (graft\ t\ x\ u) = t$

$root\ (graft\ t\ x\ u) = x$

$right\ (graft\ t\ x\ u) = u$

如同堆栈，实际上不必给出一棵空树。只要被提供一棵树，就可以建造一个有与众不同的根的树，让其表示空树。并且可能不需要 *tree* 归纳。

----------数据-树理论结束

7.0.5 数据－树实现

假设表和递归数据定义已实现，那么可以用如下定义实现一个整数树：

tree = *emptree*, *graft tree int tree*

emptree = [*nil*]

graft = ⟨*t*: *tree*→⟨*x*: *int*→⟨*u*: *tree*→[*t*; *x*; *u*]⟩⟩⟩

left = ⟨*t*: *tree*→*t* 0⟩

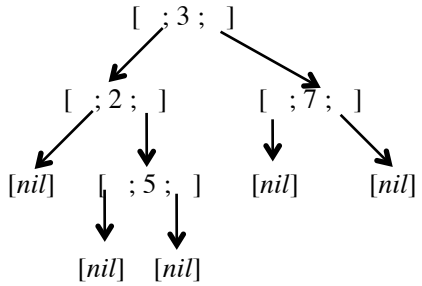
right = ⟨*t*: *tree*→*t* 2⟩

root = ⟨*t*: *tree*→*t* 1⟩

过程 *graft* 生成由三个项组成的表，其中两个项可以是表本身。对于表的一个合理实现策略是：对于每个项，分配一个小的空间，其中只能存放一个整数或者数据地址。如果一个项是整数，那么将它直接放入它的存贮空间；如果一个项是表，那么表放在别的地方，而将它的指针(数据地址)放入它的存贮空间。在这个表实现系统中，当需要时指针是自动提供的。例如，树

[[[*nil*]; 2; [[*nil*];5;[*nil*]]];3; [[*nil*];7;[*nil*]]]

可表示成



下面是另一个数据—树的实现：

tree = *emptree*, *graft tree int tree*

emptree = 0

graft = ⟨*t*: *tree*→⟨*x*: *int*→⟨*u*: *tree*→(“left”→*t* | “root”→*x* | “right” →*u*)⟩⟩⟩

left = ⟨*t*: *tree*→*t* “left”⟩

right = ⟨*t*: *tree*→*t* “right”⟩

root = ⟨*t*: *tree*→*t* “root”⟩

在这种实现下，一个树的值看起来如下：

```
“left” → ( “left” → 0
         | “root” → 2
         | “right” → ( “left” → 0
                     | “root” → 5
                     | “right” → 0) )
| “root” → 3
| “right” ( “left” → 0
           | “root” → 7
           | “right” → 0 )
```

如果所用的实现系统没有包括递归数据定义功能, 那么必须自己建立一个指针结构。例如, 用 C 语言可以实现一个二叉树如下：

```
struct tree { struct tree *left; int root; struct tree *right; };
struct tree *emptree=NULL;
struct tree *graft(struct tree *t, int t, strect tree *u)
    {   struct tree *g; g=malloc(size of(struct tree));
        (*g).left=t; (*g).root=x; (*g).right=u;
        return g;
    }
struct tree *left(struct tree *t) { return (*t).left; }
int root(struct tree *t) {return (*t).root; }
struct tree *right (struct tree *t) { return (*t).right; }
```

可以看到, 用 C 编写的实现代码很笨拙, 因此将程序理论直接应用到 C 编码上不是一个好主意。在递归数据定义尚未实现时使用指针(数据地址)就像在递归程序定义尚未实现或实现得很糟时使用 **go to**(程序地址)一样。

----------数据—树实现结束

----------数据理论结束

数据理论创建了一个新类型, 或者一个值空间, 或者一个旧类型的扩充。程序理论创建了新程序, 或者说, 创建了新规范, 当理论实现时它就变成了程序。这两种类型的理论对应着两种类型的程序设计： 函数式和命令式。

7.1 程序理论

在程序理论中, 状态划分成两类变量：使用者变量和实现者变量。理论的使用者可以直接访问使用者变量，但不可直接访问(查看或改变)实现者变量。使用者只能通过理论去访问实现者变量。而另一方面, 理论的实现者可以直接访问实现者变量, 但不可直接访问(查看或改变)使用者变量。并且实现者也只能通过理论去访问使用者变量。一些程序设计语言中的“模块”或者“对象”的构造就是用于这一目的。而在另外一些语言中, 只用禁止对不属于自己这边的变量的非授权使用。

如果仅仅需要一个堆栈、一个队列、或一棵树, 可以通过隐式方式获得一个经济的表达式和执行。如果只有一个堆栈, 就不必指明将数据推入哪一个堆栈，对于其他操作和数据结构也可作类似的处理。本书介绍的每一种程序理论将只提供其一种数据结构给使用者, 但可通过给每个操作附加一个参数使其普遍化。

7.1.0 程序－堆栈理论

最简单的程序－堆栈理论引入了三个名： *push* (一个带有类型 *X* 参数的过程), *pop* (一个程

序), *top*(类型为 *X* 的项)。在这一理论中, *push* 3 是一个程序(假定 3: *X*)，它将改变状态。在这个程序之后, 并且在任何其他的推入和弹出操作之前, *print top* 将打印 3。下面两个公理已足够使用了：

$$top' = x \Leftarrow push\ x$$

$$ok \Leftarrow push\ x.\ pop$$

其中 *x*: *X*。

上面第二个公理表明一个 *pop* 操作解除一个 *push* 操作。实际上, 它说明任意自然次数的 *push* 操作可由相同次数的 *pop* 操作来解除。

$$\begin{aligned} & ok && \text{利用第二个公理} \\ \Leftarrow\ & push\ x.\ pop && \text{对于相关组合 } ok \text{ 是恒等的} \\ =\ & push\ x.\ ok.\ pop && \text{再次利用第二个公理进行逐步精化} \\ \Leftarrow\ & push\ x.\ push\ y.\ pop.\ pop \end{aligned}$$

可以证明类似以下的形式

$$top' = x \Leftarrow push\ x.\ push\ y.\ push\ z.\ pop.\ pop$$

这表明当把一些东西推入堆栈后, 会在适当的时候发现它们。这正是实际所需要的。

---程序—堆栈理论结束

7.1.1 程序－堆栈实现

为了实现程序－堆栈理论，引入一个实现者变量 *s*: [*X]并定义：

$$push = \langle x: X \rightarrow s := s + [x] \rangle$$

$$pop = s := s[0; .. \#s - 1]$$

$$top = s\ (\#s - 1)$$

当然, 必须证明上述定义满足前面的公理。这里只做第一个公理, 余下的留作练习 391。

$$\begin{aligned} & (top' = x \Leftarrow push\ x) && \text{利用 } push \text{ 和 } pop \text{ 的定义} \\ = & \ (s'\ (\#s' - 1) = x \Leftarrow s := s + [x]\ ) && \text{表理论} \\ = & \ \mathsf{T} \end{aligned}$$

---程序—堆栈实现结束

7.1.2 复杂程序－堆栈理论

刚刚介绍的程序－堆栈理论对应于早先介绍的简单数据－堆栈理论。这里介绍一种比前面要稍微复杂一点的程序－堆栈理论, 它引入两个新名： *mkempty*(使堆栈为空的程序)和 *isempty*(判断堆栈是否为空的条件)。令 *x*: *X*, 公理是

$$top' = x \wedge \neg isempty' \Leftarrow push\ x$$

$$ok \Leftarrow push\ x.\ pop$$

$$isempty' \Leftarrow mkempty$$

---复杂程序—堆栈理论结束

一旦利用表实现程序－堆栈理论, 就可知如果表理论是一致的, 那么程序－堆栈理论也是一致的。与数据－堆栈理论一样, 程序－堆栈理论也是不完备的。这种不完备性对实现者来说是一种自由, 他们可以尽量从经济方面而不是从健壮性方面考虑实现的代价。如果关心如何进行权衡的话, 就应该强化这个理论。例如, 可以增加一个公理：

$$print\ \text{"error"} \Leftarrow mkempty.\ pop$$

7.1.3 弱程序－堆栈理论

前面第一个介绍的程序－堆栈理论可以被弱化, 并且仍然保持它的堆栈特性。必须保留公理

$$top' = x \Leftarrow push\ x$$

但不须要组合式 *push x*. *pop* 来保持所有变量不变。仍然要求：任意自然数次数的 *push* 操作之后再作相同次数的 *pop* 操作, 得到原来的堆栈顶项。相应公理为：

$top' = top \Leftarrow balance$

$balance \Leftarrow ok$

$balance \Leftarrow push\ x.\ balance.\ pop$

其中 *balance* 是为帮助写公理而引入的规范，但它不是对理论的扩充，因此不需实现。为了证明实现是正确的，必须提出一个 *balance* 的定义，这个定义用了实现者变量，但它不必是一个程序。这个弱理论允许实现系统中的弹出操作不必将实现者变量 *s* 恢复到推入堆栈操作之前的值, 而是将最后的项标志成“废料”。

一个弱理论可以用一个强理论所无法采用的方式进行扩充。例如, 可以加入名 *count* (具有类型 *nat*)和 *start* (一个程序), 带有如下公理：

$count' = 0 \Leftarrow start$

$count' = count+1 \Leftarrow push\ x$

$count' = count+1 \Leftarrow pop$

这样, *count* 就可统计自最后一次用 *start* 之后推入和弹出的次数。

----------弱程序—堆栈理论结束

7.1.4 程序－队列理论

程序—队列理论引入五个名： *mkemptyq*(一个函数, 使队列为空), *isemptyq*(一个条件, 表明队列是否为空), *join*(一个过程, 其参数具有类型 *X*), *leave*(一个程序)以及 *front* (为类型 *X*)。公理为：

$isemptyq' \Leftarrow mkemptyq$

$isemptyq \Rightarrow front' = x \wedge \neg isemptyq' \Leftarrow join\ x$

$\neg isemptyq \Rightarrow front' = front \wedge \neg isemptyq' \Leftarrow join\ x$

$isemptyq \Rightarrow (join\ x.\ leave = mkemptyq)$

$\neg isemptyq \Rightarrow (join\ x.\ leave = leave.\ join\ x)$

----------程序—队列理论结束

7.1.5 程序－树理论

通常, 有多种方法来说明这个理论。想象一棵树在所有方向都是无限的, 没有叶子也没有根。在树中的某个节点，将面对三种方向的可能：*up*（向该节点的父项）， *left*（向该节点的左子项），*right*（向该节点的右子项）。变量 *node*(具有类型 *X*)告诉你那个项的值，它可以被赋予一个新值。变量 *aim* 表示面对的方向，它可被赋予一个新的方向。程序 *go* 将按照你面对的方向移动到下一节点，并将你转回面向你来的方向。例如，我们可能从

$$aim := up.\ go$$

开始，然后看看 *aim* 得知我们从哪里来。后面我们可能赋值

$$node := 3$$

这个公理用了一个助于写公理的辅助规范，但它不是这个理论的延伸，也就没必要实现它：*work* 表示“ 做任何事，如果需要可以改变节点的值，但不从该节点（你的位置是 *work* 开始处的位置）以该方向（在 *work* 开始处变量 *aim* 的值）进行 *go* 操作。从你开始的地方结束，面向你开始面向的方向。”。公理如下：

$$(aim = up) = (aim' \neq up) \Leftarrow go$$

$$node' = node \wedge aim' = aim \Leftarrow go.\ work.\ go$$

$$work \Leftarrow ok$$

$$work \Leftarrow node := x$$

$$work \Leftarrow a{=}aim{\neq}b \wedge (aim := b.\ go.\ work.\ go.\ aim := a)$$

$$work \Leftarrow work.\ work$$

下面是定义程序－树的另一种方式。令 *T*（树）和 *p*（指针）为实现者变量。公理如下：

$$tree = [tree; X; tree]$$

$$T : tree$$

$$p : *(0,1,2)$$

$$node = T@(p;1)$$

$$change = \langle x: X \rightarrow T := (p;1) \rightarrow x \mid T \rangle$$

$$goUp = p := p_{0;\ldots\leftrightarrow p-1}$$

$$goLeft = p := p;0$$

$$goRight = p := p;2$$

如果字符串和@运算符已实现，则该理论已实现。若没有实现，则仍是一个理论，为清楚起见可与前面的理论进行比较。

----------程序—树理论结束

----------程序理论结束

7.2 数据转换

程序是计算机行为的规范。有时(并不总是)程序是最清楚的规范。有时它又是最容易写的规范。如果将一个规范写成一个程序, 就没有必要再去实现它了。尽管一个规范可能已经是一个程序了, 如果愿意, 仍然可以用不同的方法实现它。在一些程序设计语言中, 实现者变量被放在“模块”或“对象”中以便区分, 这样对它们的改变在模块或对象外是不可见的。也许选用的实

现者变量是用来使规范尽可能地清楚, 但其他实现者变量可能会获得更大的存贮效率, 或者提供更快的平均存取速度。正因理论的使用者除了通过该理论之外, 无法访问实现者变量, 所以实现者可以自由地以任何方式改变它们, 而不影响提供同样的理论给使用者。下面就是一种方法。

现在假设要用新的实现者变量 w 代替实现者变量 v，需采用一个数据转换式(data transformer)来完成这一转换, 它是一个与 v 和 w 相关联的二元表达式 D:

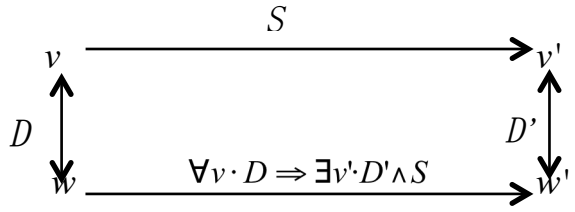
$$\forall\, w\cdot \exists\, v\cdot D$$

这里的 v 和 w 可以表示任意数量的变量。令 D' 与 D 相同, 只不过其所有变量都加上了撇号(')。那么这个理论中的每个规范 S 可转换成

$$\forall v\cdot D \Rightarrow \exists\, v'\cdot D' \wedge S$$

规范 S 含有非局部变量 v（及使用者变量）, 而转换后的规范含有非局部变量 w（及使用者变量）。

数据转换对使用者是不可见的。使用者想象实现者变量初始时状态为 v，然后根据规范 S，终结状态为 v'。事实上，实现者变量将具有根据 D 与 v 相关的初始状态 w, 但使用者仍然假设实现者变量具有状态 v, 因为 $\forall w\cdot \exists v\cdot D$。实现者变量将根据已转换的规范 $\forall v\cdot D \Rightarrow \exists\, v'\cdot D' \wedge S$, 从状态 w 改变成 w'。这说明无论使用者想象的相关初始变量 v 是什么, 总存在一个相关终结变量 v', 使用者可想象为 S 的结果, 因此这种想象是可以维持的。如下图所示:



在变量 (v, v') 上规范 S 的实现，经过转换 (D, D') 变成新的变量 (w, w') 上的新的规范。

第一个例子是练习 413(a)。其中使用者变量是 u: *bin*, 实现者变量是 v: *nat*。理论提供了三种运算, 具体为:

zero = $v := 0$

increase = $v := v+1$

inquire = $u := even\ v$

因为有关实现者变量的唯一问题是它是否为偶数, 所以我们决定根据数据转换式 $w = even\ v$，将原实现者变量用一个新的实现者变量 w: *bin* 来替换。这样, 第一个运算 *zero* 变成

$$\forall v\cdot w = even\ v \Rightarrow \exists\, v'\cdot w' = even\ v' \wedge (v := 0)$$

其中赋值句指向一个包含 u 和 v 的状态。

$$= \quad \forall v\cdot w = even\ v \Rightarrow \exists\, v'\cdot w' = even\ v' \wedge u' = u \wedge v' = 0$$ 单点定律

$$= \quad \forall v\cdot w = even\ v \Rightarrow w' = even\ 0 \wedge u' = u$$ 变量改变定律，简化

$$= \quad w' = \mathsf{T} \wedge u' = u$$ 现在状态包含 u 和 w。

$$= \quad w := \mathsf{T}$$

运算 *increase* 变成：

$$\forall v\cdot w = even\ v \Rightarrow \exists v'\cdot w' = even\ v' \wedge (v := v+1)$$

$$= \quad \forall v\cdot w = even\ v \Rightarrow \exists v'\cdot w' = even\ v' \wedge u' = u \wedge v' = v+1 \qquad \text{单点定律}$$

$$= \quad \forall v\cdot w = even\ v \Rightarrow w' = even\ (v+1) \wedge u' = u \qquad \text{变量改变定律，简化}$$

$$= \quad \forall r : even\ nat \cdot w = r \Rightarrow w' = \neg r \wedge u' = u \qquad \text{单点定律}$$

$$= \quad w' = \neg w \ \wedge u' = u$$

$$= \quad w := \neg w$$

运算 *inquire* 变成：

$$\forall v\cdot w = even\ v \Rightarrow \exists v'\cdot w' = even\ v' \wedge (u := even\ v)$$

$$= \quad \forall v\cdot w = even\ v \Rightarrow \exists v'\cdot w' = even\ v' \wedge u' = even\ v \wedge v' = v \qquad \text{单点定律}$$

$$= \quad \forall v\cdot w = even\ v \Rightarrow w' = even\ v \wedge u' = even\ v \qquad \text{变量改变定律}$$

$$= \quad \forall r : even\ nat \cdot w = r \Rightarrow w' = r \wedge u' = r \qquad \text{单点定律}$$

$$= \quad w' = w \ \wedge u' = w$$

$$= \quad u := w$$

在前面的例子中, 用一个小状态空间替换了一个大状态空间。练习 415(a)显示出两种状态空间都有效。其中使用者变量是 *u*: *bin*, 实现者变量是 *v*: *bin* 。理论提供了三种运算, 定义为：

$set \ = \ v := \mathsf{T}$

$flip \ = \ v := \neg v$

$ask \ = \ u := v$

根据数据转换式 $v = even\ w$, 将实现者变量 *v* 用一个新的实现者变量 *w*: *nat* 代替(也许是为了在一些计算机上更易于访问)。于是第一个运算 *set* 变成：

$$\forall v\cdot v = even\ w \Rightarrow \exists v'\cdot v' = even\ w' \wedge (v := \mathsf{T}) \qquad \text{单点定律, 使用两次}$$

$$= \quad even\ w' \ \wedge \ u' = u$$

$$\Leftarrow \quad w := 0$$

运算 *flip* 变成：

$$\forall v\cdot v = even\ w \Rightarrow \exists v'\cdot v' = even\ w' \ \wedge (v := \neg v) \qquad \text{单点定律, 使用两次}$$

$$= \quad even\ w' \neq even\ w \ \wedge \ u' = u$$

$$\Leftarrow \quad w := w+1$$

运算 *ask* 变成：

$$\forall v\cdot v = even\ w \Rightarrow \exists v'\cdot v' = even\ w' \ \wedge (\ u := v\ ) \qquad \text{单点定律, 使用两次}$$

$$= \quad even\ w' = even\ w = u'$$

$$\Leftarrow \quad u := even\ w$$

一个数据转换没有必要替换所有的实现者变量, 并且替换的变量数目与被替换的变量数目也不必相等。一个数据转换可以以小的转换的序列的形式一步步地完成。一个数据转换也可以通过小的转换的合并一个部分一个部分地完成。下面几个小节中的例子将说明以上观点。

7.2.0 安全开关

练习 418 是设计一个安全开关。它有三个二元用户变量 a，b 和 c。用户对变量 a 和 b 赋值作为开关的输入。开关的输出赋给 c。当两个输入都改变时输出也改变。更精确地说，当两个输入的值都与上次输出改变时它们的值不同时，输出的值改变。其概念是当一个用户想改变输出时切换他的开关，但输出要等到另一个用户也切换其开关表示同意改变输出时才能改变。如果第一个用户在第二个用户切换以前又将开关切换回原来的状态，那么输出仍然不会改变。

可以使用两个二元实现者变量来实现这个开关：

A 用来记录输出最后一次转换时输入 a 的状态

B 用来记录输出最后一次转换时输入 b 的状态

有两个运算：

$a := \neg a$. **if** $a \neq A \wedge b \neq B$ **then** $c := \neg c$. $A := a$. $B := b$ **else** ok **fi**

$b := \neg b$. **if** $a \neq A \wedge b \neq B$ **then** $c := \neg c$. $A := a$. $B := b$ **else** ok **fi**

在每个运算中，用户翻转（flip）输入变量的值，开关检查这个输入赋值是否使两个输入都与上次输出改变时的值不同；如果是，输出改变，并记录当前输入变量的值。该实现是这个问题的直接形式化，但可以通过数据转换进行简化。

根据转换式，对实现者变量 A 和 B 进行以下替换：

$$A = B = c$$

为了检查这是一个转换式，可以检查：

$\exists A, B \cdot A = B = c$ 对 A 和 B 都使用 c 普遍化

$\Leftarrow \top$

这里没有新的变量，因此没有全称量词。转换不影响对 a 和 b 的赋值，因此只有一个转换要做：

$\forall A, B \cdot A = B = c$

$\Rightarrow \exists A', B' \cdot A' = B' = c'$

$\wedge$ **if** $a \neq A \wedge b \neq B$ **then** $c := \neg c$. $A := a$. $B := b$ **else** ok **fi**

扩展赋值和 ok

$=$ $\forall A, B \cdot A = B = c$

$\Rightarrow \exists A', B' \cdot A' = B' = c'$

$\wedge$ **if** $a \neq A \wedge b \neq B$ **then** $a'=a \wedge b'=b \wedge c'=\neg c \wedge A'=a \wedge B'=b$

else $a'=a \wedge b'=b \wedge c'=c \wedge A'=A \wedge B'=B$ **fi**

对 A' 和 B' 进行单点定律

$=$ $\forall A, B \cdot A = B = c$

$\Rightarrow$ **if** $a \neq A \wedge b \neq B$ **then** $a'=a \wedge b'=b \wedge c'=\neg c \wedge c'=a \wedge c'=b$

else $a'=a \wedge b'=b \wedge c'=c \wedge c'=A \wedge c'=B$ **fi**

对 A 和 B 进行单点定律

$=$ **if** $a \neq c \wedge b \neq c$ **then** $a'=a \wedge b'=b \wedge c'=\neg c \wedge c'=\neg c \wedge c'=\neg c$

else $a'=a \wedge b'=b \wedge c'=c \wedge c'=c \wedge c'=c$ **fi**

$=$ **if** $a \neq c \wedge b \neq c$ **then** $c := \neg c$ **else** ok **fi**

$= \quad c := (a \neq c \wedge b \neq c) \neq c$

输出 c 成为 a， b 和 c 中占大多数的那个值。（作为一个电路，就是三个“异或 (exclusive or) ”门和一个“与 (and) ”门。）

----------安全开关结束

7.2.1 取一个数

下一个例子是练习 420(取一个数)：维护一个自然数组成的表, 其中的自然数代表它们正在“被使用”。三个运算为：

- 将表清空(初始化)
- 将变量 n 赋成一个当前没被使用的数, 并且将该数加入表中(它现在被使用了)
- 给定一个当前被使用的数 n, 将它从表中移出(它现在不再被使用了, 但以后还可再被使用)

使用者变量是 n: nat。虽然该练习讨论的是表, 但从这些运算可以看出, 所有的项都保证是不相同的, 它们的次序无关紧要，也没有嵌套结构；因此可以使用一个束变量。但那样就要在其上使用量词，所以这里它需要是一个元素。于是用集合变量 $s \subseteq \{nat\}$ 作为实现者变量。上面三个运算就成为：

$start \quad = \quad s' = \{null\}$

$take \quad = \quad \neg n' \in s \wedge s' = s \cup \{n'\}$

$give \quad = \quad n \in s \Rightarrow \neg n \in s' \wedge s' \cup \{n\} = s$

下面是一个数据转换式, 根据它可将集合 s 替换成自然数 m：

$$s \subseteq \{0,..m\}$$

这样就不是去严格维护那个被使用的数的集合, 而是维护一个可能更大的集合。仍然保证不会提供一个正被使用的数。*start* 可转换成如下：

$\forall s \cdot s \subseteq \{0,..m\} \Rightarrow \exists s' \cdot s' \subseteq \{0,..m'\} \wedge s' = \{null\}$ 单点定律和恒等律

$= \quad \mathsf{T}$

$\Leftarrow \quad ok$

这个转换后的规范正好是 T, 它最有效的精化式是 *ok*。因为 s 仅是包含于{0,..m}的子集, 不必等于{0,..m}, 所以 m 是什么没有关系；也可以将它放一边不管。运算 *take* 可转换成：

$\forall s \cdot s \subseteq \{0,..m\} \Rightarrow \exists s' \cdot s' \subseteq \{0,..m'\} \wedge \neg n' \in s \wedge s' = s \cup \{n'\}$

几个省略的步骤

$= \quad m \leq n' < m'$

$\Leftarrow \quad n := m.\ m := m+1$

运算 *give* 可转换成：

$\forall s \cdot s \subseteq \{0,..m\} \Rightarrow \exists s' \cdot s' \subseteq \{0,..m'\} \wedge (n \in s \Rightarrow \neg n \in s' \wedge s' \cup \{n\} = s)$

几个省略的步骤

$= \quad (n+1=m \Rightarrow n \leq m') \ \wedge \ (n+1<m \Rightarrow m \leq m')$

$\Leftarrow \quad ok$

由于进行了数据转换, 我们得到了该问题的一个极其有效的解。也许有人会争辩说现在根

本还没解决问题,因为没有维护一个“正被使用”的数表。但谁能说明这点? 对那个表的唯一使用就是获得一个当前还没被使用的数,而这一功能已经提供了。

对这个“取一个数”问题的实现对应于一台“取一个数”的机器, 这些机器通常运行于忙碌的服务中心。现在假设需要提供两台“取一个数”的机器, 它们能够独立运算。根据数据转换式 $s \subseteq \{0,..\max i\ j\}$, 可将 s 用两个变量 i, j: *nat* 来代替。那么运算 *take* 变成:

$$\forall s \cdot s \subseteq \{0,..\max ij\} \Rightarrow \exists s' \cdot s' \subseteq \{0,..\max i'\ j'\} \wedge \neg n' \in s \wedge s' = s \cup \{n'\}$$

几个省略的步骤

$$= \quad max\ i\ j \le n' < max\ i'\ j'$$

$$\Leftarrow \quad n := max\ i\ j.\ \textbf{if}\ i \ge j\ \textbf{then}\ i := i+1\ \textbf{else}\ j := j+1\ \textbf{fi}$$

从上面程序的最后一行, 可以看到这个数据转换式并没有提供所希望的两台机器的独立运算。或许一个不同的数据转换式会改善这种情况。将偶数放入一台机器, 而将奇数放入另一台机器。那么新的变量为 i: 2×*nat* 和 j: 2×*nat*+1 , 数据转换式为:

$$\forall k :\sim s \cdot even\ k \wedge k < i \vee odd\ k \wedge k < j$$

现在 *take* 就变成:

$$\forall s \cdot (\forall k :\sim s \cdot even\ k \wedge k < i \vee odd\ k \wedge k < j)$$
$$\Rightarrow \exists s' \cdot (\forall k :\sim s' \cdot even\ k \wedge k < i' \vee odd\ k \wedge k < j') \wedge \neg n' \in s \wedge s' = s \cup \{n'\}$$

几个省略的步骤

$$= \quad even\ n' \wedge i \le n' < i' \vee odd\ n' \wedge j \le n' < j'$$

$$\Leftarrow \quad (n := i.\ i := i+2\ ) \vee (n := j.\ j := j+2)$$

现在对问题得到了一个“分布”解:可以从其中任何一台机器中获得一个数, 而不干扰另外一台。分布的代价是失去了两台机器间所有的公平性;一个最近才到达的客户可能比更早达到的使用另一台机器的客户更早得到服务。

----------获得一个数字结束

7.2.2 语法分析(Parsing)

练习 411(语法分析):定义 E 为字母组成的字符串的一个束, 满足下面的不动点等式:

$$E \quad = \quad [\text{"}x\text{"}], [\text{"}\textbf{if}\text{"}]; E; [\text{"then"}]; E; [\text{"else"}]; E; [\text{"fi"}]$$

给一个字母组成的字符串, 写一个程序决定串是否在束 E 中。

为了使问题非平凡, 假设递归数据定义和束包含没有实现。解必须是一个搜索,因此需要一个变量表示仍在讨论中的串的束,初始时为 E 中的所有串,然后逐步消去串,当指定串被找到或剩余串中不包含指定串时结束。

令指定串为 s (一个常量)。第一个决定是从左向右分析,因此引入自然数变量 n, n 从 0 增加到至多 $\leftrightarrow s$, 指明我们已经分析过多少 s 的项。令 A 为一个变量, 它的值是字母组成的字符串的束。根据我们已经分析过的 s, 束 A 由 E 中可能是 s 的所有串所组成。可以用一个二元变量 q 的终结值表示结果。

为了减少需要考虑的情况，可以使用两个哨符。假设 s 以哨符[“eos”]（串的结束）作为结束；这是一个除了在 s 的结束处，不会在其他地方出现的项（有些程序设计语言自动提供了该哨符）。当对 A 进行初始化时，将在每个串的结束处加上哨符 [“eog”]（语法的结束），并假设 [“eog”]除了在 A 中的串的结束处，其他地方均不出现。问题和它的精化如下：

$$q' = (s_{0;..\leftrightarrow s-1} : E) \Leftarrow A := E;[\text{“eog”}].\ n := 0.\ P$$

其中，P = $n \le \leftrightarrow s \wedge A_{0;..n} = s_{0;..n} \Rightarrow q' = (s_{0;..\leftrightarrow s-1}; [\text{“eog”}]: A)$。口头上，新问题 P 表示: 如果 A 中的串一直到索引 n 之前都看上去像 s, 那么原问题就变成 s 是否属于 A（伴随哨符的适当调整）。该精化的证明使用 E 为非空束的事实，但并不需要 E 是一个非空串的束的事实 。下面是剩余问题的精化。

$$P \Leftarrow \textbf{if } s_n : A_n \textbf{ then } A := (\S a: A \cdot a_n = s_n).\ n := n+1.\ P$$
$$\textbf{else } q := [\text{“eog”}]: A_n \wedge s_n = [\text{“eos”}] \textbf{ fi}$$

从 P 可以知道 A 中的所有串一直到索引 n 之前都与 s 相同。如果在 A 中有串在索引 n 上与 s 一致，那么将 A 减为只有那些串，并前进一个索引。如果没有，要么不再有侯选，这时需要将 q 赋值为⊥，要么现在已到达 s 的结尾并同时到达一个侯选的结尾，此时应该将 q 赋值为 T。为了重点说明现在的话题——数据转换，这里忽略这些精化的证明。

现在将 A 由变量 b 替换，其值为一个文本串。用[“E”]表示假设不能出现在指定串 s 中的束 E。（在语法分析理论中，“E”被称为“非终结符”。）例如，文本串

[“if”]; [“x”]; [“then”]; [“E”]; [“else”]; [“E”]; [“fi”]

表示以下文本串的束：

[“if”]; [“x”]; [“then”]; E; [“else”]; E; [“fi”]

其数据转换可非形式地描述为：

A = （将 b 中所有项[“E”]的出现替换为束 E）

令 Q 为转换 P 的结果。转换的结果如下：

$$q' = (s_{0;..\leftrightarrow s-1} : E) \Leftarrow b := [\text{“E”}]; [\text{“eog”}].\ n := 0.\ Q$$

$$Q \Leftarrow \textbf{if } s_n = b_n \textbf{ then } n := n+1.\ Q$$
$$\textbf{else if } b_n = [\text{“E”}] \wedge s_n = [\text{“x”}] \textbf{ then } b := b_{0;..n}; [\text{“x”}]; b_{n+1;..\leftrightarrow b}.\ n := n+1.\ Q$$
$$\textbf{else if } b_n = [\text{“E”}] \wedge s_n = [\text{“if”}]$$
$$\textbf{then } b := b_{0;..n}; [\text{“if”}]; [\text{“E”}]; [\text{“then”}]; [\text{“E”}]; [\text{“else”}]; [\text{“E”}]; [\text{“fi”}]; b_{n+1;..\leftrightarrow b}.$$
$$n := n+1.\ Q$$
$$\textbf{else } q := b_n = [\text{“eog”}] \wedge s_n = [\text{“eos”}] \textbf{ fi fi fi}$$

这里可以做一点小小的提高，将 E 的表示从[“E”]改变成[“x”]，这样的话一种情况就消失了，得到：

$q' = (s_{0;..\leftrightarrow s-1} : E) \Leftarrow b := [\text{“x”}]; [\text{“eog”}].\ n := 0.\ Q$

$Q \Leftarrow$ **if** $s_n = b_n$ **then** $n := n+1.\ Q$
else if $b_n = [\text{“x”}] \wedge s_n = [\text{“if”}]$
then $b := b_{0;..n}; [\text{“if”}]; [\text{“x”}]; [\text{“then”}]; [\text{“x”}]; [\text{“else”}]; [\text{“x”}]; [\text{“fi”}]; b_{n+1;..\leftrightarrow b}.$
$n := n+1.\ Q$
else $q := b_n = [\text{“eog”}] \wedge s_n = [\text{“eos”}]$ **fi fi**

下一个提高在于注意到其实不需要 b 的初始部分，因为它和 s 的初始部分是相等的。因此再次进行转换，使用以下转换式将 b 替换为 c：

$$b = s_{0;..n}; c$$

令 R 为转换 Q 的结果。转换的结果如下：

$q' = (s_{0;..\leftrightarrow s-1} : E) \Leftarrow c := [\text{“x”}]; [\text{“eog”}].\ n := 0.\ R$

$R \Leftarrow$ **if** $s_n = c_0$ **then** $c := c_{1:..\leftrightarrow c}.\ n := n+1.\ R$
else if $c_0 = [\text{“x”}] \wedge s_n = [\text{“if”}]$
then $c := [\text{“x”}]; [\text{“then”}]; [\text{“x”}]; [\text{“else”}]; [\text{“x”}]; [\text{“fi”}] c.\ n := n+1.\ R$
else $q := c_0 = [\text{“eog”}] \wedge s_n = [\text{“eos”}]$ **fi fi**

----------语法分析结束

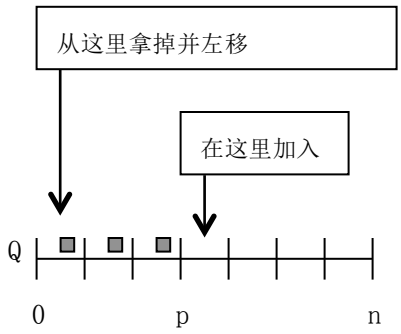
7.2.3 有界队列

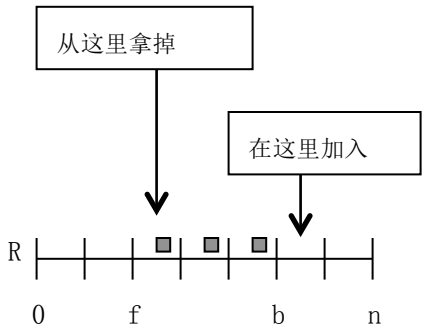
下一个例子，练习 421，要求转换一个有界队列使之得到一个与原始实现不符的时间界。一个有界队列是一个只有有限个数的位置用以存放项的队列。令界限为一个正的自然数 n，令 $Q : [n * X]$ 和 $p : nat$ 为实现者变量。则原始的实现如下：

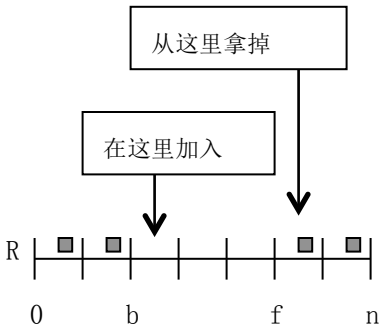
mkemptyq = $p := 0$
isemptyq = $p = 0$
isfullq = $p = n$
join x = $Qp := x.p := p + 1$
leave = **for** $i := 1;..p$ **do** $Q(i-1) := Qi$ **od**. $p := p-1$
front = $Q0$

该理论的使用者需要在使用 *join* 前测试 ¬ *isfullq*，在使用 *leave* 或 *front* 前测试 ¬ *isemptyq*。

在队列的尾部的位置 p 加入一个新的项将花费时间零（递归时间度量）。最头部的项总是在位置 0 立即得到。不幸的是，将头部的项删除将花费 $p-1$ 的时间将后面剩余的项前移一个位置。对队列进行转换使得所有的运算都是立即的。变量 Q 和 p 将被 $R : [n * X]$ 和 $f, b : 0;..n$ 代替，f 和 b 代表当前的头部和尾部。







其主要思想是 b 和 f 是沿着表循环移动的；当 f 在 b 的左边时，队列中的项在它们之间；当 b 在 f 的左边时，队列的项在它们的外侧两边。数据转换 D 如下：

$$0 \le p = b - f < n \wedge Q[0;..p] = R[f;..b]$$
$$\vee\ 0 < p = n - f + b \le n \wedge Q[0;..p] = R[(f;..n);(0;..b)]$$

现在进行转换，首先是 *mkemptyq*：

$$\forall Q, p \cdot D \Rightarrow \exists Q', p' \cdot D' \wedge p' = 0 \wedge Q' = Q \qquad \text{几个省略的步骤}$$
$$= \quad f' = b'$$
$$\Leftarrow \quad f := 0.b := 0$$

下面转换 *isemptyq*。尽管 *isemptyq* 正好是二元值，可以被解释成一个未实现的规范，但它的目的（类似 *front*，它不是二元值）是告诉用户队列的状态。这里不转换任意的表达式，而转换实现的规范（通常是程序）。因此假设 *c* 是一个用户变量，转换 *c*:= *isemptyq*。

$$\forall Q, p \cdot D \Rightarrow \exists Q', p' \cdot D' \wedge c' = (p = 0) \wedge p' = p \wedge Q' = Q \qquad \text{几个省略的步骤}$$

$$
\begin{array}{ll}
= & f<b \wedge f'<b' \wedge b-f=b'-f' \wedge R[f;..b]=R'[f';..b'] \wedge \neg c' \\
\vee & f<b \wedge f'>b' \wedge b-f=n+b'-f' \wedge R[f;..b]=R'[(f';..n);(0;..b')] \wedge \neg c' \\
\vee & f>b \wedge f'<b' \wedge n+b-f=b'-f' \wedge R[(f;..n);(0;..b)]=R'[f';..b'] \wedge \neg c' \\
\vee & f>b \wedge f'>b' \wedge b-f=b'-f' \wedge R[(f;..n);(0;..b)]=R'[(f';..n);(0;..b')] \wedge \neg c'
\end{array}
$$

一开始 R 可能在“中间”或“两边”的构造，而最后的 R' 还是如此，因此有四个析取式。但可疑的是，在每种情况下都有 $\neg c'$。这是因为丢失了 $f = b$！因此这个转换后的运算是不可实现的。转换通过这种方式告诉我们新的变量没有足够的信息来回答队列是否为空的问题。问题发生在 $f = b$ 的情况，因为它可以是空队列也可以是满队列。一个解决的办法是增加一个新的变量 *m: bin* 来说明是在“中间”的模式还是在“两边”的模式。对转换 D 修改如下：

$$
\begin{array}{l}
m \wedge 0 \le p = b - f < n \wedge Q[0;..p] = R[f;..b] \\
\vee \neg m \wedge 0 < p = n - f + b \le n \wedge Q[0;..p] = R[(f;..n);(0;..b)]
\end{array}
$$

现在重新转换 *mkemptyq*。

$$
\begin{array}{ll}
 & \forall Q, p \cdot D \Rightarrow \exists Q', p' \cdot D' \wedge p' = 0 \wedge Q' = Q \qquad \text{几个省略的步骤} \\
= & m' \wedge f' = b' \\
\Leftarrow & m := \mathsf{T}.\ f := 0.b := 0
\end{array}
$$

下面转换 *c*:= *isemptyq*。

$$
\begin{array}{ll}
 & \forall Q, p \cdot D \Rightarrow \exists Q', p' \cdot D' \wedge c'=(p=0) \wedge p'=p \wedge Q'=Q \qquad \text{几个省略的步骤} \\
= & m \wedge f<b \wedge m' \wedge f'<b' \wedge b-f=b'-f' \wedge R[f;..b]=R'[f';..b'] \wedge \neg c' \\
\vee & m \wedge f<b \wedge \neg m' \wedge f'>b' \wedge b-f=n+b'-f' \\
 & \wedge R[f;..b]=R'[(f';..n);(0;..b')] \wedge \neg c' \\
\vee & \neg m \wedge f>b \wedge m' \wedge f'<b' \wedge n+b-f=b'-f' \\
 & \wedge R[(f;..n);(0;..b)]=R'[f';..b'] \wedge \neg c' \\
\vee & \neg m \wedge f>b \wedge \neg m' \wedge f'>b' \wedge b-f=b'-f' \\
 & \wedge R[(f;..n);(0;..b)]=R'[(f';..n);(0;..b')] \wedge \neg c' \\
\vee & m \wedge f=b \wedge m' \wedge f'=b' \wedge c' \\
\vee & \neg m \wedge f=b \wedge \neg m' \wedge f'=b' \wedge R[(f;..n);(0;..b)]=R'[(f';..n);(0;..b')] \wedge \neg c' \\
\Leftarrow & c' = (m \wedge f=b) \wedge f'=f \wedge b'=b \wedge R'=R \\
= & c := m \wedge f=b
\end{array}
$$

该转换后的运算使得可以在 R 内部旋转队列，但我们不这么做。对于其他的数据结构，在一个运算之间重新组织数据结构有时是个好策略，并且数据转换总能说明什么样的重组是可能的。每一个剩余的转换提供一样的机会，但没理由旋转队列，于是每次都不这么做。

下面转换 *c*:= *isemptyq*，*join x* 和 *leave*。

$$
\begin{array}{ll}
 & \forall Q, p \cdot D \Rightarrow \exists Q', p' \cdot D' \wedge c'=(p=n) \wedge p'=p \wedge Q'=Q \qquad \text{几个省略的步骤} \\
\Leftarrow & c := \neg m \wedge f=b
\end{array}
$$

$$\forall Q, p \cdot D \Rightarrow \exists Q', p' \cdot D' \wedge Q' = Q[0;..p] + [x] + Q[p+1;..n] \wedge p' = p+1$$

几个省略的步骤

$\Leftarrow$ $Rb := x.$ **if** $b+1 = n$ **then** $b := 0.\ m := \bot$ **else** $b := b+1$ **fi**

$$\forall Q, p \cdot D \Rightarrow \exists Q', p' \cdot D' \wedge Q' = Q[(1;..p)(p-1;..n)] \wedge p' = p-1$$

几个省略的步骤

$\Leftarrow$ **if** $f+1 = n$ **then** $f := 0.\ m := \top$ **else** $f := f+1$ **fi**

最后转换 $x := front$，其中 x 是与项类型相同的一个用户变量。

$$\forall Q, p \cdot D \Rightarrow \exists Q', p' \cdot D' \wedge x' = Q0 \wedge p' = p \wedge Q' = Q$$

几个省略的步骤

$\Leftarrow$ $x := Rf$

---有界队列结束

7.2.4 可靠性与完备性 可选节

数据转换是可靠的，用户应该感觉不到转换的发生，这也是它的设计标准。但（从用户的视角），也可能发现两个规范的行为是相同的，但却没有数据转换能将一个转换到另一个。在这种情况下，数据转换是不完备的。

练习 422 说明了这个问题。用户变量是 i，实现者是变量 j，都有类型 0，1，2。运算是：

initialiaze = $i' = 0$

step = **if** $j > 0$ **then** $i := i+1.\ j := j-1$ **else** *ok* **fi**

用户可以看见 i 但不能看见 j。用户可以运算 *initialize*，它将 i 初始化为 0，将 j 初始化为 3 个值中的任意一个。用户可以重复执行 *step* 并观察 i 增加 0 次或 1 次或 2 次然后停止增加，它有效地告诉用户 j 的初始值是什么。

如果这是一个实际的问题，通过发现 *initialize* 可以精化，从而解决不确定性。例如：

$$initialize \Leftarrow i := 0.\ j := 0$$

这样可以通过转换 *initialize* 和 *step* 消除 j，把 j 用无来代替。转换式是 $j = 0$。该转换将 *initialize* 的实现转换如下：

$$\forall j \cdot j = 0 \Rightarrow \exists j' \cdot j' = 0 \wedge i' = j' = 0$$

= $i := 0$

step 的转换如下：

$$\forall j \cdot j = 0 \Rightarrow \exists j' \cdot j' = 0 \wedge$$ **if** $j > 0$ **then** $i := i+1.\ j := j-1$ **else** *ok* **fi**

= *ok*

如果这是个实际的问题，那么就已完成了。但理论的问题是，保持非确定性，将 j 替换为一个二元变量 b。因此：

initialize 转换为 $i' = 0$

step 转换为 **if** $b \wedge i < 2$ **then** $i' = i+1$ **else** *ok* **fi**

现在转换后的 *initialize* 要么 b 从 $\top$ 开始，表示 i 会增加，要么从 $\bot$ 开始，表示 i 不会增加。

每次对转换后的 *step* 执行测试 b 以判断是否要增加 i，并测试 $i < 2$ 来保证增加后的 i 不超过 2 。如果 i 增加了， b 又会被赋予其中一个值。用户会看到 i 从 0 开始，增加了 0 次或 1 次或 2 次然后停止增加，就像原始的规范一样。非确定性会保持。但没有包含 i， j 和 b 的转换式能完成任务。那是因为 j 的初始值提供了 3 个不同的行为，但二元变量 b 的初始值不能区分这三个行为。

----------可靠性与完备性结束

----------数据转换结束

----------理论设计和实现结束

第八章 并发

并发(Concurrency)，也叫并行(parallelism)，意味着两个或更多的活动同时发生。在一些其它的书本上，词语“并发”和“并行”用来表示以不指定的顺序发生的活动，或它们可以由以交互顺序发生的更小的活动组成。但在本书中，它们表示同时发生多于一个的活动。

8.0 独立组合

规范 P 与 Q 的独立组合，记作 $P \| Q$（读作“P 与 Q 并行”），定义为：计算机的行为同时(并行地)满足 P 和 Q。$\|$的运算对象称作进程（processes)。

当定义 P 和 Q 的相关组合时，要求 P 和 Q 有相同的状态变量，这样就可以使 P 的终结状态与 Q 的开始状态相等。对于独立组合 $P \| Q$，要求 P 和 Q 具有完全不同的状态变量，且 $P \| Q$ 的状态变量是那些既属于 P 又属于 Q 的状态变量。如果忽略时间和空间，独立组合是合取：

$$P \| Q \quad = \quad P \wedge Q$$

当决定创造一个独立组合时，需要决定如何划分变量。给定规范 S，如果想这样精化它 $S \Leftarrow P \| Q$，则必须决定 S 中的哪些变量属于 P，哪些属于 Q。例如，下面的规范包含变量 x，y 和 z：

$$x' = x + 1 \wedge y' = y + 2 \wedge z' = z$$

如果划分变量，可以使用下面的独立组合来精化：

$$x := x + 1 \| y := y + 2$$

为了使对 x 的赋值有意义，x 应属于左边的进程，类似地，y 应属于右边的进程。而 z 属于哪个进程都没关系；都有

$$x := x + 1 \| y := y + 2 \quad = \quad x' = x + 1 \wedge y' = y + 2 \wedge z' = z$$

提出一个独立组合的人有义务决定如何划分变量。如果有一个独立组合，但写这个组合的人没有说明变量是如何划分的，那么必须决定一个有意义的划分。这里有一个通常有用的方法：如果有 x' 或 $x :=$ 出现在一个进程的规范中，则 x 属于这个进程。如果没有任何 x' 或 $x :=$ 出现，则 x 可以属于划分后的任一部分。这个划分方法不适用于当 x' 或 $x :=$ 同时出现在两个进程规范中。

在下面的例子

$$x := y \| y := x$$

中，x 属于左边的进程，y 属于右边的进程，z 属于任何进程。在左边的进程中，y 出现了，但 y' 或 $y :=$ 都没出现，因此 y 是一个状态常量，而不是状态变量，在左边的进程中。类似地，x 在右边的进程中是一个状态常量。结果是：

$$x := y \| y := x \quad = \quad x' = y \ \wedge \ y' = x \ \wedge \ z' = z$$

变量 x 和 y 交换了值，很显然没有使用临时变量。事实上，进程的实现中将对属于另一个进程

的变量的初始值做一个私有的拷贝，如果另一个进程中包含对该变量的赋值的话。

使用二元变量 b 和整数变量 x，

$b := x=x \parallel x := x+1$ 用=代替 $x=x$

$= \quad b := \mathsf{T} \parallel x := x+1$

根据第一行，似乎可在估量进程左边两个 x 的过程中间，计算右边的 x 的增值，这样就可使 b 赋值为⊥。甚至不能保证 $x = x$，这将是一个数学灾难。根据最后一行，上述情况不会发生，在左边进程中 x 的两次出现都表示变量 x 的初始值。可以根据相等性的自反和透传公理，用 T 代替 $x=x$。

在第 4 章定义的相关组合中，变量的中间值对相关组合是局部的；它们被量词 $\exists x'', y'', \ldots$ 隐藏起来了。如果一个进程是相关组合，其他的进程是看不见它的中间值的。例如：

$$(x := x+1.x := x-1) \| y := x$$

$$= \quad ok \| y := x$$

$$= \quad y := x$$

在第一行，似乎右边的进程在左边的进程对 x 的两次赋值之间对 x 取值是可能的。但根据最后一行，这并不会发生。右边进程中的 x 指的是 x 的初始值。下一章将介绍进程间的交互变量和通信通道，使得它们能看到对方变量的中间值，但本章只讨论非交互的进程。

在前面的例子中，我们将 $(x := x+1.x := x-1)$ 用 ok 来替换。当然如果 x 是状态变量之一的话也可以进行反方向的替换。尽管 x 是下面的组合的一个变量：

$$ok \| x := 3$$

但由于在右边的进程中有对它的赋值，因此它不是左边进程 ok 的变量。因此不能将该组合等同于

$$(x := x+1.x := x-1) \| x := 3$$

有时共享存储变量的需要是由于不好的程序结构导致的。例如，假设有下面的两个进程：

$$(x := x+y.x := x \times y) \| (y := x-y.y := x/y)$$

第一个进程对 x 修改了两次，第二个进程对 y 修改了两次。但假设需要每个进程的第二个赋值使用的是第一次赋值以后的 x 和 y 的值。那么似乎不仅需要共享存储变量，而且需要两个进程在中间点同步，使快的进程等待慢的进程，然后才允许两个进程都从修改后的新的 x 和 y 的值继续进行。事实上，如果写成下式，就可以既不需要共享存储变量也不需要同步：

$$(x := x+y \| y := x-y).(x := x \times y \| y := x/y)$$

到目前为止，独立组合还只是合取，并且没必要为合取再引入一个运算符 $\|$。但现在要考虑时间了。时间变量不是需要划分的变量，它属于两个进程。在 $P \| Q$ 中，P 和 Q 都从时间 t 开始执行，但可能在不同的时间结束。组合 $P \| Q$ 执行结束的时间是 P 和 Q 都执行结束的时间。

考虑时间，独立组合定义如下：

$$P \| Q \;=\; \exists\, t_P, t_Q \cdot \langle t' \to P \rangle t_P \wedge \langle t' \to Q \rangle t_Q \wedge t' = max\ t_P\ t_Q$$

$$= \exists\, t_P, t_Q \cdot \quad (\text{将 } P \text{ 中的 } t' \text{ 置换为 } t_P)$$

$$\wedge\ (\text{将 } Q \text{ 中的 } t' \text{ 置换为 } t_Q)$$

$$\wedge\ t' = max\ t_P\ t_Q$$

8.0.0 独立组合定律

令 x 和 y 为两个不同的状态变量，令 e, f, b 为前置状态的表达式， 令 P, Q, R 和 S 为规范。那么

$(x := e \| y := f).\ P$ = (将 P 中 x 置换为 e 且独立地将 y 置换为 f)	独立置换
$P \| Q = Q \| P$	对称性
$P \| (Q \| R) = (P \| Q) \| R$	结合性
$P \| t' = t = P = t' = t \| P$	恒等性
$P \| Q \vee R = (P \| Q) \vee (P \| R)$	分配性
$P \|$ **if** b **then** Q **else** R **fi** = **if** b **then** $P \| Q$ **else** $P \| R$ **fi**	分配性
if b **then** $P \| Q$ **else** $R \| S$ **fi** = **if** b **then** P **else** R **fi** $\|$ **if** b **then** Q **else** S **fi**	分配性

结合定律说明可以将任意数量的进程组合在一起而不必担心它们是如何分组的。下面是一个使用置换定律的例子：

$$(x := x+y \,\|\, y := x \times y).\ z' = x - y \;=\; z' = (x+y) - (x \times y)$$

注意，每次仅对所有原始出现的相关变量进行置换。这个定律将早先介绍的置换定律从一个变量推广到两个变量。类似地，也可进一步推广到任意多个变量的置换。

逐步精化定律适用于独立组合：

如果 $A \Leftarrow B \| C$ 和 $B \Leftarrow D$ 和 $C \Leftarrow E$ 是定理，那么 $A \Leftarrow D \| E$ 是定理。

部分精化定律也适用于独立组合：

如果 $A \Leftarrow B \| C$ 和 $D \Leftarrow E \| F$ 是定理，那么 $A \wedge D \Leftarrow B \wedge E \,\|\, C \wedge F$ 是定理。

----------独立组合定律结束

8.0.1 表并发

通过划分变量，定义了独立组合。为了得到更加细粒度的并发，可以将相同的思想扩充到表变量的单个项上。在第五章中定义了对表项的赋值为：

$$Li := e \;=\; L'i = e \wedge (\forall j \cdot j \neq i \Rightarrow L'j = Lj) \wedge x' = x \wedge y' = y \wedge \ldots$$

说明不仅被赋值的项获得正确的最终值，而且其它项和其它变量的值保持不变。对于独立组合，必须只说明划分的一边的项和变量的终结值。

练习 167 是表并发的一个代表例子：在一个表中寻找最大项。表中最大项可用量词 *MAX* 容易地表示，不过这里假设 *MAX* 尚未实现。解决这一问题最容易且最简单的方法可能是这样的函数，其参数（运算符的运算对象）总可以并行计算。为了使用平行运算符，这里介绍一种

命令式的解。令 L 是欲查找最大项的表。如果 L 是一个空表，那么它的最大项是 $-\infty$；现假设 L 非空。进一步假设 L 是一变量，在找到其最大项之后，其值可以忽略（稍后将去掉这个假设）。规范为：

$$L'0 = MAX\ \ L \wedge t' = t+log(\#L)$$

最后，表 L 的 0 项将是所有初始项中的最大项。第一步，先将对非空表求最大项，推广到对一个表的非空段求最大项。因此定义：

$$findmax\ =\ \langle i, j \longrightarrow i<j\ \Rightarrow\ \ L'\,i = MAX\,(L\,[i;..j]) \wedge t' = t+log(j-i)\rangle$$

这样，原规范就是 $findmax\ 0\ (\#L)$。对它作如下精化：

$$findmax\ i\ j\ \Leftarrow\ \textbf{if}\ \ j-i=1\ \ \textbf{then}\ ok$$
$$\textbf{else}\ t:=t+1.\ (findmax\ i\ (div\ (i+j)\ 2)\ \|\ findmax\ (div\ (i+j)\ 2)\ \ j).$$
$$L\ i:=max\ (L\ i)\ (L\ (div\ (i+j)\ 2))\ \textbf{fi}$$

如果 $j-i\ =\ 1$，那么相应段中只有一项；最大项（仅有的那个项）就在索引 i 处，什么都不需改变。在其它情况下，一个段包含有多个项；将该段一分为二，将各子段中的最大项放置在子段的头一个位置上。在这个平行组合中，两个进程 $findmax\ i\ (div\ (i+j)\ 2)$和 $findmax\ (div\ (i+j)\ 2)\ j$ 分别作用在表的两个不相交的段上。然后再将这两个最大项中的最大项放置到整个段的开始位置，问题得解。所用的递归执行时间是 $ceil\ (log\ (j-i))$，与二叉树搜索所用时间完全一样，其程序也非常相似。

如果表 L 必须保持不变，可以用一个与 L 具有相同类型的新表 M 来存储部分结果。重新定义：

$$findmax\ =\ \langle i, j \longrightarrow i<j \Rightarrow M'\,i = MAX\,(L\,[i;..j]) \wedge t' = t+log(j-i)\rangle$$

而在程序中将 ok 改成 $M\ i:=L\ i$，并且将最后一个赋值语句改成：

$$M\ i:=max\ (M\ i)\ (M\ (div\ (i+j)\ 2))$$

---表并发结束

---独立组合结束

8.1 顺序到并行的转换

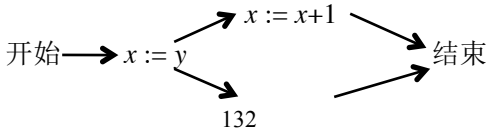
本小节的目标是将不含并发的程序转换成含有并发的程序。忽略时间，用下面这个简单例子说明上述思想。

$$\begin{aligned} & x := y.\ \ x := x+1.\ \ z := y \\ = \quad & x := y.\ \ (x := x+1 \parallel z := y) \\ = \quad & (x := y.\ \ x := x+1) \parallel z := y \end{aligned}$$

为上面第一行编写程序，其执行过程描述如下：

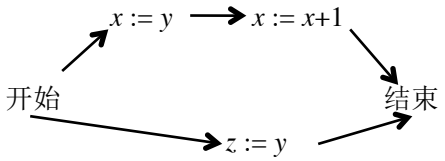
开始 $\longrightarrow$ $x := y$ $\longrightarrow$ $x := x+1$ $\longrightarrow$ $z := y$ $\longrightarrow$ 结束

开始的两个赋值操作不能并行，但最后的两个赋值操作却可以并行，将其转换成程序。执行过程现在可描述为：



$z := y$

现在第一和最后的赋值操作按顺序紧接着执行；它们也可以并行执行。执行如下所示：



当两个程序按顺序出现时，只要没有在一个程序中对在另一个程序中被赋值的变量进行赋值，也没有在第一个程序中被赋值的变量出现在第二个程序中，就可以将它们并行执行；任何出现在第一个程序中的变量的初始值必须进行拷贝，并用于在第二个程序中进行赋值。只要两个程序按顺序出现，每个程序中被赋予值的变量不出现在另一个程序中，那么不需任何初始值拷贝就可以并行执行。该转换不改变计算的结果，但可能减少时间，这也是这么做的原因。

获取并行的程序转换一般可以通过编译器自动实施。有时它只能被编译器来实施，因为其结果不能被表示为一个源程序。

8.1.0 缓冲区

考查两个程序，*produce* 和 *consume*，它们之间唯一的公共变量是 *b*。*produce* 给 *b* 赋值，而 *consume* 使用 *b* 的值。

produce = ... *b*:= *e*.....

consume = ...*x*:= *b*.....

这两个程序交替执行，一直循环往复下去。

$$control = produce.consume.control$$

用 *P* 表示 *produce*，用 *C* 表示 *consume*，那么执行过程如下所示：

$$P \longrightarrow C \longrightarrow P \longrightarrow C \longrightarrow P \longrightarrow C \longrightarrow P \longrightarrow C \longrightarrow$$

许多程序中都有这种生产者与消费者的成分。变量 *b* 称作一个缓冲区；它可能是一个大的数据结构。其思想是 *produce* 和 *consume* 都会消耗时间，如果并行执行它们将会节约时间。然而，这里不能将它们并行因为第一个程序对 b 赋值而第二个程序使用了 b。因此可以将循环展开一次。

$control = produce.newcontrol$

$newcontrol = consume.produce.newcontrol$

而 *newcontrol* 可以被转换成

$$newcontrol = (consume \| produce).newcontrol$$

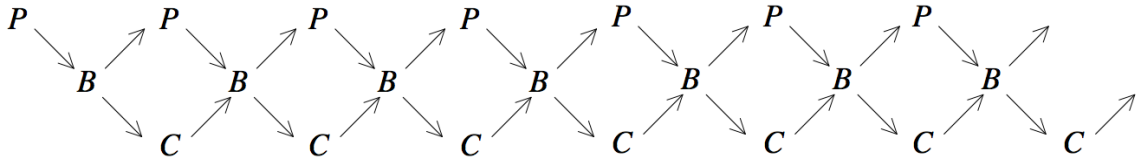
在转换后的程序中，编译器必须获得 b 的初始值的一个拷贝提供给 *consume* 所用。或者，可以在源代码级用变量 *c* 进行这个拷贝，如下所示：

produce = ... *b*:= *e*.....

consume = ...*x*:=*c*.....

$$control = produce.newcontrol$$
$$newcontrol = c:=b.(consume \| produce).newcontrol$$

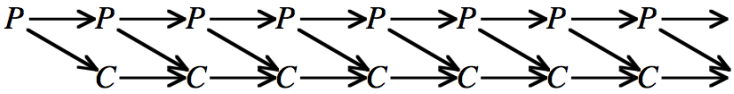
用 B 表示赋值 $c := b$，那么执行过程变为：



如果 *produce* 和 *consume* 中有一个执行时间始终大于另一个，那么这是一个最佳情况。如果它们的执行时间是变化的，有些循环中 *produce* 执行的时间长，而有时 *consume* 时间长，那么我们应当作些改进，将缓冲区分割成一个无穷表。用自然数变量 w 记录 *produce* 向缓冲区写的次数，用自然数变量 r 记录 *consume* 从缓冲区读的次数。将 w 和 r 都初始化成 0。那么有：

$$produce = ...bw:=e.w:=w+1.....$$
$$consume = ...x:=br.r:=r+1.....$$
$$control = produce.consume.control$$

如果 $w \neq r$，那么 *produce* 和 *consume* 可以并行执行， 执行过程如下：



当 *produce* 的执行快于 *consume* 时，它的执行可以领先 *consume* 执行的任意多少。当 *consume* 的执行快于 *produce* 时，它的执行可以赶上但不能超过 *produce* 的执行；当 $w=r$ 时，先 P 后 C 的顺序就可以保持。并行执行的机会可以被程序语言编译器自动发现，或者通过某种适当的表示告诉编译器 。但在这个例子中，不使用额外的交互结构就无法用源程序的形式表达结果所得的执行模式（第 9 章）。

如果缓冲区是长度为 n 的有限表，可用循环方式执行上述过程：

$$produce = ...bw:=e.w:=mod(w+1)n.....$$
$$consume = ...x:=br.r:=\mathrm{mod}(r+1)n.....$$
$$control = produce.consume.control$$

同前面一样，*consume* 的执行不能超过 *produce*，因为当缓冲区为空时 $w=r$。不过此时 *produce* 的执行领先 *consume* 的次数不能超过 n, 因为当缓冲区已经满时 $w=r$ 也成立。

----------缓冲区结束

计算机程序在不考虑并发时，有时比较容易开发和证明。开发并行性的责任可以交给一个聪明的编译器。同步是在所有并行的机会均被找出后剩余的顺序执行。

8.1.1 插入排序

练习 203 要求写一个给表排序的程序，排序时间界限是表长度的平方。下面给出一个解答。

令表为 L，且定义：

$$sort \;=\; \langle n \to \forall i, j : 0,..n \cdot i \le j \Rightarrow Li \le Lj \rangle$$

这样 *sort n* 表示将 L 排序到索引 n 处。其规范是：

$$(L' \text{ 是 } L \text{ 的一个排列}) \wedge sort'\,(\#L) \wedge t' \le t + (\#L)^2$$

第一个合取式是用非形式化描述的，并且采用下式对 L 进行的所有修改可保证它被满足：

$$swap\ i\ j \;=\; Li := Lj \,\|\, Lj := Li$$

忽略最后一个合取式；程序转换将提供一个执行时间呈线性关系的解。因为 *sort* 0 是一个定理，所以第二个合取式等价于 $sort\ 0 \Rightarrow sort'\,(\#L)$。

$$sort\ 0 \Rightarrow sort'\,(\#L) \;\Leftarrow\; \textbf{for}\ n := 0;..\#L\ \textbf{do}\ sort\ n \Rightarrow sort'\,(n+1)$$

为了求解 $sort\ n \Rightarrow sort'\,(n+1)$，参考一个例子：

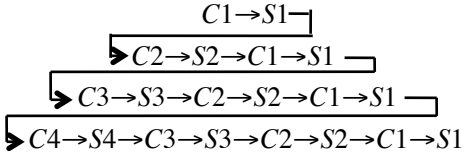
［$L\,0$； $L\,1$； $L\,2$； $L\,3$； $L\,4$］

0 1 2 3 4 5

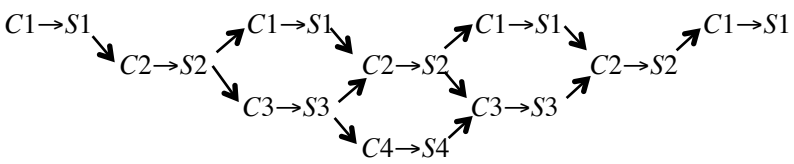
$$sort\ n \Rightarrow sort'\,(n+1) \;\Leftarrow\; \textbf{if}\ n = 0\ \textbf{then}\ ok$$
$$\textbf{else if}\ L\,(n-1) \le L\ n\ \textbf{then}\ ok$$
$$\textbf{else}\ swap\ (n-1)\ n.\ sort(n-1) \Rightarrow sort'\ n\ \textbf{fi fi}$$

如果将 $sort\ n \Rightarrow sort'\ (n+1)$看成是具有参数 n 的一个过程就可以结束了；最后一个规范 $sort(n-1) \Rightarrow sort'\ n$ 就可看成是以 $n-1$ 为实参调用该过程的结果。或者，可以令 n 为一个变量代替 **for**-循环的索引，且在最后那个调用之前将它减 1。现在暂时不考虑细节，转而检查其有无并行执行的可能性。

令 $C\ n$ 代表比较式 $L\ (n-1) \le L\ n$ 且令 $S\ n$ 代表 $swap\ (n-1)\ n$。假设$\#L$=5，最坏的顺序执行情况如下图所示：



如果 i 和 j 相差超过 1，那么 $S\ i$ 和 $S\ j$ 可以并行执行。在相同的条件下，$S\ i$ 和 $C\ i$ 可以并行执行和计算。当然，任意两个表达式诸如 $C\ i$ 和 $C\ j$ 总可以并行地计算。执行过程变为：



为了便于书写一个平方时间的顺序的排序，可以提供一个巧妙的编译器，从而获得一个线性时间的并行排序。

----------插入排序结束

8.1.2 哲学家就餐问题

练习 445：五个哲学家坐在圆桌前。在桌子中间有一碗无限供量的面条。在每对相邻的哲学家之间有一只筷子。当一个哲学家饥饿时，他就去拿他左边的筷子和右边的筷子，因为吃面条需要两只筷子。如果相邻的哲学家在使用筷子，那么就至少有一只筷子不可用，那么饥饿的哲学家必须等待直到筷子再次可用。当两只筷子都可用时，哲学家就可以吃一会儿面条，然后放下两只筷子，继续思考，直到再饥饿。问题是写一个程序模拟哲学家的生活。有可能发生这样的情况，五个哲学家同时感到饥饿，于是都拿起左边的筷子，但发现右边的筷子不在，于是等待右边的筷子的同时一直拿着左边的筷子。这样就发生了死锁(deadlock)，写程序时必须避免这种情况。如果程序任一时间只允许一个哲学家饥饿，那么就不会发生死锁，但也不会有并发了。

这是一个很多教科书中用于说明并发程序设计的标准问题。经常还会增加一个标准：每个哲学家可以无限次地吃面条。但不需考虑这一点。现在从不包含并发和没死锁的情况：每次只有一个哲学家饥饿的版本开始。哲学家编号为 0 到 4。筷子也同样编号，那么哲学家 i 的筷子编号是 i 和 $i+1$（练习中所有的加法都对 5 取模）。

$life = (P0 \vee P1 \vee P2 \vee P3 \vee P4).life$

$Pi = up\ i.\ up(i{+}1).\ eat\ i.\ down\ i.\ down(i{+}1)$

$up\ i = chopstick\ i := \mathsf{T}$

$down\ i = chopstick\ i := \bot$

$eat\ i = \ldots.\ chopstick\ i\ \ldots\ldots chopstick(i{+}1)\ldots..$

以上定义说明 *life* 是 Pi 动作的完全任意的序列（任选一个，然后重复），而 Pi 动作指哲学家 i 拿起左边的筷子，再拿起右边的筷子，再吃面条，再放下左边的筷子，再放下右边的筷子。为了将这些定义变成程序，需要决定在每次迭代时如何选择 Pi（这里就符合了每个哲学家可以无限多次吃面条的标准）。除了要用两只筷子，如何定义 *eat i* 是不清楚的。（如果程序的意图是完成某个目标，我们可以去除变量 *chopstick*，而将 *eat i* 中的出现用 T 代替。但本程序要求描述一个活动，而吃面条要使用两只筷子。）

现在进行转换来获取并发性。

如果 $i \neq j$, (*up i*. *up j*)变成(*up i* $\|$ *up j*)。

如果 $i \neq j$, (*up i*. *down j*)变成(*up i* $\|$ *down j*)。

如果 $i \neq j$, (down *i*. *up j*)变成(*down i* $\|$ *up j*)。

如果 $i \neq j$, (*down i*. *down j*)变成(*down i* $\|$ *down j*)。

如果 $i \neq j \wedge i+1 \neq j$, (*eat i*. *up j*)变成(*eat i* $\|$ *up j*)。

如果 $i \neq j \wedge i \neq j+1$, (*up i*. *eat j*)变成(*up i* $\|$ *eat j*)。

如果 $i \neq j \wedge i+1 \neq j$, (*eat i*. *down j*)变成(*eat i* $\|$ *down j*)。

如果 $i \neq j \wedge i \neq j+1$, (*down i*. *eat j*)变成(*down i* $\|$ *eat j*)。

如果 $i \neq j \wedge i+1 \neq j \wedge i \neq j+1$, (*eat i*. *eat j*)变成(*eat i* $\|$ *eat j*)。

不同的筷子可以同时被拿起或放下。只要这根筷子没有用来吃面条，那么筷子的拿起和放下可

以和吃并行。并且，只要是不相邻的哲学家，可以同时吃面条。这些转换可以很容易地从 *up*，*down* 和 *eat* 的定义以及独立组合的定义得到。它们不都能立即应用于原程序，但只要有了转换，就可以进行进一步的转换。

在进行转换之前，是没有可能发生死锁的。以上的转换也没有产生这种可能性。其结果是最大程度的并行，但不会导致死锁。一个聪明的编译器可以从初始程序（没有并发的）进行转换。

在解决哲学家就餐问题时，时常会产生从太多并行开始的错误。

$$life = P0 \| P1 \| P2 \| P3 \| P4$$

$$Pi = (up\ i \| up(i+1)).eat\ i\ .(down\ i \| down(i+1)).Pi$$

$P0$ 不能和 $P1$ 并行，因为它们都使用和对 *chopstick* 1 赋值。如果程序要这样开始，就必须通过加入互斥和防死锁的功能来更正错误，这样就使问题变得困难。最好不要制造错误，那么互斥和防死锁的功能就不需要了。

----------哲学家就餐问题结束

----------顺序到并行的转换结束

----------并发结束

第九章 交互(Interaction)

前面已经讨论了使用状态变量的初始值和终结值来描述计算。一个状态变量的声明

$$\textbf{var}\, x : T \cdot S = \exists x, x' : T \cdot S$$

说明状态变量其实是两个数学变量，一个表示初始值一个表示终结值。在声名的作用域内，x 和 x' 是在规范 S 中使用的。在相关（顺序）组合中有中间变量，但这些中间变量局部于相关组合的定义。

$$P.Q = \exists x'', y'', \ldots \cdot \langle x', y', \ldots \rightarrow P \rangle\, x''\, y''\, \ldots \wedge \langle x, y, \ldots \rightarrow Q \rangle\, x''\, y''\, \ldots$$

考虑 $(P.Q) \| R$ 的情况。P 和 Q 的中间值被隐藏在相关组合中，因而对 R 不可见，因此不能用于进程的交互。

只有初始值和终结值可见的变量称为边界变量(boundary variable)，而在所有时候其值都可见的变量称为交互变量(interactive variable)。到目前为止，讨论的变量都是边界变量。现在要引入交互变量，其中间值对于并行进程是可见的。这些变量可用于对人和计算机之间、进程之间和计算的过程中的交互进行描述和推理。

9.0 交互变量

令 $\textbf{ivar}\, x : T \cdot S$ 声明 x 为作用域 S 内的类型为 T 的交互变量。它定义如下：

$$\textbf{ivar}\, x : T \cdot S = \exists x : time \rightarrow T \cdot S$$

其中 $time$ 是时间的定义域，为延伸的整数型或延伸的实数型。一个交互变量是一个时间的函数。变量 x 在时间 t 的值为 $x\, t$。

假设 a 和 b 为边界变量，x 和 y 为交互变量，t 为时间。对独立组合需要将所有的状态变量分开，也就是边界变量和交互变量需要分开。假设 a 和 x 属于 P，b 和 y 属于 Q。

$$\begin{aligned} P \| Q = \exists tP, tQ \cdot \quad & \langle t' \rightarrow P \rangle tP \wedge (\forall t'' \cdot tP \le t'' \le t' \Rightarrow xt'' = x(tP)) \\ & \wedge \langle t' \rightarrow Q \rangle tQ \wedge (\forall t'' \cdot tQ \le t'' \le t' \Rightarrow yt'' = y(tQ)) \\ & \wedge \; t' = max\, tP\, tQ \end{aligned}$$

新的部分说明当更短的进程结束时，它的交互变量的值在更长的进程完成的过程中保持不变。

使用前面段落中相同的进程和变量，进程 P 中的赋值语句 $x := a + b + x + y$ 将四个变量的值相加赋给 x。由于 a 和 x 是进程 P 的变量，它们的值是 P 中所赋给的最后的值，或者如果 P 中没有对它们赋值，则是初始值。由于 b 是 Q 中的边界变量，P 所看到的它的值不论 Q 中对它有无赋值，都是其初始值。而由于 y 是 Q 中的交互变量，P 所看到的它的值是 Q 对它赋予的最后值，或者如果 Q 没有对它赋值，则是其初始值，或者如果 Q 正在对它赋值，这时其值不可知。由于 x 是一个交互变量，其新的值对所有并行的进程是可见的。$a + b + x + y$ 这个表达式是被滥用的，因为 a 和 b 是数，而 x 和 y 是时间到数的函数；实际所赋的值应该是 $a + b + xt + yt$，但在上下文明确的情况下可以省略参数 t。类似地，用 x' 表示 xt'，用 x'' 表示 xt''。

ok 定义为边界变量和时间不改变。因此在前两段的进程 P 中，

$$ok = a' = a \wedge t' = t$$

由于 $t' = t$，所以没必要说明 $x' = x$，它等同于 $xt' = xt$。这里也不提 b 和 y，因为它们不是进程 P 的变量。

对交互变量的赋值不是瞬时的，因为随时间不同它的值不同。在一个边界变量为 a 和 b，交互变量为 x 和 y 的进程中，

$$x := e = a' = a \wedge b' = b \wedge x' = e \wedge (\forall t'' \cdot t \le t'' \le t' \Rightarrow y'' = y)$$
$$\wedge\, t' = t + \text{（计算和存储 } e \text{ 所需的时间）}$$

在对 x 的赋值过程中交互变量 y 的值保持不变。在赋值中没有对 x 的值进行说明。

如果希望的话，可以认为对边界变量的赋值是瞬间的。如果要计算它的时间，必须说在赋值过程中所有的交互变量没有改变。

相关组合隐藏了边界变量和时间变量的中间值，而交互变量的中间值是可见的。边界变量为 a 和 b，交互变量为 x 和 y，时间变量为 t，定义：

$$P.Q = \exists a'', b'', t'' \cdot \langle a', b', t' \to P \rangle\, a''\ b''\ t''\ \wedge \langle a, b, t \to Q \rangle\, a''\ b''\ t''$$

增加了交互变量后，大部分规范定律和精化定律仍然有效，但遗憾的是，置换定律不再有效。

如果进程 P 和 Q 是并行的，它们具有不同的变量。再次假设边界变量 a 和交互变量 x 属于进程 P，边界变量 b 和交互变量 y 属于进程 Q。在规范 P 中，输入为 a, b, xt 和对于 $t \le t'' \le t'$ 的 yt''。在规范 P 中，输出为 a' 和对于 $t \le t'' \le t'$ 的 xt''。当

$$\forall a, b, X, y, t \cdot \exists a', x, t' \cdot P \wedge t \le t' \wedge \forall t'' \cdot t \le t'' \le t' \vee xt'' = Xt''$$

时，规范 P 是可实现的。像以前一样，P 必须在非递减时间下可满足；新的部分说明在时间间隔 t 和 t' 之外，P 不能限制它的交互变量。可以不需知道一个进程规范的上下文来检查其可实现性；变量 b 和 y 只在全称量词的外部出现。

练习 448 是一个同样使用变量 $a, b, x, y,$ 和 t 的例子。假设时间是一个延伸的整数，每个赋值花费一个单位时间。

$$(x := 2.x := x + y.x := x + y) \parallel (y := 3.y := x + y)$$

x 是左部进程的变量，
y 是右部进程的变量。
我们将 a 用于左部进程，b 用于右部进程。

$$= (a' = a \wedge xt' = 2 \wedge t' = t + 1.a' = a \wedge xt' = xt + yt \wedge t' = t + 1.a' = a \wedge xt' = xt + yt \wedge t' = t + 1)$$
$$\parallel (b' = b \wedge yt' = 3 \wedge t' = t + 1.b' = b \wedge yt' = xt + yt \wedge t' = t + 1)$$
$$= (a' = a \wedge x(t+1) = 2 \wedge x(t+2) = x(t+1) + y(t+1) \wedge x(t+3) = x(t+2) + y(t+2) \wedge t' = t + 3)$$
$$\parallel (b' = b \wedge y(t+1) = 3 \wedge y(t+2) = x(t+1) + y(t+1) \wedge t' = t + 2)$$
$$= a' = a \wedge x(t+1) = 2 \wedge x(t+2) = x(t+1) + y(t+1) \wedge x(t+3) = x(t+2) + y(t+2)$$

$$
\begin{aligned}
& \wedge b' = b \wedge y(t+1) = 3 \wedge y(t+2) = x(t+1) + y(t+1) \wedge y(t+3) = y(t+2) \wedge t' = t+3 \\
= \quad & a' = a \wedge x(t+1) = 2 \wedge x(t+2) = 5 \wedge x(t+3) = 10 \\
& \wedge b' = b \wedge y(t+1) = 3 \wedge y(t+2) = y(t+3) = 5 \wedge t' = t+3
\end{aligned}
$$

该例子出现了一个锁步 (lock-step) 的同步，仅因为每次赋值的花费是一个单位时间。更现实地，不同的赋值花费不同的时间，可能要用下界和上界来非确定性地说明。不论决定采用什么时间策略，无论是确定性的还是非确定性的，离散的还是连续的，定义和理论都是不变的。当然，复杂的时间会很快导致描述所有可能交互的非常复杂的表达式。如果只想知道关于可能行为的一些事，而不是所有的事，可以使用蕴含式而不是等式，通过弱化达到简化的目的。程序设计走的是另一条路：从想要的行为的规范开始，根据需要不断强化以获得程序。

9.0.0 自动调温器

练习 452：定义一个燃气控制装置的自动调温器 (Thermostat) 。自动调温器与其它进程并行操作

$$thermometer \| control \| thermostat \| burner$$

温度计和控制器一般位置在一起，但在逻辑上是区分开的。自动调温器的输入为：

- 实数 *temperature*，它来自温度计，表示实际温度。
- 实数 *desired* , 它来自控制器，表示想要的温度。
- 二元值 *flame*, 它来自于燃气装置内的一个火焰传感器，表示燃气是否在燃烧。

这三个变量必须为交互变量，因为它们的值可能在任何时间被另一个进程改变，而自动调温器将根据它们的当前值进行反应。这三个变量不属于自动调温器，也不能被它赋值。自动调温器的输出为：

- 二元值 *gas*, 如果开启燃气则对其赋值 T，如果关闭燃气则对其赋值 ⊥ 。
- 二元值 *spark*，对其赋值 T 会引发火花从而点燃燃气。

变量 *gas* 和 *spark* 属于自动调温器的进程。它们也必须是交互变量，燃气装置需要它们的当前值。

当实际温度比所希望的温度低于 ε 时需要加热，而如果实际温度比所希望的温度高 ε 时就不再需要加热，ε 足够小以至不被注意，又足够大以防迅速震荡。要加热时，使用火花于燃气至少需要一秒钟才能点燃燃气并使火焰成为稳定状态。而一个安全规程要求燃气不可以处于开启但非点燃状态超过 3 秒钟。另一个规程要求燃气装置在关闭之后，至少必须等待 20 秒钟再开启，以清除上次聚集而尚未烧完的燃气。最后，燃气装置必须对它的输入在 1 秒钟内作出响应。

以下是关于此燃气装置的一个规范：

thermostat = (*gas*:= ⊥ || *spark*:= ⊥). *GasIsOff*

GasIsOff = **if** *temperature* < *desired* - ε
then (*gas*:= T || *spark*:= T || t+1 ≤ t' ≤ t+3). *spark*:= ⊥. *GasIsOn*
else (**frame** *gas*, *spark*· *ok*) || $t < t' \le t+1$). *GasIsOff* **fi**

$GasIsOn$ = **if** $temperature < desired + \varepsilon \wedge flame$
then (**frame** $gas, spark \cdot ok$) $\| \; t < t' \le t+1$). $GasIsOn$
else ($gas := \perp \| $ (**frame** $spark \cdot ok$) $\| \; t+20 \le t' \le t+21$). $GasIsOff$ **fi**

这里正在用时间变量来表示真实时间，时间单位为秒。规范 $t+1 \le t' \le t+3$ 表示时间的迁移至少为 1 秒但不超过 3 秒。规范 $t+20 \le t' < t+21$ 与它类似。如果一个规范的计算足够慢，那么它总是可以很容易地被满足。而如果规范要求很快的计算速度，就需要去制造足够快的硬件。在这种情况下是简单的，因为指令时间为微秒，而时间界为秒。

总有人会争论是否一个形式化的规范可以捕捉到一个非形式化的规范的所有思想。例如，如果燃气处于关闭状态，这时需要加热，于是开始点燃燃气的过程，接着又不再需要加热，这最后的输入可能在 3 秒钟内不会被注意到。这一点可能会引起争议：有人认为燃气装置没有在 1 秒钟之内对输入作出响应，另一些人会认为整个点火过程是对第一个输入的响应，而在此完成之前，无需要求对后继输入作出响应。至少这个形式化的规范是清楚明确的。

----------自动调温器结束

9.0.1 空间

交互变量的主要作用是为进程的交互提供一种手段。本节将介绍另一个用处。为了看到计算过程中空间的占用情况，可以将空间变量 s 加入交互变量。练习 449 设法尽可能简单地在一个无限计算中包含时间和空间的计算，这里将它作为一个例子。

假设 $alloc$ 分配一个内存空间，并花费 1 个单位的时间。然后下面的计算慢慢地分配内存。

$$GrowSlow \Leftarrow \textbf{if } t = 2 \times x \textbf{ then } (alloc \| x := t) \textbf{ else } t := t+1 \textbf{ fi}.\ GrowSlow$$

如果时间等于 $2 \times x$，那么分配一个空间，同时 x 成为此分配的时间戳，否则时钟继续。该进程会永远重复。证明如果空间初始时小于时间的对数，且 x 是适当初始化的，那么在所有时间内空间都小于时间的对数。

什么是对 x 的适当的初始化是不明确的，因此先将它放在一边，定义 $GrowSlow$ 为想要的规范。

$$GrowSlow = s < \log t \Rightarrow (\forall t'' \cdot t'' \ge t \Rightarrow s'' < \log t'')$$

其中 s 是一个交互变量，因此 s 实际上是 st 而 s'' 是 st''。这里只对空间计算而不对实际分配空间感兴趣，所以可以将 $alloc$ 看成 $s := s+1$。x 没有必要是交互的，因此将它作为边界变量。为了使证明更简单，令所有的变量为延伸的自然数，尽管这里所证明的结果对实数时间同样成立。

现在需要证明精化，将它分成几段会有帮助。循环的体可写做一个析取式。

if $t = 2 \times x$ **then** $s := s+1 \| \; x := t$ **else** $t := t+1$ **fi**

= $t = 2 \times x \wedge s' = s+1 \wedge x' = t \wedge t' = t+1 \vee t \ne 2 \times x \wedge s' = s \wedge x' = x \wedge t' = t+1$

现在，精化有如下形式

$$(A \Rightarrow B \Leftarrow C \vee D.A \Rightarrow B) \qquad \vee \text{ 上的. 分配}$$

$$= \quad (A \Rightarrow B \Leftarrow (C.A \Rightarrow B) \vee (D.A \Rightarrow B)) \qquad \text{反分配律}$$

$$= \quad (A \Rightarrow B \Leftarrow (C.A \Rightarrow B)) \wedge (A \Rightarrow B \Leftarrow (D.A \Rightarrow B)) \qquad \text{两次移动定律}$$

$$= \quad (B \Leftarrow A \wedge (C.A \Rightarrow B)) \wedge (B \Leftarrow A \wedge (D.A \Rightarrow B))$$

因此可将证明分成两种情形：

$$B \Leftarrow A \wedge (C.A \Rightarrow B)$$

$$B \Leftarrow A \wedge (D.A \Rightarrow B)$$

每次从右边（前件）开始，向左边（后件）进行。第一种情形：

$$s < \log t \wedge (t = 2 \times x \wedge s' = s + 1 \wedge x' = t \wedge t' = t + 1.$$
$$s < \log t \Rightarrow \forall t'' \cdot t'' \geq t \Rightarrow s'' < \log t'')$$

去除相关组合，记住 s 是交互变量

$$= \quad s < \log t \wedge (\exists x'', t''' \cdot t = 2 \times x \wedge s''' = s + 1 \wedge x'' = t \wedge t''' = t + 1$$
$$\wedge (s''' < \log t''' \Rightarrow \forall t'' \cdot t'' \geq t''' \Rightarrow s'' < \log t''))$$

使用 $s''' = s + 1$，并消除它。使用单点定律消除 $\exists x'', t'''$

$$\Rightarrow \quad s < \log t \wedge t = 2 \times x \wedge (s + 1 < \log(t + 1) \Rightarrow \forall t'' \cdot t'' \geq t + 1 \Rightarrow s'' < \log t'')$$

下一步应该为消除。需要

$$s < \log t \wedge t = 2 \times x \Rightarrow s + 1 < \log(t + 1)$$

$$= \quad 2^s < t = 2 \times x \Rightarrow 2^{s+1} < t + 1$$

$$= \quad 2^s < t = 2 \times x \Rightarrow 2^{s+1} \leq t$$

$$= \quad 2^s < t = 2 \times x \Rightarrow 2^{s+1} \leq 2 \times x$$

$$= \quad 2^s < t = 2 \times x \Rightarrow 2^s \leq x$$

$$\Leftarrow \quad 2^s \leq x$$

这里是刚才缺少的 x 的初始化部分。因此回头重新定义 *GrowSlow*。

$$GrowSlow = s < \log t \wedge x \geq 2^s \Rightarrow (\forall t'' \cdot t'' > t \Rightarrow s'' < \log t'')$$

现在再做证明。第一种情形：

$$s < \log t \wedge x \geq 2^s \wedge (t = 2 \times x \wedge s' = s + 1 \wedge x' = t \wedge t' = t + 1.$$
$$s < \log t \wedge x \geq 2^s \Rightarrow \forall t'' \cdot t'' > t \Rightarrow s'' < \log t'')$$

去除相关组合，记住 s 是交互的

$$= \quad s < \log t \wedge x \geq 2^s$$
$$\wedge (\exists x'', t''' \cdot t = 2 \times x \wedge s''' = s + 1 \wedge x'' = t \wedge t''' = t + 1$$
$$\wedge (s''' < \log t''' \wedge x'' \geq 2^{s'''} \Rightarrow \forall t'' \cdot t'' \geq t''' \Rightarrow s'' < \log t''))$$

使用 $s''' = s + 1$，并消除它。使用单点定律消除 $\exists x'', t'''$

$$\Rightarrow \quad s < \log t \wedge x \geq 2^s \wedge t = 2 \times x$$
$$\wedge (s + 1 < \log(t + 1) \wedge t \geq 2^{s+1} \Rightarrow \forall t'' \cdot t'' \geq t + 1 \Rightarrow s'' < \log t'')$$

消除，和前面一样的演算

$$= \quad s < \log t \wedge x \geq 2^s \wedge t = 2 \times x \wedge \forall t'' \cdot t'' \geq t + 1 \Rightarrow s'' < \log t''$$

当 $t'' = t$ 时 $s'' = s$，因为 $s < \log t$，t'' 的定义域可增长

$$\Rightarrow \quad \forall t'' \cdot t'' \geq t \Rightarrow s'' < \log t''$$

第二种情形比第一种要容易。

$$s < \log t \wedge x \geq 2^s \wedge (t \neq 2 \times x \wedge s' = s \wedge x' = x \wedge t' = t + 1.$$
$$s < \log t \wedge x \geq 2^s \Rightarrow \forall t'' \cdot t'' \geq t \Rightarrow s'' < \log t'')$$

去除相关组合，记住 s 是交互的

$$= \quad s < \log t \wedge x \geq 2^s$$
$$\wedge (\exists x'', t''' \cdot t \neq 2 \times x \wedge s''' = s \wedge x'' = x \wedge t''' = t + 1$$
$$\wedge (s''' < \log t''' \wedge x'' \geq 2^{s'''} \Rightarrow \forall t'' \cdot t'' \geq t''' \Rightarrow s'' < \log t''))$$

使用 $s''' = s$，并消除它。使用单点定律消除 $\exists x'', t'''$

$$\Rightarrow \quad s < \log t \wedge x \geq 2^s \wedge t \neq 2 \times x$$
$$\wedge (s < \log t \wedge x \geq 2^s \Rightarrow \forall t'' \cdot t'' \geq t + 1 \Rightarrow s'' < \log t'')$$

消除

$$= \quad s < \log t \wedge x \geq 2^s \wedge t \neq 2 \times x \wedge \forall t'' \cdot t'' \geq t + 1 \Rightarrow s'' < \log t''$$

当 $t'' = t$ 时 $s'' = s$，而因为 $s < \log t$，t'' 的定义域可增长

$$\Rightarrow \quad \forall t'' \cdot t'' \geq t \Rightarrow s'' < \log t''$$

----------空间结束

----------交互变量结束

一个共享变量是一个可以被任何进程读和写的变量。共享变量对于进程交互是普遍的，但对于想对程序进行推理的人和必须建立硬件和软件来实现它们的人，共享变量带来了巨大的问题。因为会带来麻烦，因此就无益了。交互变量不是充分共享的，所有的进程都可以读交互变量，但只有一个进程可以写它。交互变量比充分共享变量更容易推理和实现。即使边界变量也有一点共享：它们的初始值是对所有进程可见的。它们最容易推理和实现，但它们提供的交互最少。

尽管交互变量比共享变量容易处理，但它仍然存在两个问题。第一个问题是它们提供了太多的信息。一般地，进程并不需要所有交互变量在所有时刻的值；每个进程只需要值的某些部分（比如交互变量的一个表达式），并且只是某些时刻的值。另一个问题是进程可能在不同的处理器上执行，且执行的速率不一定是相等的。这就使得很难判断什么时候读一个交互变量的值；当拥有它的进程正在对它进行写的时候，肯定不能去读它的值。

现在来讨论没有这些问题的进程间通信的形式：它只提供所需的信息，调解进程间的时间。而且，很反常，它基本上安全地提供了充分共享变量的手段。

9.1 通信

本节介绍有名通信(communication)信道(channels)，计算通过它们与外界环境通信，这个环境可以是人或其他的并行计算过程。对每个信道，只有一个进程（人或计算）对它进行写，但所有进程可以根据其自身的速度读所有的信息。对于双向的通信，则使用两个信道。在本节的开始只考虑一个读进程，它可能与写进程相同，也可能不同。在后面 9.1.9 的广播一节再考虑多个读进程的情况。

在信道 c 上的通信可以描述成两张无限字符串 Mc 和 Tc，称为信息脚本(message script) 和时间脚本(time script)，和两个延伸自然数变量 rc 和 wc ，称为读游标(read cursor)和写游标(write cursor)。信息脚本是所有信息的字符串，它包括了在这个信道上所传送的过去、现在和将来的信息。时间脚本是信息已被传送的时间和将被传送的时间的相应字符串。这些脚本为状态常量，不是状态变量。读游标是一个状态变量，它代表了在这个信道上已读过或输入过的信息。写游标也是一个状态变量，代表了在这个信道上已经写过或输出过的信息。如果只有一个信道，或如果该信道是上下文已知的，可以省略信道名，把脚本和游标的名称简写为 M，T，r 和 w。

在执行过程中，读写游标的值随输入输出的发生不断增加，越来越多的脚本项被看到但脚本本身并不变化。在任何时候，信道上的未来信息和将被传送的时间可能是未知的，但它们可以作为项存在于脚本中。例如，在信道 c 上再两次读取之后的下一次输入的信息将为信息项 Mc_{rc+2}，而再五次写之后的下一次输出的信息将为 Mc_{wc+5}并且它将发生在时间项 Tc_{wc+5}时刻。省略脚本和游标下标的信道名，再两次读之后的下一次输入将为 M_{r+2} ，而再五次写之后的下一次输出的信息将为 M_{w+5}，它将发生在时刻 T_{w+5} 。

$$M = 6 ; 4 ; 7 ; 1 ; 0 ; 3 ; 8 ; 9 ; 2 ; 5 ; ...$$
$$T = 3 ; 5 ; 5 ; 20 ; 25 ; 28 ; 31 ; 31 ; 45 ; 48 ; ...$$

r w

脚本和游标都不是程序设计记号，但利用它们可以指定任意期望的通信。以下是一个规范的例子。它表示：如果在信道 c 上的下一次输入为偶数，那么在信道 d 上的下一次输出将为 $\top$，否则将为 $\bot$。形式化地，可以写成：

$$\textbf{if } even\,(Mc_{rc}) \textbf{ then } Md_{wd} = \top \textbf{ else } Md_{wd} = \bot \textbf{ fi}$$

或更为简单地写成:

$$Md_{wd} = even\,(Mc_{rc})$$

如果在一个信道上只有有限次的通信，那么在最后的信息之后，时间脚本的项都为 ∞，而对信息脚本的项我们不再感兴趣。

9.1.0 可实现性

考虑如下计算：它包含两个存储变量 x 和 y，一个时间变量 t，和发生在一个单信道（无需下标）上的通信。计算状态包括存储变量值，时间变量值，以及游标变量值。在计算过程中，存储变量可以在任何方向上改变值，但时间和游标只能增长。一旦读了一个输入，这个输入就不再是未读的信息；一旦写了一个输出，这个输出就不再是未写的了。每一个计算都满足：

$$t' \geq t \ \wedge \ r' \geq r \ \wedge \ w' \geq w$$

一个可实现的规范能够指明计算所写的脚本段是什么，也就是写游标的初始值和最后值之间的脚本段 $M_{w:..w'}$ 和 $T_{w:..w'}$ ，但不能指明这个段之外的脚本。进一步，时间脚本必须是单调的，

且其在这一段中的所有值必须在范围 t 和 t' 之间。

一个规范 S（初始状态为σ，终结状态为σ' ,信息脚本为 M，时间脚本为 T）是可实现的当且仅当：

$$\begin{aligned}\forall \sigma, M'', T''\cdot \exists \sigma', M, T\cdot \quad & S \wedge t' \geq t \wedge r' \geq r \wedge w' \geq w \\ & \wedge M_{(0\,;..w)\,;\,(w';..\,\infty)} = M''_{(0\,;..w)\,;\,(w';..\,\infty)} \\ & \wedge T_{(0\,;..w)\,;\,(w';..\,\infty)} = T''_{(0\,;..w)\,;\,(w';..\,\infty)} \\ & \wedge \forall i, j: w\,;..w'\cdot i \leq j \Rightarrow t \leq T_i \leq T_j \leq t'\end{aligned}$$

如果有许多信道，对于每个信道就需要类似的的合取式。如果没有信道，规范的可实现性就与第 4 章中定义的相同。

没有必要为了实现通信信道去构造两个无限字符串。在任意给定的时间，只有那些已被写入但还没有被读的信息是需要存储的。时间脚本仅用于规范和证明，是根本不需要存储的。

----------可实现性结束

9.1.1 输入和输出

这里为通信提供五种程序设计记号。令 c 为一个信道。用 $c!e$ 表示向信道 c 写一个输出信息 e，用 $c!$表示向信道 c 发一个信号（没有信息；发送信号就是唯一的信息）。用 $c?$表示从该信道上读取一个输入，用信道名 c 表示上一次在该信道上读取的信息。用$\surd c$ 代表一个二元表达式，表示“在信道 c 上有未读的输入信息”。以下是它们的形式化定义：

$c!e$	$= M_w = e \wedge T_w = t \wedge (w := w+1)$	“c 输出 e”
$c!$	$= T_w = t \wedge (w := w+1)$	“c 发信号”
$c?$	$= r := r+1$	“c 输入”
c	$= M_{r-1}$	
$\surd c$	$= T_r \leq t$	“检测 c ”

假设来自键盘的输入信道名为 *key*，输出到屏幕的输出信道名为 *screen*，于是以下程序

if $\surd key$
then *key*?.
　　if *key* = “y” **then** *screen*! “ If you wish.” **else** *screen*! “Not if you don't want.” **fi**
else *screen*! “Well?” **fi**

的执行表示：测试是否有一个可用的输入字符，若有，读取它并打印与读取的字符相关的一些输出；若没有，打印另外的输出信息。

现在来精化前面给出的规范 $Md_{wd} = even\,(Mc_{rc})$。

$$Md_{wd} = even\,(Mc_{rc}) \Leftarrow c?.\ d!\ even\ c$$

为了证明这个精化，可将解重写成如下形式：

$$\begin{aligned} & c?.\ d!\ even\ c \\ = \quad & rc := rc+1.\ Md_{wd} = even\,(Mc_{rc-1}) \wedge Td_{wd} = t \wedge (wd := wd+1) \\ = \quad & Md_{wd} = even\,(Mc_{rc}) \wedge Td_{wd} = t \wedge rc'=rc+1 \wedge rd'=rd \wedge wd'=wd+1\end{aligned}$$

此解蕴含了所求证的问题。

一个规范应尽可能地写得清楚和易于理解。程序员通过精化这个规范以获得一个计算机可执行的程序。在例子中，程序看上去比规范更易于理解！一旦遇到这种情形，就应考虑直接用程序来书写规范，从而不需要精化了。

下一个例子是从信道 c 上读数，然后将所读数的两倍的数写到信道 d 上。忽略时间，可将规范写成：

$$S = \forall n: nat \cdot Md_{wd+n} = 2 \times Mc_{rc+n}$$

这里无法假定这里的输入和输出分别为信道 c 和 d 上的第一个输入和输出。而只能要求从现在开始，从初始的读游标 rc 和初始的写游标 wd 开始，输出为输入的双倍。这个规范可以精化成如下形式：

$$S \Leftarrow c?.\ d!\ 2\times c.\ S$$

其证明为：

$$\begin{aligned}
& c?.\ d!\ 2\times c.\ S \\
= \quad & rc := rc+1\ .\ Md_{wd} = 2\times Mc_{rc-1} \wedge (wd := wd+1).\ S \\
= \quad & Md_{wd} = 2\times Mc_{rc} \wedge \forall n: nat \cdot Md_{wd+1+n} = 2 \times Mc_{rc+1+n} \\
= \quad & \forall n: nat \cdot Md_{wd+n} = 2 \times Mc_{rc+n} \\
= \quad & S
\end{aligned}$$

----------输入和输出结束

9.1.2 通信计时

如果采用真实时间度量，必须知道输出需要多少时间，通信传输需要多少时间，以及输入需要多少时间，并且需要设置适当的时间增长量。为了不依赖这些实现细节，可以使用传输时间(transit time)度量。在这种时间度量下，假设输入和输出不占用时间，而通信传输占用一个时间单位。

在信道 c 上下一次要读的的信息是 Mc_{rc}。这个信息曾经或正在或将要在时间 Tc_{rc} 时刻被传送。依据传输时间度量，它的到达时间应为 $Tc_{rc}+1$。所以输入变为：

$$t := max\ t\ (Tc_{rc} + 1).\ c?$$

如果输入已经到达，那么 $Tc_{rc} + 1 \le t$，并且等待输入的时间为零；否则，$c?$的执行将被延迟到输入到达之后。输入检测$\surd c$ 变为：

$$\surd c = Tc_{rc} + 1 \le t$$

在某些应用中(称为“批处理”)，所有的输入都在执行开始时已经可用，对于这些应用，输入可以不考虑对时间的赋值，也不需进行输入检测。在其他一些应用中(称为“进程控制”)，输入通过物理采集设备在规律的时间间隔内到达；时间脚本(而不是信息脚本)事先就已知道。在另外一些应用中(称为 “交互式计算”)，由人类提供的输入在不规律的时间间隔内到达，因此无法确定时间脚本。在这种情况下，只能不考虑等待输入的时间，只在计算中注明执行时间将随任何等待输入的时间的增加而增加。

练习 466(a): 令 W 为“等待信道 c 的输入，然后读取它”。形式地可以写成：

$$W \;=\; t := max\ t\ (T_r + 1).\ c?$$

求证 $W \Leftarrow$ **if** $\sqrt{c}$ **then** $c?$ **else** $t := t{+}1.\ W$ **fi**，假定时间为一个延伸的整数。该练习的关键之处在于输入常常被实现成如下方式：测试输入是否到达，若未到达，循环测试。证明：

if $\sqrt{c}$ **then** $c?$ **else** $t{:=}t{+}1.\ W$ **fi** **替换**$\sqrt{c}$ 和 W

$=$ **if** $T_r{+}1 \le t$ **then** $c?$ **else** $t{:=}t{+}1.\ t := max\ t\ (T_r{+}1).\ c?$ **fi**

$=$ **if** $T_r + 1 \le t$ **then** $t{:=}t.\ c?$ **else** $t{:=}\ max\ (t{+}1)\ (T_r{+}1).\ c?$ **fi**

如果 $T_r{+}1 \le t$，那么 $t = max\ t\ (T_r{+}1)$。

如果 $T_r{+}1{>}t$，那么 $max(t{+}1)\ (T_r{+}1) = T_r{+}1 = max\ t\ (T_r{+}1)$。

$=$ **if** $T_r{+}1 \le t$ **then** $t{:=}\ max\ t\ (T_r{+}1).\ c?$ **else** $t{:=}\ max\ t\ (T_r{+}1).\ c?$ **fi**

$=$ W

----------通信计时结束

9.1.3 递归定义的通信 **可选节；需要第 6 章的知识**

用不动点结构定义 dbl 为(包含递归时间但不考虑输入等待)：

$$dbl \;=\; c?.\ d!\ 2{\times}c.\ t{:=}t{+}1.\ dbl$$

把 dbl 看作未知，上式有几个解。其中最弱的一个为：

$$\forall\ n{:}\ nat \cdot Md_{wd+n} = 2 \times Mc_{rc+n} \wedge Td_{wd+n} = t{+}n$$

最强的一个可实现的解为：

$$(\forall\ n{:}\ nat \cdot Md_{wd+n} = 2 \times Mc_{rc+n} \wedge Td_{wd+n} = t{+}n)$$
$$\wedge\ \ rc' = wd' = t' = \infty \wedge wc' = wc \wedge rd' = rd$$

最强的一个解为$\perp$。如果这个不动点构造是所有所知的关于 dbl 的信息，那么不能说它等价于哪一个特定的解。但是可以说：它精化了最弱的解。

$$\forall\ n{:}\ nat \cdot Md_{wd+n} = 2 \times Mc_{rc+n} \wedge Td_{wd+n} = t{+}n \Leftarrow dbl$$

并且它又被不动点构造的右式精化：

$$dbl \Leftarrow c?.\ d!\ 2{\times}c.\ t{:=}t{+}1.\ dbl$$

于是可以用它来解决问题，然后执行。

如果从下式开始递归构造：

$dbl_0 = \mathsf{T}$

则有

$dbl_1 \;=\; c?.\ d!\ 2{\times}c.\ t := t{+}1.\ dbl_0$

$=\ rc := rc{+}1\ .\ Md_{wd} = 2{\times}Mc_{rc-1} \wedge Td_{wd} = t \wedge (wd := wd{+}1).\ t := t{+}1.\ \mathsf{T}$

$= Md_{wd} = 2{\times}\ Mc_{rc} \wedge Td_{wd} = t$

$dbl_2 \;=\; c?.\ d!\ 2{\times}c.\ t := t{+}1.\ dbl_1$

$=\ rc := rc{+}1\ .\ Md_{wd} = 2{\times}Mc_{rc-1} \wedge Td_{wd} = t \wedge (wd := wd{+}1).$

$\quad t := t{+}1.\ Md_{wd} = 2{\times}\ Mc_{rc} \wedge Td_{wd} = t$

$= Md_{wd} = 2{\times}Mc_{rc} \wedge Td_{wd} = t \wedge\ Md_{wd+1} = 2{\times}Mc_{rc+1} \wedge Td_{wd+1} = t{+}1$

然后依次类推。构造的结果

$dbl_\infty = \forall\, n{:}\, nat \cdot Md_{wd+n} = 2{\times}Mc_{rc+n} \wedge Td_{wd+n} = t{+}n$

就是*dbl*不动点构造式的最弱解。如果从包含$t' \geq t \wedge rc' \geq rc \wedge wc' \geq wc \wedge rd' \geq rd \wedge wd' \geq wd$的递归结构出发，就可得到最强的可实现的解。

----------递归定义的通信结束

9.1.4 合并

合并(merge)意指重复地从两个或两个以上的输入信道上读取信息，然后将所读的信息输出到第三个信道上。这个输出是不同输入信道上的输入信息的交替再现。它必须是也只能是所输入的信息，而且必须保持信息原来的读取次序。无穷合并可以形式化地定义成如下：令 c 和 d 为输入信道，e 为输出信道，于是

$$merge = (c?.\, e!\, c) \vee (d?.\, e!\, d).\, merge$$

这个规范并未指出每一步不同输入信道的选择准则。但为了写出合并程序，必须确定一个选择准则。这里可以根据输入的值或信息到达的时间来选择输入信道。

练习 472(a)（时间合并）要求在每一步选择第一个可用的输入。如果输入在信道 c 和 d 都可用，可以选任何一个。如果只在一个信道可用，则选取那一个。如果在两个信道都不可用，等待第一个并选取它（在达到可用的时间相同的情况下，可以任选）。下面是规范：

$$timemerge = (\surd c \vee Tc_{rc} \leq Td_{rd}) \wedge (c?.\, e!\, c) \vee (\surd d \vee Tc_{rc} \geq Td_{rd}) \wedge (d?.\, e!\, d).\, timemerge$$

为了计算等待输入的时间，应该在每个输入操作前插入 $t := max\ t\ (T_{\cdot} + 1)$，对于递归时间则应在递归调用前插入 $t := t + 1$。

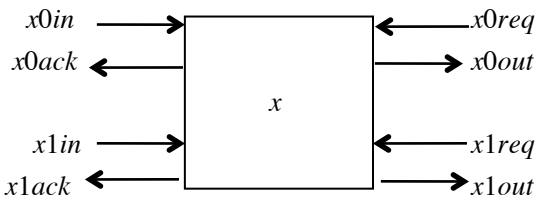
9.1.2 的通信时间一节证明了等待输入可以被递归实现。使用同样的原因，假设时间是个延伸整数，可以将 *timemerge* 实现如下：

$$timemerge \Leftarrow \textbf{if}\ \surd c\ \textbf{then}\ c?.\, e!\, c\ \textbf{else}\ ok\ \textbf{fi}.\ \textbf{if}\ \surd d\ \textbf{then}\ d?.\, e!\, d\ \textbf{else}\ ok\ \textbf{fi}.\ t{:=}t{+}1.\ timemerge$$

----------合并结束

9.1.5 监控器

为了得到一个完全的共享变量的效果，这里创建一个称为监控器(monitor)的进程，以解决对变量的冲突使用的问题。当变量 x 的监控器接收到从其他进程经过信道 $x0in,\ x1in,\ ...$传来的要求写到该变量上的数据时，就通过信道 $x0ack,\ x1ack,\ ...$之一发送一个确认信息给写进程。当收到从其他进程经过信道 $x0req,\ x1req,\ ...$传来的要求读取该变量值的请求时，就通过信道 $x0out, x1out, ...$之一将该变量的值传送给发要求的进程。



一个带有两个写进程和两个读进程的变量 x 的监控器可被定义如下。令 m 为下一个输入在每个输入信道的时间 $Tx0in_{rx0in}$，$Tx1in_{rx1in}$，$Tx0req_{tx0req}$和 $Tx1req_{tx1req}$ 的最小值，那么

$$
\begin{aligned}
monitor = \quad & (\surd x0in \vee Tx0in_{rx0in} = m) \wedge (x0in?.\, x := x0in.\, x0ack!) \\
\vee \quad & (\surd x1in \vee Tx1in_{rx1in} = m) \wedge (x1in?.\, x := x1in.\, x1ack!) \\
\vee \quad & (\surd x0req \vee Tx0req_{tx0req} = m) \wedge (x0req?.\, x0out!\, x) \\
\vee \quad & (\surd x1req \vee Tx1req_{tx1req} = m) \wedge (x1req?.\, x1out!\, x). \\
& monitor
\end{aligned}
$$

正如 *timemerge*，监控器获取第一个输入变量并回应它。几个变量，几个写进程，几个读进程的监控器也是类似的。当有超过一个的输入可用时，实现必须进行选择。下面是一个实现监控器的方法，假定时间是延伸整数：

$$
\begin{aligned}
monitor \Leftarrow \quad & \textbf{if } \surd x0in \textbf{ then } x0in?.\, x := x0in.\, x0ack! \textbf{ else } ok \textbf{ fi}. \\
& \textbf{if } \surd x1in \textbf{ then } x1in?.\, x := x1in.\, x1ack! \textbf{ else } ok \textbf{ fi}. \\
& \textbf{if } \surd x0req \textbf{ then } x0req?.\, x0out!\, x \textbf{ else } ok \textbf{ fi}. \\
& \textbf{if } \surd x1req \textbf{ then } x1req?.\, x1out!\, x \textbf{ else } ok \textbf{ fi}. \\
& t := t+1.\, monitor
\end{aligned}
$$

前面解决了练习 452，使用交互变量 *gas*，*temperature*，*desired*，*flame*，和 *spark* 表示了一个燃气炉的自动调温器，如下。

$$thermostat = (gas := \bot \parallel spark := \bot).\, GasIsOff$$

$$
\begin{aligned}
GasIsOff = \quad & \textbf{if } temperature < desired - \varepsilon \\
& \textbf{then } (gas := \top \parallel spark := \top \parallel t+1 \le t' \le t+3).\, spark := \bot.\, GasIsOn \\
& \textbf{else } (\textbf{frame } gas, spark \cdot ok) \parallel t < t' \le t+1).\, GasIsOff \textbf{ fi}
\end{aligned}
$$

$$
\begin{aligned}
GasIsOn = \quad & \textbf{if } temperature < desired + \varepsilon \wedge flame \\
& \textbf{then } (\textbf{frame } gas, spark \cdot ok) \parallel t < t' \le t+1).\, GasIsOn \\
& \textbf{else } (gas := \bot \parallel (\textbf{frame } spark \cdot ok) \parallel t+20 \le t' \le t+21).\, GasIsOff \textbf{ fi}
\end{aligned}
$$

如果使用通信信道代替交互变量，必须构造这些变量的监控器，并重写自动调温器的规范。下面是结果。

$$thermostat = ((gasin! \bot.\, gasack?) \parallel (sparkin! \bot.\, sparkack?)).\, GasIsOff$$

$$
\begin{aligned}
GasIsOff = \quad & ((temperaturereq!.\, temperature?) \parallel (desiredreq!.\, desired?)). \\
& \textbf{if } temperature < desired - \varepsilon
\end{aligned}
$$

then ((*gasin*! ⊤. *gasack*?.) ‖ (*sparkin*! ⊤. *sparkack*?) ‖ $t+1 \le t' \le t+3$) .

sparkin! ⊥ . *sparkack*?. *GasIsOn*

else $t \le t' \le t+1$. *GasIsOff* **fi**

GasIsOn = ((*temperaturereq*!. *temperature*?) ‖ (*desiredreq*!. *desired*?)

‖ (*flamereq*!. *flame*?)).

if *temperature* < *desired* + ε ∧ *flame*

then $t \le t' \le t+1$. *GasIsOn*

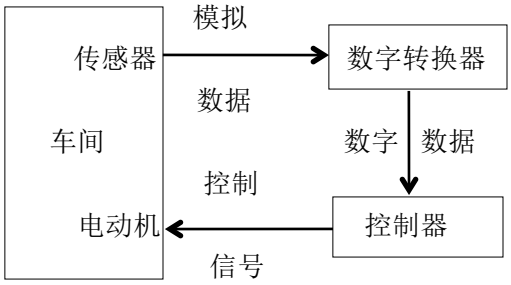
else ((*gasin*! ⊥ . *gasack*?) ‖ $t+20 \le t' \le t+21$) . *GasIsOff* **fi**

----------监控器结束

当存在并发时，空间计算有时可能要求一个有空间变量的监控器，以至于任何进程都可以要求更新，并且更新可以被通信到所有进程。空间变量的监控器也是互相竞争的空间分配要求之间的仲裁者。

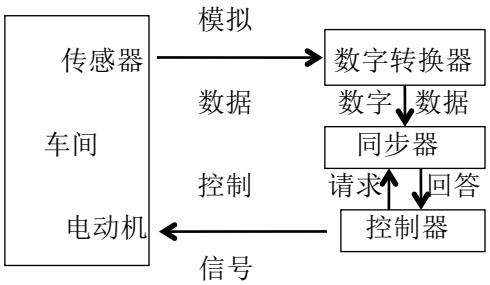
9.1.6 反应控制器

许多反应装置都是通过反馈循环来控制的，如下图所示：



其中“车间”可以是一个化学反应器，或者是一个核反应堆，甚至仅为一个装配车间。传感器负责测试浓度、温度或位置的模拟数据，并将它们传送给数字转换器。数字转换器将这些数据转换成控制器所需的数字形式。控制器根据这些数据计算出对此装置所应进行的下一步控制；或许应该进一步推进一些杆，或者打开一些阀门，或者沿某个方向移动机器人手臂。控制器向车间发送一些信号从而引起合适的改变。

这里有一个问题，传感器不断地将数据传送给数字转换器，而数字转换器又始终如一地将数字数据迅速传送给控制器。由于控制器计算输出信号所需的时间随输入信息的不同而不同；有时计算很简单，能够跟上数据输入的速度；但有时计算较为复杂，就可能处于落后状态。当有几个输入被堆积起来之后，控制器不应该为了追赶速度而连续不断地依次读取它们并计算输出。相反，我们希望控制器只处理最后的那个输入而抛弃其余的。控制信号的生成速度是否能和数字数据的生成一样快这一点并不重要。关键在于每一个控制信号的生成必须是基于最近获得的数据。如何能做到这一点？一个解决办法是在数字转换器和控制器之间加入一个同步器，如图所示：



同步器的工作和数字转换器一样简单统一，因此能够跟上处理速度。它重复地从数字转换器读取数据，总是保留最后一个读取的数据。每当控制器请求读取某些数据时，同步器将最近的数据传送给它。这一功能与监控器的功能完全一样，因此可以采用与监控器一样的实现方法来实现同步器。但是同步器比监控器更为简单，这体现在两个方面：首先，仅有一个写进程和一个读进程；第二，写进程总是比读进程要快。以下是它的定义：

synchronizer = *digitaldata*?.
if √*request* **then** *request*? ‖ *reply*! *digitaldata* **else** *ok* **fi**.
synchronizer

如果使用交互变量替代信道，那么就不会有读取老的数据的问题；读取交互变量总是读取最近的值，即使该变量写的频率高于读的频率。但保证交互变量在被写时不被读却是个问题。

----------反应控制器结束

9.1.7 信道声明

信道的下一个输入未必就是信道中最后写的信息。在一个变量 x 和一个信道 c 的情况下（忽略时间），

$$c!2.\ c?x := c$$
$$= \ M_w = 2 \wedge w' = w+1 \wedge r' = r+1 \wedge x' = M_r$$

由于不知道初始时 $w = r$，因此不能下结论说最后有 $x' = 2$。因为可能有一个前面的写尚未被读取。例如，

$$c!1.\ c!2.\ c?.\ x := c$$

信道的下一个输入总是信道中尚未被读取的第一个。在独立组合中也同样成立。

$$c!2 \parallel (c?.\ x := c)$$
$$= \ M_w = 2 \wedge w' = w+1 \wedge r' = r+1 \wedge x' = M_r$$

同样不能说 $x' = 2$ 因为前面可能有未被读取的输出

$$c!1.\ (c!2 \parallel (c?.\ x := c)).\ c?$$

x 的最后值可能是从前面的输出得到的 1，伴随着 2 成了后面的输入。为了获取进程间有用的通信，必须引入一个局部信道。

信道声明与变量说明类似，它在一个程序或规范的某些局部区域定义了一个新的信道。信道声明根据本书最后一页的优先顺序作用于跟随其后的部分。以下是一个语法结构及等价的规

范：

$$\textbf{chan}\ \ c\colon T \cdot P \ =\ \exists\, Mc\colon \infty^{*} T \cdot \exists\, Tc\colon \infty^{*} xreal \cdot\ \textbf{var}\ rc, wc\colon xnat := 0 \cdot P$$

类型 T 指明了在这个新的信道上什么样的通信是可能的。这个说明引入了两个脚本，都为无限表；它们不是状态变量，而是值未知的状态常量(数学变量)。我们令时间为延伸实数，但不能令时间为延伸整数。这个信道声明也引入了一个读游标 rc, 它的初始值为 0，表示一开始该信道上没有输入，同时还引入了一个写游标 wc，其初始值也为 0，表示一开始该信道上没有输出。

局部信道可以如一个没有并发的队列或缓冲区一样使用。例如，

$$\textbf{chan}\ \ c : \text{int} \cdot\ c!3.\ c!4.\ c?.\ x := c.\ c?.\ x := x + c$$

将 7 赋给 x。下面是包括时间的证明。

$$\textbf{chan}\ \ c : \text{int} \cdot c!3.\ c!4.\ t := \max\ t\,(Tr + 1).\ c?.\ x := c.\ t := max\ t\,(Tr\ +1).\ c?.\ x := x + c$$

$$= \ \exists M : \infty^{*} \text{int} \cdot \exists\, T : \infty^{*}\, x\text{int} \cdot\ \textbf{var}\, r, w : xnat := 0 \cdot$$
$$M_w = 3 \wedge T_w = t \wedge (w := w + 1).$$
$$M_w = 4 \wedge\ T_w = t \wedge (w := w + 1).$$
$$t := \max\ t\ (T_r + 1).\ r := r + 1.$$
$$x := M_{r-1}.$$
$$t := \max t\ (T_r + 1).\ r := r + 1.$$
$$x := x + M_{r-1}$$

现在用几次置换定律

$$= \ \exists M : \infty^{*} \text{int} \cdot \exists\, T : \infty^{*}\, x\, \text{int} \cdot\ \exists r, r', w, w' : xnat \cdot$$
$$M_0 = 3 \wedge T_0 = t \wedge M_1 = 4 \wedge T_1 = t \wedge r' = 2 \wedge w' = 2 \wedge x' = M_0 + M_1$$
$$\wedge t' = \max\ (\max t\ (T_0 + 1))(T_1 + 1) \wedge \text{(其他变量不变)}$$

$$= \ x' = 7 \wedge t' = t + 1 \wedge \text{（其他变量不变）}$$

这里有两个进程，它们之间发生通信。忽略时间，

$$\textbf{chan}\ \ c\colon int \cdot c!\ 2 \parallel (c?.\ x := c)$$

利用局部信道声明的定义，和前面独立组合的结果

$$= \ \exists\, M\colon \infty^{*} int \cdot\ \textbf{var}\ r, w\colon xnat := 0 \cdot$$
$$M_w = 2 \wedge w' = w + 1 \ \wedge r' := r + 1 \wedge x' = M_r \ \wedge \text{(其他变量值不变)}$$

现在利用置换定律来应用初始化条件 $r := 0$ 和 $w := 0$

$$= \ \exists\, M\colon \infty^{*} int \cdot\ \textbf{var}\ r, w\colon xnat \cdot$$
$$M_0 = 2 \ \wedge w' = 1 \wedge r' = 1 \wedge x' = M_0 \ \wedge \text{(其他变量值不变)}$$
$$= \ x' = 2 \wedge \text{(其他变量值不变)}$$
$$= \ x := 2$$

用任意一个表达式替换 2，就可以得到一个更为一般的理论，它把对一个局部信道的通信等同于赋值。如果包括时间，结果是：

$$x' = 2 \wedge t' = t + 1 \wedge \text{（其他变量值不变）}$$
$$= \ x := 2.\ t := t + 1$$

----------信道声明结束

9.1.8 死锁

前一节说明了局部信道可以用作缓冲区。现在来看看如果先读后写会发生什么。插入等待输入到下式中：

$$\textbf{chan}\ \ c\colon int \cdot c?.\ c!\ 5$$

得到：

$$\textbf{chan}\ \ c\colon int\cdot\ t := max\ t\ (T_r + 1).\ c?.\ c!\ 5$$

$$= \quad \exists\, M\colon \infty\ {}^*int\cdot \exists\, T\colon \infty\ {}^*xint \cdot \textbf{var}\ r, w\colon xnat := 0\cdot$$

$$t := max\ t\ (T_r + 1).\ r = r + 1.\ M_w = 5 \wedge T_w = t \wedge (w := w+1)$$

现在一步一步慢慢处理这个等式。首先，扩展 **var** 和 $w := w+1$，把 r, w, x, 和 t 作为状态变量。

$$= \quad \exists\, M\colon \infty\ {}^*int\cdot \exists\, T\colon \infty\ {}^*xint\cdot \exists\, r,r', w,w'\colon xnat\cdot$$

$$r := 0.\ w := 0.\ t := max\ t\ (T_r + 1).\ r := r + 1.$$

$$M_w = 5 \wedge T_w = t \wedge r' = r \wedge w' = w+1 \wedge x' = x \wedge t' = t$$

现在应用置换定律四次。

$$= \quad \exists\, M\colon \infty\ {}^*int\cdot \exists\, T\colon \infty\ {}^*xint\cdot \exists\, r,r', w,w'\colon xnat\cdot$$

$$M_0 = 5 \wedge T_0 = max\ t\ (T_0 + 1) \wedge r' = 1 \wedge w' = 1 \wedge x' = x \wedge t' = max\ t\ (T_0 + 1)$$

考虑合取式 $T_0 = max\ t\ (T_0 + 1)$，对任意起始时间 $t > -\infty$ 有 $T_0 = \infty$。

$$= \quad x' = x \wedge t' = \infty$$

这个定理说明：由于等待输入的时间为无穷，因此执行将永远进行下去。

单词“死锁” (deadlock) 经常用于表示几个进程互相等待，例如第 8 章中的哲学家就餐的例子。但它也可能用于表示一个顺序计算内部的等待，像上一段的例子。以下是包含两个进程的更传统的例子：

$$\textbf{chan}\ \ c, d\colon int\cdot\ (c?.\ d!\ 6\ ) \parallel (d?.\ c!\ 7\ )$$

加入输入等待，得到：

$$\textbf{chan}\ \ c, d\colon int\cdot\ (t := max\ t\ (Tc_{rc} + 1).\ c?.\ d!\ 6\ ) \parallel (t := max\ t\ (Td_{rd} + 1).\ d?.\ c!\ 7)$$

再进行一些加工，得到

$$= \quad \exists\, Mc, Md\colon \infty\ {}^*int\cdot \exists\, Tc, Td\colon \infty\ {}^*xint\cdot \exists\, rc, rc', wc, wc', rd, rd', wd, wd'\colon xnat\cdot$$

$$Md_0 = 6 \wedge Td_0 = max\ t\ (Tc_0 + 1) \wedge Mc_0 = 7 \wedge Tc_0 = max\ t\ (Td_0 + 1)$$

$$\wedge\ rc' = wc' = rd' = wd' = 1 \wedge x' = x \wedge t' = max\ (max\ t\ (Tc_0 + 1))\ (max\ t\ (Td_0 + 1))$$

对起始时间 $t > -\infty$，合取式 $Td_0 = max\ t\ (Tc_0 + 1)$ 和 $Tc_0 = max\ t\ (Td_0 + 1)$告诉我们有 $Td_0 = Tc_0 = \infty$。

$$= \quad x' = x \wedge t' = \infty$$

为了证明一个计算没有死锁，需要证明所有消息时间是有限的。

----------死锁结束

9.1.9 广播

一个信道由一个信息脚本，一个时间脚本，一个读游标和一个写游标组成。当一个计算分化成并行进程时，状态变量必须在这些进程间进行划分。脚本不是状态变量，它们不属于任何进程。游标是状态变量，因此一个进程可以向信道中写，一个进程（可能是同一个，也可能是不同的进程）可以从信道中读。假设结构是：

$$P.\ (Q \parallel R \parallel S).\ T$$

假设Q向信道c中写而R从信道c中读。Q所写的信息跟在P所写的信息后面，T所写的信息跟在Q所写的信息后面。R所读的信息跟在P所读的信息后面，T所读的信息跟在R所读的信息后面。不存在两个进程想同时写的问题，时间规定保证一个信息的读取要等到它被写入以后。

尽管信道的通信，到目前为止的定义，是从一个写者向一个读者的单方向，可以根据需要使用任意多的信道。因此可以达到所有进程对之间的双向对话。但有时从一个进程向多个并行进程的广播(broadcast)更为便捷。在前小段的程序结构中，可能希望Q写入且R和S都从同一个信道中读。广播通过多个读游标得以实现，每个读进程一个读游标。然后所有读进程以自己的速率读取同样的信息。两个进程读取同一个信息，即使是同时读取都是没有问题的。但广播有一个问题：哪一个进程的读游标成为T的读游标呢？每个读游标开始于同样的值，但可能不结束于同一个值。从那个信道继续读没有明智的方法。因此只在并行组合后面不会顺序跟着一个从那个信道读的程序时才允许在该信道上广播。

下面的广播例子在一个漂亮的小程序中同时包含通信进程，局部信道声明，和动态进程生成。它也是一个典型的例子，说明了好的记号和好的理论的重要性。这个问题在没有这些记号和理论之前也可以被“解决”，但那个“解”需要写满很多页纸，复杂的同步变量，缺乏证明，有时还有可能是错的。

练习 478 是一个幂级数乘法：写一个程序, 从信道 a 上读取一个幂级数 $a_0 + a_1x + a_2x^2 + a_3x^3+...$ 的无穷系数序列 a_0 , a_1 , a_2 , a_3 ..., 并且并行地从信道 b 上读取幂级数 $b_0 + b_1x + b_2x^2 + b_3x^3+...$的无穷系数序列 b_0 , b_1 , b_2 , b_3 ..., 然后向信道 c 写幂级数 $c_0 + c_1x + c_2x^2 + c_3x^3+...$的无穷系数序列 c_0 , c_1 , c_2 , c_3 ..., 它等于前两个输入级数的乘积。假设所有的输入已经可用，没有输入延迟。要求在每一个时间单位内生成一个输出。

该问题提供了一种表示系数的记号：$a_n = Ma_{ra+n}$，$b_n = Mb_{rb+n}$和$c_n = Mc_{rc+n}$。用 A，B，C 代表幂级数，于是可以将希望的结果表示成：

$$\begin{aligned} C &= A \times B \\ &= (a_0 + a_1 \times x + a_2 \times x^2 + a_3 \times x^3+...) \times (b_0 + b_1 \times x + b_2 \times x^2 + b_3 \times x^3+...) \\ &= a_0 \times b_0 + (a_0 \times b_1 + a_1 \times b_0) \times x + (a_0 \times b_2 + a_1 \times b_1 + a_2 \times b_0) \times x^2 \\ &\quad + (a_0 \times b_3 + a_1 \times b_2 + a_2 \times b_1 + a_3 \times b_0) \times x^3+... \end{aligned}$$

从上式可看出 $c_n = \sum i: 0,..n+1 \cdot a_i \times b_{n-i}$ 。虽然该问题无需考虑输入时间，但还要考虑输出时间。完整的规范为：

$$C = A \times B \ \wedge \ \forall n \cdot Tc_{wc+n} = t+n$$

考虑如下问题：输出系数 n 的计算需要对 $2(n+1)$个输入系数进行 $n+1$ 次乘法和 n 次加法，而且必须在前一个系数得出之后的一个时间单位时计算出来。为了做到这一点，随着执行的进行需要存储越来越多的数据和进行越来越多的并行计算。

照常，首先集中精力考虑结果，之后再考虑时间。令

$A_1 = a_1 + a_2 \times x + a_3 \times x^2 + a_4 \times x^3 + \ldots$

$B_1 = b_1 + b_2 \times x + b_3 \times x^2 + b_4 \times x^3 + \ldots$

它们表示以系数 1 开头的从信道 a 和 b 上得到的幂级数。于是

$$\begin{aligned} & A \times B \\ = \quad & (a_0 + A_1 \times x) \times (b_0 + B_1 \times x) \\ = \quad & a_0 \times b_0 + (a_0 \times B_1 + A_1 \times b_0) \times x + A_1 \times B_1 \times x^2 \end{aligned}$$

现在可以考虑五个新问题来代替原来的 $A \times B$ 问题。第一个问题是从每个输入信道读取一个系数，然后输出它们的乘积，这很简单。接着两个问题是计算 $a_0 \times B_1$ 和 $A_1 \times b_0$，它们是将两个幂级数分别与两个常数相乘，这比直接求两个幂级数的乘积要简单，它只需要一个循环。第四个问题是求 $A_1 \times B_1$，它正好是一开始要求解的问题，但系数向下移了一位，这个问题可通过递归解决。最后，必须把三个幂级数相加。遗憾的是这三个幂级数的计算无法很好地同步起来。现在必须在尚未得到 $A_1 \times B_1$ 的系数之前，先将 $a_0 \times B_1$ 和 $A_1 \times b_0$ 的前面的对应系数相加，然后再将它的第 $n+1$ 个系数加到 $A_1 \times B_1$ 的第 n 个系数上。为了使它们的计算同步起来，将 $a_0 \times B_1$ 和 $A_1 \times b_0$ 再向下移一个系数，令

$A_2 = a_2 + a_3 \times x + a_4 \times x^2 + a_5 \times x^3 + \ldots$

$B_2 = b_2 + b_3 \times x + b_4 \times x^2 + b_5 \times x^3 + \ldots$

它们表示以系数 2 开头的从信道 a 和 b 上得到的幂级数。继续前一个 $A \times B$ 的等式，有：

$$\begin{aligned} = \quad & a_0 \times b_0 + (a_0 \times (b_1 + B_2 \times x) + (a_1 + A_2 \times x) \times b_0) \times x + A_1 \times B_1 \times x^2 \\ = \quad & a_0 \times b_0 + (a_0 \times b_1 + a_1 \times b_0) \times x + (a_0 \times B_2 + A_1 \times B_1 + A_2 \times b_0) \times x^2 \end{aligned}$$

从这个期望的乘积的扩展式中，几乎可以直接写出一个解。

现在还剩下一个问题。为了计算 $A \times B$ 的系数，需要使用一次递归调用以获得乘积 $A_1 \times B_1$ 的系数序列。但是 $A_1 \times B_1$ 的输出信道不可以为信道 c，因为它已经是主计算 $A \times B$ 的输出信道。因此必须使用一个局部信道作为 $A_1 \times B_1$ 的输出之用。这里需要一个信道参数，因此引入一个记号⟨! ⟩。一个信道参数实际上为四个参数：一个信息脚本参数，一个时间参数，一个读游标参数，以及一个写游标参数。（这里的游标为变量，所以它们的参数为引用参数，参见 5.5.2 节。）

现在准备好了。定义 P (为了乘积)为所需的规范(暂时忽略时间)，被输出信道参数化：

$$P = \langle !c : rat \rightarrow C = A \times B \rangle$$

精化 $P\ c$ 成如下：

$P\ c \Leftarrow (a? \parallel b?).\ c!\ a \times b.$

var $a0$: $rat := a \cdot$ **var** $b0$: $rat := b \cdot$ **chan** d: $rat \cdot$

$P\ d \Vert ((a? \parallel b?).\ c!\ a0 \times b + a \times b0.\ C = a0 \times B + D + A \times b0)$

$$C = a0\times B + D + A\times b0 \Leftarrow (a?\|b?\|d?).\ c!\ a0\times b + d + a\times b0.\ C = a0\times B + D + A\times b0$$

这就是整个程序：仅有 4 行！首先，分别从信道 a 和 b 上读取一个输入，并将其乘积输出到信道 c 上，这样得到 $a_0\times b_0$。这些值将会被再次用到，因此定义局部变量 $a0$ 和 $b0$ (实际为常数)来保留它们的值。现在已经从每个输入信道上读取了一个信息，调用 $P\ d$，在局部信道 d 上获取 $A_1\times B_1$ 的系数，这个计算与程序的其余部分并行执行。$P\ d$ 和与它并行的进程会用不同的读游标从信道 a 和 b 上读取信息；这之后没有按顺序跟着它们的计算。和 $P\ d$ 并行读下一个输入 a_1 和 b_1，并输出系数 $a_0\times b_1 + a_1\times b_0$。最后执行循环 $C = a0\times B + D + A\times b0$，其中 D 是从信道 d 读取的系数的幂级数。

这个证明非常直观，以下是它的详细证明。从第一个精化式的右边开始。

$(a?\ \|\ b?).\ c!\ a\times b.$

var $a0$: $rat := a\cdot$ **var** $b0$: $rat := b\cdot$ **chan** d : $rat\cdot$
$P\ d\ \|\ ((a?\|\ b?).\ c!\ a0\times b + a\times b0.\ C = a0\times B + D + A\times b0\ )$

$=$ $(r_a := r_a+1\ \|\ r_b := r_b+1).\ Mc_{wc} = Ma_{ra-1}\times Mb_{rb-1} \wedge (wc := wc+1).$

$\exists\ a0, a0', b0, b0', Md, rd, rd', wd, wd'\cdot$
$a0 := Ma_{ra-1}.\ b0 := Mb_{rb-1}.\ rd := 0.\ wd := 0.$
$(\forall n\cdot Md_{wd+n} = (\sum i: 0,..\ n+1\cdot Ma_{ra+i}\times Mb_{rb+n-i}))$
$\wedge\ ((ra := ra+1\ \|\ rb := rb+1).\ Mc_{wc} = a0\times Mb_{rb-1} + Ma_{ra-1}\times b0 \wedge (wc := wc+1).$
$\forall n\cdot Mc_{wc+n} = a0\times Mb_{rb+n} + Md_{wd+n} + Ma_{ra+n}\times b0)$

对赋值语句作所有的置换。

$=$ $Mc_{wc} = Ma_{ra}\times Mb_{rb}$
$\wedge\ \exists\ a0, a0', b0, b0', Md, rd, rd', wd, wd'\cdot$
$(\forall n\cdot Md_n = \sum i: 0,..\ n+1\cdot Ma_{ra+1+i}\times Mb_{rb+1+n-i})$
$\wedge\ Mc_{wc+1} = Ma_{ra}\times Mb_{rb+1} + Ma_{ra+1}\times Mb_{rb}$
$\wedge\ (\forall n\cdot Mc_{wc+2+n} = Ma_{ra}\times Mb_{rb+2+n} + Md_n + Ma_{ra+2+n}\times Mb_{rb})$

利用第一个全称量词式去替换第二个全称量词式中的 Md_n。
然后丢弃第一个全称量词式(弱化我们的表达式)。
现在所有的存在量词式都没有被使用，因此可以将它们抛弃。

$\Rightarrow$ $Mc_{wc} = Ma_{ra}\times Mb_{rb}$
$\wedge\ Mc_{wc+1} = Ma_{ra}\times Mb_{rb+1} + Ma_{ra+1}\times Mb_{rb}$
$\wedge\ \forall n\cdot Mc_{wc+2+n} = Ma_{ra}\times Mb_{rb+2+n}$
$+\ (\sum i: 0,..\ n+1\cdot Ma_{ra+1+i}\times Mb_{rb+1+n-i})$
$+\ Ma_{ra+2+n}\times Mb_{rb}$

现在合并这三个合取式。

$=$ $\forall n\cdot Mc_{wc+n} = \sum i: 0,..\ n+1\cdot Ma_{ra+i}\times Mb_{rb+n-i}$
$=$ $P\ c$

还必须证明对循环的精化：

$$(a?\,\|\,b?\,\|\,d?).\ c!\ a0\times b + d + a\times b0.\ C = a0\times B + D + A\times b0$$

$$= \quad (ra := ra+1\,\|\,rb := rb+1\,\|\,rd := rd+1).$$

$$Mc_{\,wc} = a0\times Mb_{\,rb-1} + Md_{\,rd-1} + Ma_{\,ra-1}\times b0 \wedge (wc := wc+1).$$

$$\forall n\cdot Mc_{\,wc+n} = a0\times Mb_{\,rb+n} + Md_{\,rd+n} + Ma_{\,ra+n}\times b0$$

对赋值语句作所有的置换。

$$= \quad Mc_{\,wc} = a0\times Mb_{\,rb} + Md_{\,rd} + Ma_{\,ra}\times b0$$

$$\wedge\ \forall n\cdot Mc_{\,wc+1+n} = a0\times Mb_{\,rb+1+n} + Md_{\,rd+1+n} + Ma_{\,ra+1+n}\times b0$$

合并这两个合取式。

$$= \quad \forall n\cdot Mc_{\,wc+n} = a0\times Mb_{\,rb+n} + Md_{\,rd+n} + Ma_{\,ra+n}\times b0$$

$$= \quad C = a0\times B + D + A\times b0$$

根据递归时间度量，必须把一个时间增量放置在递归调用 Pd 和递归调用 $C = a0\times B + D + A\times b0$ 之前。根据这个问题提供的信息，不必在信道 a 和 b 的输入之前加入时间增量。但在信道 d 的输入之前需加一个时间增量。如果只加入这些必要的时间增量，输出 $c_0 = a_0\times b_0$ 将在预期的 $t+0$ 时刻发生，但输出 $c_1 = a_0\times b_1 + a_1\times b_0$ 也将在 $t+0$ 时刻发生，这就太快了。为了使得 c_1 在预期的 $t+1$ 时刻发生，必须在头两个输出之间加入一个时间增量。这里可以用这个时间增量来说明实际的计算时间，或者仅作为一个延迟（参见 5.3 节“时间和空间依赖”）。以下是加入时间后的程序：

$$Q\ c \Leftarrow (a?\,\|\,b?).\ c!\ a\times b.$$

$$\textbf{var}\ a0\colon rat := a\cdot\ \textbf{var}\ b0\colon rat := b\cdot\ \textbf{chan}\ d\colon rat\cdot$$

$$(t := t+1.\ Q\ d)\ \|\ ((a?\,\|\,b?).\ t := t+1.\ c!\ a0\times b + a\times b0.\ R)$$

$$R \Leftarrow (a?\,\|\,b?\,\|\,(t := max\ t\ (Td_{\,rd}+1).\ d?)).\ c!\ a0\times b + d + a\times b0.\ t := t+1.\ R$$

其中 Q 和 R 定义为：

$$Qc = \forall n\cdot Tc_{\,wc+n} = t + n$$

$$Qd = \forall n\cdot Td_{\,wd+n} = t + n$$

$$R = (\forall n\cdot Td_{\,rd+n} = t + n) \Rightarrow (\forall n\cdot\ Tc_{\,wc+n} = t + 1 + n)$$

在循环 R 内，赋值语句 $t := max\ t\ (Td_{\,rd}+1)$表示第一次循环执行的延迟为一个时间单位(因为 $t = Td_{\,rd}$)，而之后的每次循环执行的延迟时间为 0(因为 $t = Td_{\,rd}+1$)。这使得证明非常难看。为了使证明漂亮一点，用 $t := max\ (t+1)\ (Td_{\,rd}+1)$替换 $t := max\ t\ (Td_{\,rd}+1)$，并且删除调用 R 之前的 $t := t+1$。这些改变合在一起一点也不改变原来的计时，它们仅仅使得证明更为容易。赋值句 $t := max\ (t+1)\ (Td_{\,rd}+1)$显然使得时间至少加 1，所以循环体内包含一种时间增量并没有 $t := t+1$。于是加上时间的程序现在变为：

$$Q\ c \Leftarrow (a?\,\|\,b?).\ c!\ a\times b.$$

$$\textbf{var}\ a0\colon rat := a\cdot\ \textbf{var}\ b0 : rat := b\cdot\ \textbf{chan}\ d\colon rat\cdot$$

$$(t := t+1\ .\ Q\ d)\ \|\ ((a?\|\ b?).\ t := t+1.\ c!\ a0\times b + a\times b0.\ R)$$

$$R\ \Leftarrow\ (a?\|\ b?\|\ t := max\ (t+1)\ (Td_{\ rd}+1).\ d?)).\ c!\ a0\times b + d + a\times b0.\ R$$

下面是第一个精化式的证明，从右边开始。

$$(a?\|\ b?).\ c!\ a\times b.$$

$$\textbf{var}\ a0\colon rat := a\cdot\ \textbf{var}\ b0\colon rat := b\cdot\ \textbf{chan}\ d\colon rat\cdot$$

$$(t := t+1.\ Q\ d)\ \|\ ((a?\|\ b?).\ t := t+1.\ c!\ a0\times b + a\times b0.\ R)$$

可以忽略 $a?$和 $b?$，因为它们不影响计时(它们是对不出现在 $Q\ d$ 和 R 中的变量置换)。也可以忽略输出信息，而仅考虑它们所需的时间，于是也忽略变量 $a0$ 和 $b0$。

$$\Rightarrow\quad Tc_{\ wc} = t\ \wedge\ (wc := wc+1).$$

$$\exists\ Td, rd, rd', wd, wd'\cdot\ rd := 0.\ wd := 0.$$

$$(t := t+1.\ \forall n\cdot\ Td_{\ wd+n} = t+n)$$

$$\wedge\quad (t := t+1.\ Tc_{\ wc} = t\ \wedge\ (wc := wc+1).$$

$$(\forall n\cdot\ Td_{\ rd+n} = t+n) \Rightarrow (\forall n\cdot\ Tc_{\ wc+n} = t+1+n)\ )$$

对赋值句作所有的置换。

$$=\quad Tc_{\ wc} = t$$

$$\wedge\ \exists\ Td, rd, rd', wd, wd'\cdot$$

$$(\forall n\cdot\ Td_{\ n} = t+1+n)$$

$$\wedge\quad Tc_{\ wc+1} = t+1$$

$$\wedge\quad ((\forall n\cdot\ Td_{\ n} = t+1+n) \Rightarrow (\forall n\cdot\ Tc_{\ wc+2+n} = t+2+n)\ )$$

利用第一个全称量词式可去除前件。

然后抛弃这个全称量词式(弱化我们的表达式)。

现在所有的存在量词式都未被使用，因此也可被抛弃。

$$\Rightarrow\quad Tc_{\ wc} = t \wedge Tc_{\ wc+1} = t+1 \wedge \forall n\cdot\ Tc_{\ wc+2+n} = t+2+n$$

现在合并这三个合取式。

$$=\quad \forall n\cdot\ Tc_{\ wc+n} = t+n$$

$$=\quad Q\ c$$

还必须证明循环的精化。

$$(R\ \Leftarrow\ (a?\|\ b?\|\ (t := max\ (t+1)\ (Td_{\ rd}+1).\ d?)).\ c!\ a0\times b + d + a\times b0.\ R)$$

忽略 $a?$和 $b?$以及输出信息。

$$\Leftarrow\quad ((\forall n\cdot\ Td_{\ rd+n} = t+n) \Rightarrow (\forall n\cdot\ Tc_{\ wc+n} = t+1+n)\ )$$

$$\Leftarrow\quad (t := max\ (t+1)\ (Td_{\ rd}+1).\ rd := rd+1.\ Tc_{\ wc} = t \wedge (wc := wc+1).$$

$$(\forall n\cdot\ Td_{\ rd+n} = t+n) \Rightarrow (\forall n\cdot\ Tc_{\ wc+n} = t+1+n))$$

利用移动定律将第一个前件移到右边成为一个合取式。

$$=\quad (\forall n\cdot\ Tc_{\ wc+n} = t+1+n)$$

$$\Leftarrow\quad (\forall n\cdot\ Td_{\ rd+n} = t+n)$$

$$\wedge\quad (t := max\ (t+1)\ (Td_{\ rd}+1).\ rd := rd+1.\ Tc_{\ wc} = t \wedge (wc := wc+1).$$

$$(\forall n\cdot Td_{rd+n} = t+n) \Rightarrow (\forall n\cdot Tc_{wc+n} = t+1+n)\ )$$

特定化$\forall n\cdot Td_{rd+n} = t+n$，令 $n=0$,

我们用 $Td_{rd}=t$ 来简化 $max\ (t+1)\ (Td_{rd}+1)$。

$$= \quad (\forall n\cdot Tc_{wc+n} = t+1+n)$$

$$\Leftarrow \quad (\forall n\cdot Td_{rd+n} = t+n)$$

$$\wedge \quad (t := t+1.\ rd := rd+1.\ Tc_{wc} = t \wedge (wc := wc+1).$$

$$(\forall n\cdot Td_{rd+n} = t+n) \Rightarrow (\forall n\cdot Tc_{wc+n} = t+1+n)\ )$$

对赋值句作所有置换。

$$= \quad (\forall n\cdot Tc_{wc+n} = t+1+n)$$

$$\Leftarrow \quad (\forall n\cdot Td_{rd+n} = t+n)$$

$$\wedge \quad Tc_{wc} = t+1$$

$$\wedge \quad ((\forall n\cdot Td_{rd+1+n} = t+1+n) \Rightarrow (\forall n\cdot Tc_{wc+1+n} = t+2+n)\ )$$

利用合取式$\forall n\cdot Td_{rd+n} = t+n$ 可以去除前件

$\forall n\cdot\ Td_{rd+1+n} = t+1+n$，然后再丢弃它。

$$\Leftarrow \quad (\forall n\cdot Tc_{wc+n} = t+1+n)$$

$$\Leftarrow \quad Tc_{wc} = t+1 \wedge (\forall n\cdot Tc_{wc+1+n} = t+2+n)$$

$$= \quad \mathsf{T}$$

----------广播结束

----------通信结束

----------交互结束

对许多学生而言，他们被教导的对程序的第一次理解是程序如何执行。对许多学生而言，这也是他们对程序的唯一理解。在这种理解之下，检查程序是否正确的唯一可行途径就是以各种输入去执行它，测试输出结果是否正确。所有程序都必须经过测试，但测试存在两个问题。测试的第一个问题是：如何知道输出是否正确？对一些程序，例如产生漂亮图片的图形程序，判断结果是否正确的唯一方法是测试程序并判断其结果。但另一些程序是回答那些不知道答案的问题的（这也是为什么要写程序的原因），并且人们无法测试结果是否正确。在这种情况下，

应该测试答案是否至少是合理的。测试的另一个问题是：人们不可能试完所有输入。即使所有的测试实例都给出了合理的答案，但在未测试的实例中仍有可能存在错误。

如果读者将本书阅读并理解直到这个部分，那么现在对程序就有了一个与执行完全不同的理解。当证明一个程序精化了一个规范，就考虑了所有输入，也就证明了输出具有规范中所描述的性质。这比测试能达到的要多得多。但它比尝试一些输入并检查输出所花费的工作也更多。那么问题来了：什么时候值得用额外的工作负担来获得额外的正确性保证？

如果你写的程序足够简单，可以不需要任何理论就保证正确，而且即使存在一些错误也无关紧要，那么使用理论所提供的额外正确性保证就是不值得的。如果写的是一个心脏节律的控制器，或控制地铁系统的软件，或航空交通控制程序，或核能工厂的软件，或其他任何关系到人们生命的程序，那么额外的保证肯定是值得的，如果不使用理论就太草率了。

在程序完成以后来证明程序精华了一个规范是一件很艰巨的任务。在写程序时实施证明要容易得多。在编程的步骤中所需要的信息往往就是证明这一步正确的信息。额外的工作就是要将这信息形式化地写出来。这信息也是以后程序修改时所需要的信息，因此每一步将它明确写下来也可以节约后面的工作。而且如果在证明一个步骤时发现这一步是不正确的，就节省了在这个错误步骤基础上继续写后面程序的额外工作。而进一步的奖赏是，在使用理论很熟练以后，你会发现它有助于程序设计；它能对程序设计步骤作出建议。到最后，也许根本不需要额外的工作。

本书只介绍了一些小程序。但理论并不只局限于小程序；它和规模大小无关，可用于任何规格的软件。在大的软件项目中，第一个设计决定可能是把一个任务分成以某些方式能合并在一起的几小块。这个决定可写成一个精化，这个精化准确表明这些小部分是什么，它们如何合并到一起，然后这个精化可被证明。运用早期阶段的理论会十分有利，因为如果一个早期步骤是错的，后面再改正它代价就十分高了。

对于一个在工业程序设计中广泛使用的程序设计理论，它必须有工具的支持。理想化的情况是，一个自动证明器对精化进行检查，在精化正确时保持沉默，而每当有错误时能进行精确的报告。目前，有一些工具提供一定的帮助，但远远没有达到理想化。工具的构造尚有许多机会，并且它需要对程序设计的实用理论有完备的知识储备。

第十章 练习

带有标记 √ 的练习已在前面的章节中做过。练习的答案参见以下链接：
www.cs.utoronto.ca/~hehner/aPToP/solutions

10.0 介绍

0 桌上有四张牌。你看见牌上的标记分别为D，E，2和3（每张牌上一个标记）。已知每张牌的一面是字母另一面是数字。你需要翻开哪张或哪些牌来确定是否每张一面是D的牌另一面是3？为什么？

1 杰克正在看安妮。安妮正在看乔治。杰克已经结婚了。乔治是单身。是否一个已婚人士正在看一个单身人士？（是）（不是）（不能确定）

2 这里有三句陈述。
(i) 恰好三句陈述中的一句是错的。
(ii) 恰好三句陈述中的两句是错的。
(iii) 这三句陈述全是错的。
这三句陈述中哪些是对的，哪些是错的？

3 这里有四句陈述。
(i) 这个陈述是对的。
(ii) 这个陈述是错的。
(iii) 这个陈述可能是对的也可能是错的。
(iv) 这个陈述既是对的又是错的。
这几句陈述中哪些是
(a) 对的？
(b) 错的？
(c) 也许对也许错（不能确定，不完备）？
(d) 既是对的又是错的（过定的，不一致）？

----------初步介绍结束

10.1 基本理论

4 真值表和计算规则可以被一个新的证明规则和一些新的公理替代。这个新的证明规则为：“当把一个二元表达式中的一个定理用另一个定理替换后，该二元表达式并不得到、失去、或改变它的分类。类似地，当把一个二元表达式中的一个反定理用另一个反定理替换后，该布尔表达式也不得到、失去、或改变它的分类”。真值表变成了新的公理；例

如，真值表的一个表项变为公理 $T \vee T$，而另一个表项变为公理 $T \vee \bot$。这两个公理可通过引入一个变量简化为一个公理：$T \vee x$。将真值表写成尽可能简单的公理和反公理。

5 化简下面的二元表达式。

(a) $x \wedge \neg x$

(b) $x \vee \neg x$

(c) $x \Rightarrow \neg x$

(d) $x \Leftarrow \neg x$

(e) $x = \neg x$

(f) $x \neq \neg x$

6 利用 1.0.1 小节介绍的证明格式和 11.4 节给出的定律，证明以下二元理论定律。不要使用完备性规则。

(a) $a \wedge b \Rightarrow a \vee b$

(b) $(a \wedge b) \vee (b \wedge c) \vee (a \wedge c) = (a \vee b) \wedge (b \vee c) \wedge (a \vee c)$

(c) $\neg a \Rightarrow (a \Rightarrow b)$

(d) $a = (b \Rightarrow a) = a \vee b$

(e) $a = (a \Rightarrow b) = a \wedge b$

(f) $(a \Rightarrow c) \wedge (b \Rightarrow \neg c) \Rightarrow \neg(a \wedge b)$

(g) $a \wedge \neg b \Rightarrow a \vee b$

(h) $(a \Rightarrow b) \wedge (c \Rightarrow d) \wedge (a \vee c) \Rightarrow (b \vee d)$

(i) $a \wedge \neg a \Rightarrow b$

(j) $(a \Rightarrow b) \vee (b \Rightarrow a)$

(k) √ $\neg(a \wedge \neg(a \vee b)$

(l) $(\neg a \Rightarrow \neg b) \wedge (a \neq b) \vee (a \wedge c \Rightarrow b \wedge c)$

(m) $(a \Rightarrow \neg a) \Rightarrow \neg a$

(n) $(a \Rightarrow b) \wedge (\neg a \Rightarrow b) = b$

(o) $(a \Rightarrow b) \Rightarrow a = a$

(p) $a=b \vee a=c \vee b=c$

(q) $a \wedge b \vee a \wedge \neg b = a$

(r) $a \Rightarrow (b \Rightarrow a)$

(s) $a \Rightarrow a \wedge b = a \Rightarrow b = a \vee b \Rightarrow b$

(t) $(a \Rightarrow a \wedge b) \vee (b \Rightarrow a \wedge b)$

(u) $(a \Rightarrow (p=x)) \wedge (\neg a \Rightarrow p) = p=(x \vee \neg a)$

7 利用 1.0.1 小节介绍的证明格式和 11.4 节给出的定律，证明以下二元理论定律。不要使用完备性规则。

(a) **if** a **then** a **else** $\neg a$ **fi**

(b) **if** b **then** c **else** $\neg c$ **fi** = **if** c **then** b **else** $\neg b$ **fi**

(c) **if** $b \wedge c$ **then** P **else** Q **fi** = **if** b **then if** c **then** P **else** Q **fi else** Q **fi**

(d) **if** $b \vee c$ **then** P **else** Q **fi** = **if** b **then** P **else if** c **then** P **else** Q **fi fi**

(e) **if** b **then** P **else if** b **then** Q **else** R **fi fi** = **if** b **then** P **else** R **fi**

(f) **if if** b **then** c **else** d **fi then** P **else** Q **fi**

= **if** b **then if** c **then** P **else** Q **fi else if** d **then** P **else** Q **fi fi**

(g) **if** b **then if** c **then** P **else** R **fi else if** c **then** Q **else** R **fi fi**

= **if** c **then if** b **then** P **else** Q **fi else** R **fi**

(h) **if** b **then if** c **then** P **else** R **fi else if** d **then** Q **else** R **fi fi**

= **if if** b **then** c **else** d **fi then if** b **then** P **else** Q **fi else** R **fi**

8 形式化地描述“三个语句中恰好一个为真”。

9 实例分析定律把三元运算符 **if** a **then** b **else** c **fi** 等同于仅含一元运算符和二元运算符的表达式。表达式中，变量 a 出现两次。寻找一个等同的仅含一元运算符和二元运算符的表达式，其中变量 a 只出现一次。提示：使用连续运算符。

10 考虑一个完全括号化的表达式，仅包含任意数量的符号 T ⊥ = ≠ （），以任意语法可接受的顺序组成。

(a) 说明所有语法可接受顺序的重新整理都是等价的。

(b) 说明做任意偶数个以下符号的替换：T 替换 ⊥，⊥ 替换 T，= 替换 ≠，≠ 替换 =，所得到的任意表达式都是等价的。

11 (对偶) 如果一个运算符作用在否定的运算对象上得到的结果是另一个运算符作用的结果的否定，那么该运算符是另一个运算符的对偶。例如零元运算符 T 和 ⊥ 互为对偶。如果 $op0(\neg a) = \neg(op1a)$，那么 $op0$ 和 $op1$ 互为对偶。如果 $(\neg a)\ op0\ (\neg b) = \neg(a\ op1\ b)$，那么 $op0$ 和 $op1$ 互为对偶。以此类推。

(a) 在 4 个一元布尔运算符中，有一对对偶，还有两个运算符是其自己的对偶。找出它们。

(b) 在 16 个二元布尔运算符中，有 6 对对偶，4 个运算符为其自己的对偶。找出它们。

(c) 三元运算符 **if then else fi** 的对偶是什么？请仅使用运算符 **if then else fi** 来描述。

(d) 不含变量的布尔表达式的对偶是这样形成的：将每个运算符用其对偶运算符来替代，为了维持优先顺序可以加入括号。解释为什么定理的对偶是反定理，反之亦然。

(e) 令 P 是一个不含变量的布尔（二元）表达式。从(d)我们知道每个以下形式的不含变量的布尔表达式

$$(P \text{ 的对偶}) = \neg P$$

是一个定理。因此，为了寻找含变量的布尔表达式的对偶，我们必须对每个变量求否定并将每个运算符替换为其对偶。例如，如果 a 和 b 是布尔表达式，那么 $a \wedge b$ 的对偶是 $\neg a \vee \neg b$。因为

$$(a \wedge b \text{ 的对偶}) = \neg(a \wedge b)$$

我们有了一个对偶定律：

$$\neg a \vee \neg b = \neg(a \wedge b)$$

另一个对偶定律可以通过将对偶等同于 $a \vee b$ 的否定。请通过将对偶等同于否定获

得本书中没出现的五个定律。

(f) 对偶运算符的真值表是互相的垂直镜面反射。例如 ∧ 的真值表（下图左）是 ∨ 的真表（下图右）的垂直镜面反射。

∧:		
T	T	T
T	⊥	⊥
⊥	T	⊥
⊥	⊥	⊥

∨:		
T	T	T
T	⊥	T
⊥	T	T
⊥	⊥	⊥

根据下面的标准对 4 个一元和 16 个二元布尔运算符设计符号（必要时也可以对已存在的符号重新设计）。

(i) 对偶运算符应该有相互垂直镜面反射的符号（像 ∧ 和 ∨ ）。这意味着自对偶的运算符有垂直对称符号，而其他运算符有垂直不对称符号。

(ii) 如果 $a\ op0\ b = b\ op1\ a$，那么 $op0$ 和 $op1$ 会有水平镜面反射的符号（像 ⇒ 和 ⇐ ）。这意味着对称运算符有水平对称符号，其它运算符有水平不对称符号。

12 设计 10 个在第 1 章中没出现过的二元布尔运算符，并找出关于它们的运算定律。

13 形式化以下的语句。对每一对语句，证明它们等价或不等价。

(a) 不要饮酒并开车。

(b) 如果饮酒，不要开车。

(c) 如果开车，不要饮酒。

(d) 不要饮酒，并不要开车。

(e) 不要饮酒，或不要开车。

14 完成下列布尔理论定律：

(a) T =

(b) ⊥ =

(c) ¬*a* =

(d) *a* ∧ *b* =

(e) *a* ∨ *b* =

(f) *a* = *b* =

(g) *a* ≠ *b* =

(h) *a* ⇒ *b* =

通过增加一个只使用下列符号的右式(任意数量)

(i) ¬ ∧ *a b* ( )

(ii) ¬ ∨ *a b* ( )

(iii) ¬ ⇒ *a b* ( )

(iv) ≠ ⇒ *a b* ( )

(v) ¬ **if then else fi** *a b* ( )

总共是 8×5=40 个问题。

15 (BDD) 一个 BDD（二叉决策树）是一个有下列三种形式的二元表达式：T，⊥，**if** 变量

then BDD **else** BDD **fi**。例如，

if *x* **then if** *a* **then** ⊤ **else** ⊥ **fi else if** *y* **then if** *b* **then** ⊤ **else** ⊥ **fi else** ⊥ **fi fi**

是一棵 BDD。一棵 OBDD（有序的 BDD）是一棵变量为有序的 BDD，且在每个 **if then else fi** 中，**if** 部分的变量必须在 **then** 和 **else** 的变量之前（“前”是按照序而言的）。例如，对变量使用字母表顺序，前面的例子不是 OBDD，但

if *a* **then if** *c* **then** ⊤ **else** ⊥ **fi else if** *b* **then if** *c* **then** ⊤ **else** ⊥ **fi else** ⊥ **fi fi**

是 OBDD。一棵 LBDD（标记的 BDD）是以下三种形式的定义集合：

标记 = ⊤

标记 = ⊥

标记 = **if** 变量 **then** 标记 **else** 标记 **fi**

标记是与变量不同的；每个在 **then** 部分和 **else** 部分的标记必须被以上定义之一进行定义；恰好只有一个标记必须被定义但不被用。下面是一棵 LBDD。

true = ⊤

false = ⊥

alice = **if** *b* **then** true **else** false **fi**

bob = **if** *a* **then** alice **else** false **fi**

一棵 LOBDD 是一棵扩展标记后成为 OBDD 的 LBDD。排序阻止了标记的递归使用。前面的例子是 LOBDD。一棵 RBDD（缩减 BDD）是一棵在每个 **if then else fi** 中，**then** 部分和 **else** 部分不同的 BDD。一棵 RLOBDD 是缩减的，标记的和有序的。前面的例子是一棵 RLOBDD。

(a) 用 BDD 表示 $\neg a$，$a \wedge b$，$a \vee b$，$a \Rightarrow b$，$a = b$，$a \neq b$ 和 **if** *a* **then** *b* **else** *c* **fi**。

(b) 如何连接两棵 OBDD 得到一棵 OBDD？

(c) 如何判断两棵 RLOBDD 是否相等？

(d) 为了有效决定对变量的赋值是否满足它（解答它，得到值 ⊤），如何表示一棵 RLOBDD？

16 有 256 个运算符，这些运算符有 3 个布尔（二元）操作元和一个布尔结果。其中有多少个运算符是退化的？一个运算符是退化的如果它的结果没有用全部操作元来表示。

17 将以下语句用二元表达式表示。先从尽量靠近语言习惯的描述开始，然后尽可能地简化（有时没有简化的可能）。对那些不能使用二元运算符的部分你可能需要引入新的基本二元表达式，像（门可以被打开），但对于象“仅当”这样的词应该使用二元运算符。将单词的含义转换为二元符号；单词的含义可能依赖于上下文甚至未明确说明的事实。形式化不是对单词使用的符号进行简单替换。

(a) 只有电梯停止了门才可以打开。

(b) 除非电梯门和楼层的门同时打开，否则电梯门和楼层的门都不能打开。

(c) 要不就是马达被堵塞，要不就是控制坏了。

(d) 灯要么是开的要么是关的。

(e) 如果你按下按纽，电梯会来。

(f) 如果电源开关是开的，系统正在运行。

(g) 有烟的地方就有火，因为没有烟，所以没有火。

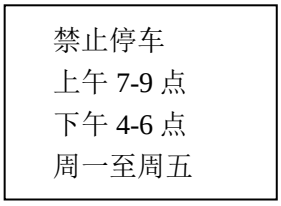
(h) 有烟的地方就有火，因为没有火，所以没有烟。

(i) 如果没射击就不得分。
(j) 只有你有钥匙的时候才能打开门。
(k) 没有痛苦就没有收获。
(l) 没有衬衣？没有鞋？没有服务！
(m) 如果它发生了，它就发生了。

18 形式化并证明下面的语句。如果下雨，简没有带伞，那么她会淋湿。正在下雨，简没有淋湿。因此她带了伞。

19 (圣诞老人) 有人告诉你："如果这句话是对的，那么圣诞老人存在。"你能从中得出什么结论？

20 有一个标志：



使用变量 t 表示一天中的时间，d 表示一周中的一天，写一个二元表达式表示什么时候禁止停车。

21 (女佣和男佣) 女佣说她看见男佣在起居室里。起居室在厨房的旁边。而枪击就发生在厨房，并且一定可以在厨房的所有邻近房间内听到。男佣的听力很好，但他说他没有听到枪声。根据以上事实，试证明有一个人在说谎。可使用以下的缩简符号：

mtt = (女佣说真话)
btt = (男佣说真话)
blr = (男佣在起居室内)
bnk = (男佣在厨房的附近)
bhs = (男佣听到了枪声)

22 (网球) 网球杂志上有这样一则广告："如果我不正在打网球，我就正在看网球。并且如果我不正在看网球，我就正在读网球杂志。"。假设说话者不可以同时做两件以上的上述活动：

(a) 证明说话者不正在读网球杂志。
(b) 试问说话者正在干什么?

23 令 p 和 q 为二元表达式，假设 p 即是定理又是反定理(该理论不一致)，

(a) 利用已提出的证明规则，证明 q 既是定理又是反定理。
(b) q=q 是定理还是反定理？

24 (首饰盒) 有两个首饰盒，一个是金的，一个是银的。在其中之一有一百万美元，而另一个有一块木炭。金首饰盒上的题词是：钱不在这里。银首饰盒上的题词是：恰好只有一句题词是真的。每句题词或者为真或者为假（不同时为二者）。基于这些题词，选择一个首饰盒。

25 (意外的鸡蛋) 有两个盒子：一个红色的，一个蓝色的。其中一个装有鸡蛋，另一个是空的。你将首先检查红盒子找寻鸡蛋，如果需要, 再检查蓝盒子。但是除非打开盒子看到鸡蛋，你无法知道究竟哪只盒子装有鸡蛋。你的推理如下：“如果我看到红盒子里没有鸡蛋，那么我不需要打开蓝盒子就可以知道鸡蛋在蓝盒子中。但是，我被告知，我无法知道鸡蛋在哪一个盒子里除非我打开盒子看到它。因此，鸡蛋不可能在蓝盒子里。现在我不需要打开红盒子就知道鸡蛋一定在里面，但这又与我被告知的规则相违背，因此鸡蛋不在任何一个盒子里。” 把两个盒子都排除掉之后，你打开了两个盒子，却意外地发现鸡蛋在其中一个盒子里，与原来所说的一样。请形式化所给的命题和推理，并解释这个悖论。

26 (骑士和恶棍) 在一个小岛上住着三个居民 P，Q 和 R。每个人不是骑士就是恶棍。骑士总是说真话，恶棍总是说谎话。形式化描述以下提供的每条信息，回答问题并证明你的结论。

(a) P 说：“如果我是骑士，我将把我的帽子吃掉。”。试问 P 会吃掉他的帽子吗？

(b) P 说：“如果 Q 是骑士，那么我就是恶棍。”。试问 P 和 Q 分别是什么？

(c) P 说：“当且仅当我是骑士时，这个岛上有金子”。试问可以确定 P 是骑士或恶棍吗？可以确定这个岛上有金子吗？

(d) P，Q 和 R 站在一起，你问 P：“你是骑士还是恶棍？”P 咕哝着他的回答而你没有听见。因此你转而问 Q：“P 说了些什么？”。Q 回答：“P 说他是恶棍。”。接着 R 说：“别相信 Q，他在说谎。”。试问 Q 和 R 分别是什么？

(e) 你问 P：“你们之中有多少是骑士？”，P 咕哝，所以 Q 说：“P 说我们之中只有一个是骑士。”。R 说：“别相信 Q，他在说谎。”。试问 Q 和 R 分别是什么？

(f) P 说：“我们都是恶棍。”。Q 说：“不，我们中间有一个是骑士。”。试问 P，Q 和 R 分别是什么？

(g) P 说 Q 和 R 是一样的（都是恶棍或都是骑士）。有人问 R 是否 P 和 Q 是一样的。R 的回答会是什么？

(h) P, Q 和 R 都说：“另外两个人是骑士”。试问有多少个骑士？

27 （金银岛）岛 X 和 Y 上的骑士总说真话，恶棍总说假话，可能有一些普通人有时说真话有时说假话。至少有一个岛上有黄金，大家都知道在哪个岛上。你从埋藏黄金的海盗那里得到一个信息，信息上有这样的线索（我们将它作为公理）：“只要这些岛上有任何普通人，那么两个岛上都有黄金。”。你只允许在一个岛上挖黄金，并且只允许问随机的一个人一个问题。为了确定在哪个岛挖黄金你应该问什么？

28 有三个人站在一排，名叫前面，中间，和后面。每个人都戴着一顶帽子，或红或蓝。后面可以看到中间和前面的帽子颜色，但是看不到自己的帽子。中间可以看到前面的帽子

颜色，但看不到后面和他自己的帽子。前面看不到任何一个人的帽子颜色。路人甲告诉他们，至少有一个人戴了顶红帽子。然后后面说：我不知道我戴着什么颜色的帽子。然后中间说：我不知道我戴着什么颜色的帽子。然后前面说：我不知道我戴着什么颜色的帽子。格式化名为前面的人的推理，并得出前面戴着什么颜色的帽子。

29 (括号代数) 有一个新的写二元表达式的方法。一个二元表达式可以是空的；换句话说，表达式中没有任何东西。如果你给一个表达式加上一对括号，你就得到另一个表达式。如果你将两个表达式挨着放在一起，你就得到另一个表达式。例如，

$$()(())((())())$$

是一个表达式。在括号代数中空表达式代表着 T；表示表达式的否定的括号代数方式是将括号放在表达式的两边，表示两个表达式的连接的括号代数方式是将表达式挨着放在一起。因此上例是如下的括号代数形式。

$$\neg \mathsf{T} \wedge \neg\neg \mathsf{T} \wedge \neg(\neg\neg \mathsf{T} \wedge \neg \mathsf{T})$$

我们也可以在括号表达式的任何地方有变量。括号代数有三条规则。如果 x，y 和 z 是括号表达式，那么

$((x))$	可以替换或被替换为	x	双重否定规则
$x()y$	可以替换或被替换为	$()$	基规则
xyz	可以替换或被替换为	$x'yz'$	上下文规则

其中 x'是 x 中增加或删除若干个 y，同样 z'是 z 中增加或删除若干个 y。上下文规则中没有说明增加或删除了多少个 y；可以是任何个数，从零到全部。证明时只用遵守规则直到表达式消失。例如，

	$((a)b((a)b))$	上下文规则：x 是空，y 是 $(a)b$，z 是 $((a)b)$
变成	$((a)b(\quad))$	基规则：x 是 $(a)b$，y 是空
变成	$(\quad(\quad))$	双重否定规则：x 是空
变成		

因为最后的表达式为空，因此整个表达式被证明。

(a) 将下面二元表达式重写为括号表达式，然后使用括号代数的规则证明之。

$$\neg(\neg(a \wedge b) \wedge \neg(\neg a \wedge b) \wedge \neg(a \wedge \neg b) \wedge \neg(\neg a \wedge \neg b))$$

(b) 尽可能直接地将下面二元表达式重写为括号表达式，然后使用括号代数的规则证明之。

$$(\neg a \Rightarrow \neg b) \wedge (a \neq b) \vee (a \wedge c \Rightarrow b \wedge c)$$

(c) 是否所有的二元表达式都能合理地重写为括号表达式？

(d) 使用括号表达式的规则，xy 能变成 yx 吗？

(e) 二元表达式中的所有定理都能合理地重写为括号表达式，并使用括号代数的规则证明吗？

(f) 我们将 T 解释为空，括号解释为否定，并列解释为连接。还有其他一致的方法来解释括号代数中的符号与规则吗？

30 形式化表示：

(a) 实数 x 的绝对值。

(b) 实数 x 的符号，为-1，0 或+1，取决于 x 是否为负的，零或正的。

31 （天堂之门）有一个门，可到达天堂或地狱。有个知道门通往哪里的守卫。如果你问守

卫一个问题，守卫可能回答真话，也可能说谎。想出一个问守卫的问题，来确认门是通往天堂还是地狱。

32 一个数可以写成一个十进制数字的序列。试想使用任意表达式的序列记号来表示，而不只是数字。例如，$1(2+3)4 = 154$。那么需要在数公理上进行哪些改变？

33 (尺度) 在程序设计语言中有一个传统，就是在有限的数字序列中使用尺度操作符 e。于是 $12e3=12\times10^3$。考虑在任意的表达式中使用尺度符号，而不只用于数字串。例如，$(6+6)e(5-2)=12e3$。试问对数公理应作哪些改变？

34 在定义数表达式时，包含了诸如$(-1)^{1/2}$ 的复数，并不是因为我们特□需要它们，而是由于包含它们后可使数表达式理论变得容易一些。如果对复数感□趣的话，将发现 11.4.2 节中提供的数公理无法证明许多希望证明的东西。例如，不能证明$(-1)^{1/2}\times 0 = 0$。如何□化公理使之能□证明有关复数的结论，但又能弱到可以给∞留足余地?

35 求证：$-\infty<y<\infty \wedge y\neq 0 \Rightarrow (x/y = z = x=z\times y)$ 。

36 试说明当加入公理：

$$-\infty<y<\infty \Rightarrow x/y\times y = x$$

时，数公理就变得不一致了。

37 (循环数) 为广义数系统重新设计公理，使得数可以循环，并有$+\infty = -\infty$ 。请注意符号 < 的传递性。

38 在数论中加入公理 0/0=5 会有害处吗？

39 令 • 为一个二元中缀运算符 (优先级为 3)，其操作元和结果都为某个类型 *T*。令 ◊ 为一个二元中缀运算符(优先级为 7)，其操作元为类型 *T*，但结果为二元类型，并定义如下：

$$a \lozenge b = a \bullet b = a$$

(a) 证明：如果 • 满足幂等律，那么 ◊ 具有自反性。
(b) 证明：如果 • 满足结合律，那么 ◊ 具有传递性。
(c) 证明：如果 • 具有对称性，那么 ◊ 具有反对称性。
(d) 如果 *T* 是二元类型，并且 • 是 ∧，那么 ◊ 是什么？
(e) 如果 *T* 是二元类型，并且 • 是∨，那么 ◊ 是什么？
(f) 如果 *T* 是自然数类型，并且 ◊ 是 ≤，那么 • 是什么？
(g) 以上公理是根据 • 来定义 ◊，请问该定义是否可逆转，使得 • 是根据 ◊ 来定义的？

40 (家族理论)定义一个人际关系理论。创建关于人的表达式，诸如：*Jack, Jill, p* 的父亲， *p* 的母亲。利用这些关于人的表达式，创建诸如：*p* 是男人， *p* 是女人，*p* 是 *q* 的父母，*p* 是 *q* 的儿子，*p* 是 *q* 的女儿，*p* 是 *q* 的孩子， *p* 与 *q* 是夫妻， *p=q* 的二元表达式。创建形如(*p* 是男人) ≠ (*p* 是女人)的公理。 形式化一个有趣的定理，并证明它。

10.2 基本数据结构

41 化简下列各式：

(a) (1, 7–3) + 4 – (2, 6, 8)

(b) *nat* × *nat*

(c) *nat* – *nat*

(d) (*nat*+1) × (*nat*+1)

42 证明¬ 7 : *null*。

43 我们用公理 *null* : *A* 定义了束 *null*，如果用公理 *A* : *all* 定义束 *all* 会有什么害处？

44 令 *A* 为一个二元值的束，满足 *A* = ¬*A*。试问 *A* 是什么？

45 令*x*为一个元素，*A*为任意束。存在定律¬*x*: *A* ⇒ ¢(*A*‘*x*)=0。证明其反向定律 ¬*x*: *A* ⇐ ¢(*A*‘*x*)=0。

46 试说明列在 2.0 节中的某些束论公理可以用其他一些公理证明出来。在不丢失任何定理的情况下，试问可省略多少公理？

47 (超束)超束 (hyperbunch) 与束类似，只是每一个元素可以出现多次，而不只为 0 次(不出现)或 1 次(出现)。元素的次序无关紧要。(超束没有特征谓词，但有一个结果为数的特征函数。)试为以下每种超束设计记号和公理。

(a) 多束(multibunch) ：一个元素可以出现任意自然数次。例如，一个多束可以包含一个 2，两个 7，三个 5，和零个任意其他的数。(注意：相应的集合，或者称为多集合，或者称为包。)

(b) 整束(wholebunch)：一个元素可以出现任意整数次。

(c) 模糊束(fuzzybunch)：一个元素可以出现位于 0 和 1 之间的任意实数次。

48 令⊗为一个二元中缀运算符 (优先级为 3)，其操作元为自然数，结果为广义自然数。非正式地，*n*⊗*m* 表示：“*n* 为 *m* 的因子的次数 ”。它可以由以下两个公理定义：

m: *n*×*nat* ∨ *n*⊗*m* = 0

n≠0 ⇒ *n*⊗ (*m*×*n*) = *n*⊗*m*+1

(a) 制作一个关于(0,..3)⊗(0,..3)值的 3×3 图表。

(b) 说明如果去除第二个公理的前件，公理将变得不一致。

(c) 如何修改公理才能允许⊗的操作元为广义自然数。

49 对自然数 *n* 和 *m*，我们可以形式化地描述命题 “*n* 是 *m* 的因子”如下：

$$m : n \times nat$$

(a) 0 的因子是什么？

(b) 0 是什么数的因子？

(c) 1 的因子是什么？

(d) 1 是什么数的因子？

50 复合数是一个具有两个或两个以上（没必要是不同的）质数因子的自然数。试尽可能简单地表示出这个复合数。

51 令 $B = 1,3,5$。下面各式分□是什么？

(a) $\mathbb{C}\ (B + B)$

(b) $\mathbb{C}\ (B \times 2)$

(c) $\mathbb{C}\ (B \times B)$

(d) $\mathbb{C}\ (B^2)$

52 复合公理为：

$$x{:}\ A\ ,\ B = x{:}\ A \vee x{:}\ B$$

在这个公理中，∨ 运算符可以被 16 个二元布尔运算符替换, 只要将束的合并运算符 (,) 用相应的束运算符替换就行。试问这 16 个布尔运算符中，有哪些与有用的束运算符相对应?

53 (冯 • 诺依曼数)

(a) 增加以下两个公理后有什么害处么？

$0 = \{null\}$	空集
$n+1 = \{n, \sim n\}$	对每一个自然数 n

(b) 这些公理在算术运算和集合运算之间引入了何种对应关系？

(c) 加入以下两个公理后有什么害处么？

$0 = \{null\}$	空集
$i+1 = \{i, \sim i\}$	对每一个整数 i

54 (康托的天国)说明 $\mathbb{C}iB > \mathbb{C}B$ 不是一个定理，也不是一个反定理。

55 定义在 2.2 节中的字符串, 其长度和索引都为自然数。添加一个表示连接的相反的运算符，得到的字符串的长度和索引为负数。

56 (前缀次序) 给出公理定义字符串的前缀偏序关系。当且仅当串 S 是串 T 的初始段, 这个次序中 S 在 T 之前。

57 在节 2.2 中，有一个自描述的表达式：

“““$_{0;0;(0;..29);28;28;(1;..28)}$”””$_{\ 0;0;(0;..29);28;28;(1;..28)}$

此表达式评估为它自己的表示。

(a) 写一个评估为其自身表示两倍的表达式。换句话说，它评估为自身的表示，并同样跟随着其自身的表示。

(b) 使它成为自打印的程序。假设！*e*打印出表达式*e*的值。

58 假设添加定律使得不同的运算符分布于字符串的连接（分号）。例如，如果 *i* 和 *j* 是项，*s* 和 *t* 是字符串，那么定律

nil+*nil* = *nil*

(*i*; *s*)+(*j*; *t*) = *i*+*j*; *s*+*t*

表示字符串是项加项（一个字符串的总和是一个有关总和的字符串）。例如,

(2; 4; 7)+(3; 9; 1) = 5; 13; 8

什么样的字符串 *f* 被定义为

$f = 0; 1; f + f_{1;..\infty}$

59 化简下列各式（不需证明）

(a) *null* , *nil*

(b) *null* ; *nil*

(c) **nil*

(d) [*null*]

(e) [**null*]

60 [0,1,2]和[0;1;2]有何差别?

61 化简下式, 假设 *i*: 0,..#*L*

(a) *i*→*Li* | *L*

(b) *L*[0;..*i*] + [*x*] + *L*[*i*+1;..#*L*]

62 化简下列各式（不需证明）

(a) 0→1 | 1→2 | 2→3 | 3→4 | 4→5 | [0;..5]

(b) (4→2 | [–3;..3]) 3

(c) ((3;2) →[10;..15] | 3→[5;..10] | [0;..5])3

(d) ([0;..5][3;4])1

(e) (2;2) → “j” | [["abc"];["de"];["fghi"]]

(f) #[*nat*]

(g) #[*3]

(h) [3;4]: [3*4**int*]

(i) [3;4]: [3;*int*]

(j) [3,4;5]: [2**int*]

(k) [(3;4);5]: [2**int*]

(l) [3;(4,5);6;(7,8,9)] ‘ [3;4;(5,6);(7,8)]

63 令 L 为一张表, 令 i 和 j 为 L 的索引。试不用符号 |表示 $i \to Lj \mid j \to Li \mid L$。

----------基本数据结构结束

10.3 函数理论

64 在以下各式中, 替换 p 为：

$\langle x\colon int \to \langle y\colon int \to \langle z\colon int \to x \geq 0 \wedge x^2 \leq y \wedge \forall z\colon int \cdot z^2 \leq y \Rightarrow z \leq x \rangle\rangle\rangle$

并化简。假设 $x, y, z, u, w\colon int$。

(a) $p\ (x+y)(2 \times u + w)\ z$

(b) $p\ (x+y)(2 \times u + w)$

(c) $p\ (x+z)(y+y)(2+z)$

65 一些数学家喜欢用记号 $\exists! x : D \cdot Px$ 表示“ D 中存在唯一的 x 使 Px 成立”。形式化地定义 $\exists!$。

66 不使用 § 表示下面各式。

(a) $\not\subset (\S\ x : D \cdot Px) = 0$

(b) $\not\subset (\S\ x : D \cdot Px) = 1$

(c) $\not\subset (\S\ x : D \cdot Px) = 2$

67 简化下列各式，无需证明。

(a) $\S n\colon nat \cdot \exists m\colon nat \cdot n = m^2$

(b) $\S n\colon nat \cdot \exists m\colon nat \cdot n^2 = m \Rightarrow n = m^2$

68 （连接）定义函数 cat ，它应用于一个全是表的表上得到他们的连接。例如，

$cat[[0;1;2];[nil];[[3]];[4;5]] = [0;1;2;[3];4;5]$

69 形式化地表示 L 是 M 的子表（不一定是连续的项）。例如， $[0;2;1]$ 是 $[0;1;2;2;1;0]$ 的子表，但 $[2;0;1]$ 不是 $[0;1;2;2;1;0]$ 的子表。

70 形式化地表示 L 是 M 的最长已排序的子表，其中

(a) 子表必须是连续的项(段)；

(b) 子表必须是连续的（段）和非空的；

(c) 子表包含的项的次序与它们在 M 中的出现次序一样，但并不一定是连续的（不一定是段）。

71 试形式化地表示 n 为表 L 的最长回文段的长度。一个回文段是这样一个表，其表项的正序排列与它的倒序排列相等。

72 使用语法结构 x 能在时刻 t 欺骗 y 来形式化以下命题。

(a) 你一直可以欺骗某些人。

(b) 你有时可以欺骗所有的人。

(c) 你不能一直欺骗所有的人。

其中“你”可以分别解释成以下含义：

(i) 某人

(ii) 任意一个人

(iii) 正在与我讲话的人

这是 3×3=9 个问题。

73 (侦探小说) 以下有 10 个命题。

(i) 一些歹徒抢劫了罗素公寓。

(ii) 抢劫了罗素公寓的人或者与公寓的佣人合谋，或者是破门而入。

(iii) 若要进入公寓, 或者砸破门或者撬锁。

(iv) 只有专业锁匠才可撬锁。

(v) 无论是谁, 只要砸破门, 就会被听见。

(vi) 没有一个人被听见。

(vii) 没有一个人可以抢劫罗素大厦，除非他欺骗了守卫。

(viii) 如果要欺骗守卫，他必须是一个令人信服的演员。

(ix) 没有一个罪犯可以既是专业锁匠又是令人信服的演员。

(x) 某个罪犯在佣人中有同谋。

(a) 选择一些好的缩略词, 将每一个命题翻译成形式逻辑。

(b) 将前 9 个命题作为公理, 求证第 10 个。

74 (元数) 函数的元数是它引入的变量(参数)个数和它可应用于的变元（arguments）的个数。写一个公理，形式化地定义αf (f的元数)。

75 有一些人，一些钥匙和一些门。令 p *holds* k 表示人物 p 有钥匙 k 。令 p *unlocks* d 表示钥匙 k 可以开门 d 。令 p *opens* d 表示人物 p 可以开门 d 。形式化表示下列各句。

(a) 如果有合适的钥匙，任何人可以打开任何门。

(b) 至少有一扇门可以（被任何人）不用钥匙打开。

(c) 专业锁匠即使没有钥匙也能打开任何门。

76 证明如果变量 i 和 j 不在谓词 P 和 Q 中出现, 那么有

$$(\forall i \cdot Pi) \Rightarrow (\exists i \cdot Qi) = (\exists i, j \cdot Pi \Rightarrow Qj)$$

77 有四个二元、结合、对称、并带有一个身份（identity）的布尔运算符。我们使用了其中两个来定义量词。另外两个会发生何种情形?

78 哪一个运算符可以用来定义一个量词以表示一个函数的作用域?

79 利用一个结合、对称、并带有一个身份的运算符，我们定义了几个量词。束的合并也

是这样的一个运算符，试问利用它是否也生成一个量词？

80 练习 13 谈到了饮酒和驾驶，但没有涉及时间。先饮酒然后马上驾驶是不对的，但先驾驶然后马上饮酒是可以的。先饮酒，6 小时后再驾驶也是可以的。令 *drink* 和 *drive* 是时间的谓词，形式化表示不能在饮酒后 6 个小时内驾驶的规则。你的规则如何评判同时饮酒和驾驶？

81 使用二元表达式表示下面语句。

(a) 每个人有时会爱上某个人。

(b) 每 10 分钟纽约某个人遭抢劫。

(c) 每 10 分钟某个人试图联系你。

(d) 每当高度低于 1000 英尺时，降落轮必须放下。

(e) 如果之前不见你，我将在周二见你。

(f) 没有什么消息是好消息。

(g) 我不同意你说的任意的话。

(h) 我不同意你说的所有话。

82 （爱）形式化下面命题。

所有人都爱我的孩子。

我的孩子只爱我。

我是我的孩子。

并证明首两个命题暗示了最后一个命题。

83 （饮酒）酒吧里有一些人。形式化并证明命题“有这样一个人，如果他喝酒，那么酒吧里所有人都喝酒。”

84 化简（所有定义域都是 *nat*）。

(a) $\forall y\cdot y=x+2 \Rightarrow y>5$

(b) $\forall y\cdot y=x+2 \vee y=x+1 \Rightarrow y>5$

85 证明$((\exists x\cdot Px)\Rightarrow(\forall x\cdot Rx\Rightarrow Qx))\wedge(\exists x\cdot Px\vee Qx)\wedge(\forall x\cdot Qx\Rightarrow Px)\Rightarrow(\forall x\cdot Rx\Rightarrow Px)$。

86(a) 如果P: $bin\to bin$是单调的，证明$(\exists x\cdot Px) = P\top$ 和 $(\forall x\cdot Px) = P\bot$。

(b) 如果P: $bin\to bin$是反单调的，证明$(\exists x\cdot Px) = P\bot$和$(\forall x\cdot Px) = P\top$。

87 （双调表）如果一个表在某个索引前是单调上升的，而在其后是反单调的，则称其为双调的。例如表[1;3;4;5;5;6;4;4;3]是双调的。形式化表示表 L 是双调表。

88 形式化命题“有一个自然数，它与任何自然数都不相等”，并证明它不成立。

89 形式化地表示以下各式。

(a) 自然数 *n* 是自然数 *m* 的最大真因子(不为 1 也不为 *m*)。

(b) *g* 是自然数 *a* 和 *b* 的最大公约数。

(c) *m* 是自然数 *a* 和 *b* 的最小公倍数。

(d) *p* 是质数。

(e) *n* 和 *m* 互质。

(f) 至少一个，至多有限个自然数满足谓词 *p*。

(g) 没有最小的整数。

(h) 在任意两个有理数之间总存在另一个有理数。

(i) 表 *L* 是表 *M* 中不包含项 *x* 的最长表段。

(j) 表 *L* 从索引 *i* (包含该项)到索引 *j* (不包含该项)的表段，其项和最小。

(k) *a* 和 *b* (分别)为表 *A* 和 *B* 的项，它们的差的绝对值最小。

(l) *P* 是非空已排序表 *L* 的最长平稳段 (项值相等)的长度。

(m) 表 *L* 中出现的所有项都出现在一个长度为 10 的表段中。

(n) 表 *L* 中所有项都不相同 (没有两项值相等)。

(o) 表 *L* 中至多有一项出现一次以上。

(p) 表 *L* 的最大项出现 *m* 次。

(q) 表 *L* 是表 *M* 的一个排列。

90 （朋友）形式化并证明以下命题：“你所知道的人，是那些被所有你所知道的人所知道的人”。

91 （交换伙伴）有一个有限情侣束，每一对情侣由一位男士和一位女士组成。年龄最大的男士和年龄最大的女士具有相同的年龄。如果任意两对交换伙伴，那么在形成的两对新情侣中，年纪较小的两个伙伴具有相同的年龄。试证明每一对情侣中，两位伙伴具有相同的年龄。

92 证明一个奇自然数的平方是奇数，偶自然数的平方是偶数。

93 利用 ∉ 和 § 表示 ∀ 和 ∃ 。

94 化简下列各式。

(a) $\sum((0,..n)\to m)$

(b) $\prod((0,..n)\to m)$

(c) $\forall\,((0,..n)\to b)$

(d) $\exists\,((0,..n)\to b)$

95 考虑以下二元表达式。

$$nil \to x = x$$

$$(S\,;\,T)\to x = S\to T\to x$$

(a) 它们与第 2，3 章中的理论一致吗？

(b) 根据 2，3 章中的理论，它们是定理吗？

96 (独角兽) 有如下命题。

所有的独角兽都是白的。

所有的独角兽都是黑的。

没有独角兽既是白的又是黑的。

试问这些命题一致吗？如果存在的话，我们可从中得到什么关于独角兽的结论？

97 (罗素的理发师) 罗素说：“在一个小镇上有一个理发师，他为这个小镇上不给自己理发的人理发。”。接着罗素问：“这个理发师为他自己理发吗？”。如果我们说是，那么从上述命题中可以得出他不该为自己理发；如果回答说不是，那么从上述命题中又得出该给自己理发。试形式化此悖论并进行解释。

98 (罗素的悖论) 定义 $rus = \langle f{:}\ (null \to bin) \to \neg\, f\, f \rangle$。

(a) 可以证明 $rus\ rus = \neg\, rus\ rus$ 么？

(b) 这个等式是否不一致？

(c) 可否加入公理 $\neg\, f{:}\ \Delta f$？是否有帮助？

99 (康托的对角线) 证明 $\neg \exists f{:}\ nat \to nat \to nat \cdot\ \forall g{:}\ nat \to nat \cdot\ \exists n{:}\ nat \cdot\ fn = g$。

100 (哥德尔/图灵非完备性) 证明我们无法一致地并完备地定义一个解释器。解释器是一个可以应用于正文的谓词 I；当它应用于代表一个二元表达式的正文时，其结果等价于所代表的表达式。例如，

$$I\ \text{“}\ \forall s : [{*}char] \cdot\ \#s \geq 0\text{”} = \forall s{:}\ [{*}char] \cdot\ \#s \geq 0$$

101 令 f 和 g 为从 nat 到 nat 的函数。

(a) f 为怎样的函数才有定理 $f\, g = g$？

(b) f 为怎样的函数才有定理 $g\, f = g$？

102 $\#[n{*}\mathsf{T}]$ 和 $\not\subset\ \S\ [n{*}\mathsf{T}]$ 有何区别？

103 不使用约束定律，证明

(a) $\forall i \cdot\ Li \leq m = (MAX\ \ L) \leq m$

(b) $\exists i \cdot\ Li \leq m = (MIN\ \ L) \leq m$

104 (鸽子洞) 证明 $(\sum L) > n \times \#L \Rightarrow \exists\, i{:}\ \Delta L \cdot\ L\, i > n$。

105 如果 $f : A \to B$ 并且 $p: B \to bin$，试证明：

(a) $\exists\, b{:}\ fA\cdot\ pb = \exists a{:}\ A\cdot\ pfa$

(b) $\forall b{:}\ fA\cdot\ pb = \forall a{:}\ A\cdot\ pfa$

106 用以下公理定义关系 R 是否有害处？

$\forall x\cdot\ \exists y\cdot\ Rxy$	完全性
$\forall x\cdot\ \neg\ Rxx$	非自反性
$\forall x, y, z\cdot\ Rxy \wedge Ryz \Rightarrow Rxz$	传递性
$\exists u\cdot\ \forall x\cdot\ x = u \vee Rxu$	单一性

107 令 n 是自然数，令 R 是 $0,..n$ 上的关系。换句话说，

$$R : (0,..n) \to (0,..n) \to bin$$

假设从 x 出发经过零步可到达 x 。如果 Rxy 则从 x 经过一步到达 y 。如果 Rxy 和 Ryz 则从 x 经过两步到 z 。以此类推。形式化说明从 x 经过若干步可到达 y 。

108 如果$\forall x, y, z\cdot\ R\ x\ y \wedge R\ y\ z \Rightarrow R\ x\ z$ 成立，那么关系 R 具有传递性。形式化地表示关系 R 是关系 Q 的传递闭包 (R 是被 Q 蕴含的最强的传递关系)。

109 这个问题开发了一个比第 3 章中所介绍的理论更简单，更优美的函数理论。我们将局部变量说明的概念从定义域概念中分离出来，并将后者普遍化，使之成为局部公理说明。变量说明具有形式$\langle v \to b\rangle$, 其中 v 是一个变量， b 是任意一个表达式 (表达式体，不包含定义域)。以下是一个应用定律

$\langle v \to b\rangle x =$ (置换 b 中的 v 为 x)

和延伸定律

$f = \langle v \to fv\rangle$

令 a 为一个二元值，b 为任意一个表达式，于是 a □ b 是一个具有与 b 同样类型的表达式。运算符 □的优先级为 12，且满足右结合。它的公理包括：

T □ $b = b$

a □ b □ $c = a \wedge b$ □ c

表达式 a □ b 是个 “单支 if—表达式”，或 “断言表达式”；它在 b 中引入 a 为一个局部公理。函数是一个体为断言表达式的变量说明，其断言表达式具有形式 $v{:}\ D$。在这种情况下，我们允许如下的一种简写：例如，函数$\langle n \to n{:}\ nat$ □ $n+1\rangle$可以简写成 $\langle n{:}\ nat \to n+1\rangle$。将这个函数应用到 3，我们发现

$\langle n \to n{:}\ nat$ □ $n+1\rangle 3$

$=$ $3{:}\ nat$ □ $3+1$

$=$ T □ 4

$=$ 4

将它应用到–3, 发现

$\langle n \to n{:}\ nat > n+1\rangle\ (-3)$

$=$ $-3{:}\ nat$ □ $-3+1$

$=$ $\perp \square -2$

现在我们被困住了；无法进一步应用公理。在这个例子中，我们同时使用了变量说明和公理说明，给回了我们曾拥有的那种函数，但通常分别使用它们。

(a) 试说明如何在这个新理论中引入值为函数的变量？

(b) 在旧理论中，什么样的表达式在新理论中无对等的表达式？它们可以近似到怎样程度？

(c) 在新理论中，什么样的表达式在旧理论中无对等的表达式？它们可以近似到怎样程度？

----------函数理论结束

10.4 程序理论

110√ 化简以下各式 (x 和 y 为整数变量)。

(a) $x{:=}y{+}1.\ y' > x'$

(b) $x{:=}x{+}1.\ y' > x \wedge x' > x$

(c) $x{:=}y{+}1.\ y' = 2{\times}x$

(d) $x{:=}1.\ x \geq 1 \Rightarrow \exists x{\cdot}\ y' = 2{\times}x$

(e) $x{:=}y.\ x \geq 1 \Rightarrow \exists y \cdot y' = x{\times}y$

(f) $x{:=}1.$ *ok*

(g) $x{:=}1.\ y{:=}2$

(h) $x{:=}1.\ P$ 其中 $P = y{:=}2$

(i) $x{:=}1.\ y{:=}2.\ x{:=}x{+}y$

(j) $x{:=}1.$ **if** $y{>}x$ **then** $x{:=}x{+}1$ **else** $x{:=}y$ **fi**

(k) $x{:=}1.\ x' > x.\ x' = x{+}1$

111 证明规范 S 在前置状态σ下是可满足的，当且仅当 $S.\mathsf{T}$ （注意：T 为 “真”或者 “最高” 二元值）。

112 令 x 为一个整数状态变量，试问以下规范中哪些是可实现的？

(a) $x \geq 0 \Rightarrow x'^2 = x$

(b) $x' \geq 0 \Rightarrow x = 0$

(c) $\neg(x \geq 0 \wedge x' = 0)$

(d) $\neg(x \geq 0 \vee x' = 0)$

113 有四个关于整数变量 x 和 y 的规范。

(i) $x{:=}2.\ y{:=}3$

(ii) $x'{=}2.\ y'{=}3$

(iii) $(x{:=}\ 2) \wedge (y{:=}\ 3)$

(iv) $x'{=}2 \wedge y'{=}3$

(a) 哪些规范使得 x 的最终值为 2，y 的最终值为 3？

(b) 哪些规范是可实现的，哪些是不可实现的？

(c) 哪些规范具有确定性，哪些具有不确定性？

(d) 如果状态变量是 x，y 和 z，其中哪些状态变量具有确定性，哪些状态变量有不确定性？

114 如果一个规范是可传递的，那么它满足：对所有状态 a, b 和 c，如果允许状态从 a 变化到 b，并且允许状态从 b 变化到 c，那么就允许状态从 a 变化到 c。试证明 S 是可传递的，当且仅当 S 可以被 $S.S$ 精化。

115 证明

(a) $x{:=}x = ok$

(b) $x{:=}e.\ x{:=}f\,x \ = \ x{:=}f\,e$

116 证明下列各式成立或不成立

(a) $R.$ **if** b **then** P **else** Q **fi** $=$ **if** b **then** $R.\ P$ **else** $R.\ Q$ **fi**

(b) **if** b **then** $P \Rightarrow Q$ **else** $R \Rightarrow S$ **fi** $=$ **if** b **then** P **else** R **fi** $\Rightarrow$ **if** b **then** Q **else** S **fi**

(c) **if** b **then** $P.\ Q$ **else** $R.\ S$ **fi** $=$ **if** b **then** P **else** R **fi.** **if** b **then** Q **else** S **fi**

117 证明

$$(R \Leftarrow P.\ \textbf{if}\ b\ \textbf{then}\ ok\ \textbf{else}\ R\ \textbf{fi}) \wedge (W \Leftarrow \textbf{if}\ b\ \textbf{then}\ ok\ \textbf{else}\ P.\ W\ \textbf{fi})$$
$$\Leftarrow (R \Leftarrow P.\ W) \wedge (W \Leftarrow \textbf{if}\ b\ \textbf{then}\ ok\ \textbf{else}\ R\ \textbf{fi})$$

118 证明

(a) P 和 Q 分别被 R 精化当且仅当它们的合取式被 R 精化。

(b) $P \Rightarrow Q$ 被 R 精化当且仅当 Q 被 $P \wedge R$ 精化。

119 （卷起）

(a) 我们总可以展开（unroll）循环么？假设 $S \Leftarrow A.\ S.\ Z$，能够得出 $S \Leftarrow A.\ A.\ S.\ Z.\ Z$ 么？

(b) 我们总可以卷起循环吗？假设 $S \Leftarrow A.\ A.\ S.\ Z.\ Z$，能□得出 $S \Leftarrow A.\ S.\ Z$ 吗？

120 试问对什么样的规范 P 和 Q，以下各式为定理：

(a) $\neg(P.\ \neg Q) \Leftarrow P.\ Q$

(b) $P.\ Q \Leftarrow \neg(P.\ \neg Q)$

(c) $P.\ Q = \neg(P.\ \neg Q)$

121 下面的证明有什么错误：

	$(R \Leftarrow R.\ S)$	使用上下文规则
=	$(R \Leftarrow \perp.\ S)$	⊥是.的基
=	$(R \Leftarrow \perp)$	⇐的基定律
=	T	

122 写一个关于以下问题的形式化规范：“改变表变量 L 的值，使之满足每一个表项都被重

复。例如，如果 L 为[6 ; 3 ; 5 ; 5 ;7]，那么它应该变为 [6 ; 6 ; 3 ; 3 ; 5 ; 5 ; 5 ; 5 ; 7 ; 7]。”

123 令 P 和 Q 为规范，C 为前置条件，C' 为相应的后置条件。试证明条件定律：

$$P.\ Q \Leftarrow P \wedge C'.\ C \Rightarrow Q$$

124 令 P 和 Q 为规范，C 为相应的前置条件，C' 为相应的后置条件。以下条件定律中的哪三个可以逆向，即可以交换问题和解？

$C \wedge (P.\ Q) \Leftarrow C \wedge P.\ Q$

$C \Rightarrow (P.\ Q) \Leftarrow C \Rightarrow P.\ Q$

$(P.Q) \wedge C' \Leftarrow P.\ Q \wedge C'$

$(P.Q) \Leftarrow C' \Leftarrow P.Q \Leftarrow C'$

$P.\ C \wedge Q \Leftarrow P \wedge C'.\ Q$

$P.\ Q \Leftarrow P \wedge C'.\ C \Rightarrow Q$

125 令 S 为规范，C 为相应的前置条件，C' 为相应的后置条件。如何使得 S 精化 C' 的精确前置条件与 $(S.\ C)$ 不同？提示：考虑预状态中 S 不可满足，然后确定性，然后非确定性。

126 我们有逐步精化法，部分精化法和情况精化法。在本问题中我们提出替代精化法：

如果 $A \Leftarrow$ **if** b **then** C **else** D **fi** 和 $E \Leftarrow$ **if** b **then** F **else** G **fi** 是定理，

那么 $A \vee E \Leftarrow$ **if** b **then** $C \vee F$ **else** $D \vee G$ **fi** 也是定理。

如果 $A \Leftarrow B.C$ 和 $D \Leftarrow E.F$ 是定理, 那么 $A \vee D \Leftarrow B \vee E.\ C \vee F$ 也是定理。

如果 $A \Leftarrow B$ 和 $C \Leftarrow D$ 是定理，那么 $A \vee C \Leftarrow B \vee D$ 也是定理。

试讨论它们各自的优缺点。

127 令 x 和 y 为实数变量。证明若 $y = x^2$ 在下式前为真，则在其后也为真。

$$x := x + 1. y := y + 2 \times x - 1$$

128√ x 为一个整数变量，

(a) 找出 $x := x+1$ 精化 $x' > 5$ 的精确前置条件 A。

(b) 找出 $x := x+1$ 精化 A 的精确后置条件，其中 A 为(a)的解。

129 证明

(a) 前置条件定律：规范 P 被规范 S 精化的充分前置条件为 C 当且仅当 $C \Rightarrow P$ 被 S 精化。

(b) 后置条件定律：规范 P 被规范 S 精化的充分后置条件为 C’当且仅当 $C' \Rightarrow P$ 被 S 精化。

130 令 L 为整数表，其余变量为整数。试找出以下各式的精确前置条件：

(a) $x'+y' > 8$ 被 $x:=1$ 精化

(b) $x' = 1$ 被 $x:=1$ 精化

(c) $x' = 2$ 被 $x:=1$ 精化

(d) $x' = y$ 被 $y := 1$ 精化

(e) $x' \geq y'$ 被 $x := y + z$ 精化

(f) $y' + z' \geq 0$ 被 $x := y + z$ 精化

(g) $x' \leq 1 \vee x' \geq 5$ 被 $x := x + 1$ 精化

(h) $x' < y' \wedge \exists x \cdot Lx < y'$ 被 $x := 1$ 精化

(i) $\exists y \cdot Ly < x'$ 被 $x := y + 1$ 精化

(j) $L'3 = 4$ 被 $L := i \rightarrow 4 | L$ 精化

(k) $x' = a$ 被 **if** $a > b$ **then** $x := a$ **else** ok **fi** 精化

(l) $x' = y \wedge y' = x$ 被$(z := x.\ x := y.\ y := z)$精化

(m) $ax'^2 + bx' + c = 0$ 被$(x := ax + b.\ x := -x/a)$精化

(n) $f' = n'!$ 被$(n := n + 1.\ f := f \times n)$精化, 其中 n 是自然数，!表示阶乘。

(o) $7 \leq c' < 28 \wedge odd\ c'$ 被$(a := b - 1.\ b := a + 3.\ c := a + b)$精化

(p) $s' = \sum L[0;..i']$ 被$(s := s + Li.\ i := i + 1)$精化

(q) $x' > 5$ 被 $x' : x + (1, 2)$精化

(r) $x' > 0$ 被 $x' : x + (-1, 1)$精化

131 在何种精确前置条件和后置条件下，以下赋值语句可使整数变量 x 的值离 0 更远？

(a) $x := x + 1$

(b) $x := abs(x + 1)$

(c)√ $x := x^2$

132 在何种精确前置条件和后置条件下，以下赋值语句可使整数变量 x 的值离 0 更远并保持在 0 的同一边？

(a) $x := x + 1$

(b) $x := abs(x + 1)$

(c) $x := x^2$

133 令 x 为整数状态变量，并且没有其他的状态变量。

(a) 在何种精确前置条件下，$x := x^2$ 使 x 为偶数？

(b) 如果说你的(a)的答案是 $x := x^2$ 使 x 为偶数的精确前置条件，这说明了什么？

(c) 在何种精确后置条件下，$x := x^2$ 使 x 为偶数？

(d) 如果说你的(c)的答案是 $x := x^2$ 使 x 为偶数的精确后置条件，这说明了什么？

134 (最弱前置规范，最弱后置规范)给定规范 P, Q，找出最弱规范 S (根据 P 和 Q)使得 P 被

(a) $S.\ Q$

(b) $Q.\ S$

精化。

135 令 x 和 n 为自然数变量。试找出一个规范 P，同时满足以下二式：

$$x = x' \times 2^{n'} \Leftarrow n := 0.\ P$$

$P \Leftarrow$ **if** *even x* **then** $x := x/2$. $n := n+1$. P **else** *ok* **fi**

136 令 x 和 y 为二元变量，化简下列各式：

(a) $x := x=y$. $x := x=y$

(b) $x := x \neq y$. $y := x \neq y$. $x := x \neq y$

137 令 a，b 和 c 为整数变量。尽可能简略地化简下列各式，不要用到量词，赋值或相关组合。

(a) $b := a-b$. $b := a-b$

(b) $a := a+b$. $b := a-b$. $a := a-b$

(c) $c := a-b-c$. $b := a-b-c$. $a := a-b-c$. $c := a+b+c$

(d) $a := a+b$. $b := a+b$. $c := a+b$

(e) $a := a+b$. $b' := a+b$. $c := a+b$

(f) $a := a+b+1$. $b := a-b-1$. $a := a-b-1$

(g) $a' := a+b+1$. $b' := a-b-1$

(h) $a := a-b$. $b := a-b$. $a := a+b$

138 (阶乘) n 和 f 为自然数变量，试证明：

$f := n! \Leftarrow$ **if** $n=0$ **then** $f := 1$ **else** $n := n-1$. $f := n!$. $n := n+1$. $f := f \times n$ **fi**

其中 $n! = 1 \times 2 \times 3 \times ... \times n$。

139 n 和 m 为自然数变量，试证明：

$P \Leftarrow n := n+1$. **if** $n=10$ **then** *ok* **else** $m := m-1$. P **fi**

其中 $P = m := m+n-9$. $n := 10$。

140 令 n 和 s 为自然数变量。程序

$R \Leftarrow s := 0$. Q

$Q \Leftarrow$ **if** $n=0$ **then** *ok* **else** $n := n-1$. $s := s+n$. Q **fi**

对前 n 个自然数求和。适当得定义 R 和 Q 并证明两个精化式。

141 令 s 和 n 为数字变量。令 Q 为一个规范，其定义为

$Q = s' = s + n \times (n-1)/2$

(a) 证明精化式

$Q \Leftarrow n := n-1$. $s := s+n$. Q

(b) 根据递归估量添加时间，把 Q 用一个时间规范替代，并重新证明这个精化式。

142 (平方)令 s 和 n 为自然数变量，试找出一个规范 P，同时满足以下二式：

$s' = n^2 \Leftarrow s := n$. P

$P \Leftarrow$ **if** $n=0$ **then** *ok* **else** $n := n-1$. $s := s+n+n$. P **fi**

这个平方程序仅用了加法，减法以及测试是否为零。

143 s 和 n 为自然数变量，证明

$$P \Leftarrow \textbf{if } n=0 \textbf{ then } ok \textbf{ else } n := n-1.\ s := s+2^n-n.\ t := t+1.\ P \textbf{ fi}$$

其中 $P = s' = s+2^n - n\times(n-1)/2 - 1 \wedge n' = 0 \wedge t' = t+n$。

144 令 x 为整数变量，证明以下精化式：

(a)√ $x'=0 \Leftarrow \textbf{if } x=0 \textbf{ then } ok \textbf{ else } x:=x-1.\ x'=0 \textbf{ fi}$

(b) $P \Leftarrow \textbf{if } x=0 \textbf{ then } ok \textbf{ else } x:=x-1.\ t:=t+1.\ P \textbf{ fi}$

其中 $P = x'=0 \wedge \textbf{if } x\geq 0 \textbf{ then } t'=t+x \textbf{ else } t'=\infty \textbf{ fi}$

145 令 x 为整数变量，令 t 为时间变量。找到一个时间规范 P 使得

$$P \Leftarrow \textbf{if } x=0 \textbf{ then } ok \textbf{ else } x:=x+1.\ t:=t+1.\ P \textbf{ fi}$$

并证明这个精化式。

146 令 x 为整数变量，令 t 为时间变量。证明以下精化式：

(a) $x'=1 \Leftarrow \textbf{if } x=1 \textbf{ then } ok \textbf{ else } x:=div\ x\ 2.\ x'=1 \textbf{ fi}$

(b)√ $R \Leftarrow \textbf{if } x=1 \textbf{ then } ok \textbf{ else } x:=div\ x\ 2.\ t:=t+1.\ R \textbf{ fi}$

其中 $R \Leftarrow x'=1 \wedge \textbf{if } x\geq 1 \textbf{ then } t' \leq t + log\ x \textbf{ else } t'=\infty \textbf{ fi}$

147 令 x 为整数变量。当 $P = x<0 \Rightarrow x'=1 \wedge t'=\infty$ 时，以下精化式：

$$P \Leftarrow \textbf{if } x=0 \textbf{ then } ok \textbf{ else } x := x-1.\ t := t+1.\ P \textbf{ fi}$$

为定理吗？这合理吗？请解释。

148 令 a 和 b 为正整数，令 x，u，和 v 为整数变量，令

$$P = u\geq 0 \wedge v\geq 0 \wedge x = u\times a - v\times b \Rightarrow x'=0$$

(a) 证明

$$P \Leftarrow \textbf{if } x>0 \textbf{ then } x:=x-a.\ u:=u-1.\ P$$
$$\textbf{else if } x<0 \textbf{ then } x:=x+b.\ v:=v-1.\ P$$
$$\textbf{else } ok \textbf{ fi fi}$$

(b) 为(a)中程序的执行时间找一个上界。

149 令 x 和 y 为自然数变量。这里有一个精化式。

$$A \Leftarrow \textbf{if } x=0 \vee y=0 \textbf{ then } ok \textbf{ else } x := x-1.\ y := y-1.\ A \textbf{ fi}$$

(a) 添加递归时间。

(b) 找到能给出准确执行时间的规范 A。

(c) 证明执行时间。

150 令 n 为自然数，令 i 和 j 为自然数变量。这里有两个精化式。

$$A \Leftarrow i:=0.\ j:=n.\ B$$

$B \Leftarrow$ **if** $i \geq j$ **then** *ok* **else** $i := i+1.\ j := j-1.\ B$ **fi**

(a) 添加递归时间。

(b) 找到能给出好的时间上界的规范 A 和规范 B，并证明精化式。

151 令 i 为整数变量。根据递归度量加入时间，然后找出满足下式的最强的 P

(a) $P \Leftarrow$ **if** *even* i **then** $i := i/2$ **else** $i := i+1$ **fi**.
if $i=1$ **then** *ok* **else** P **fi**

(b) $P \Leftarrow$ **if** *even* i **then** $i := i/2$ **else** $i := i-3$ **fi**.
if $i=0$ **then** *ok* **else** P **fi**

152 为以下程序找一个作为执行时间上界的有限函数 f，其参数为自然数变量 i 和 j，并证明

$t' \leq t + f\ i\ j \Leftarrow$ **if** $i=0 \wedge j=0$ **then** *ok*
else if $i=0$ **then** $i := j \times j.\ j := j-1.\ t := t+1.\ t' \leq t + f\ i\ j$
else $i := i-1.\ t := t+1.\ t' \leq t + f\ i\ j$ **fi fi**

153 令 P 表示：自然数变量 a 和 b 的终结值分别为 2 和 3 的最大指数，从而满足它们的幂都可以被一个正整数 x 的初始值整除。

(a) 形式化定义 P。

(b) 定义合适的 Q 并证明

$P \Leftarrow a := 0.\ b := 0.\ Q$

$Q \Leftarrow$ **if** x: $2 \times nat$ **then** $x := x/2.\ a := a+1.\ Q$
else if x: $3 \times nat$ **then** $x := x/3.\ b := b+1.\ Q$
else *ok* **fi fi**

(c) 为(b)中的程序执行时间找一个上界。

154 (芝诺) 以下是一个循环。

$R \Leftarrow x := x+1.\ R$

假设递归调用的时间为 2^{-x}，使得每次循环所用时间为上一个循环的一半。证明执行时间是有限的。

155 令 t 为时间变量。 对于 $P = t' = 5$，能够证明以下精化式吗？

$P \Leftarrow t := t+1.\ P$

这是否意味着执行将在时刻 5 终止。有什么错误？

156 在以下精化式中，令 n 和 r 为自然数变量。

$P \Leftarrow$ **if** $n=1$ **then** $r := 0$ **else** $n := n\ div\ 2\ .\ P.\ r := r+1$ **fi**

假设运算符 *div* 和+的执行分别需要 1 个时间单位，而其他操作均不需花费时间(甚至调用也不需时间)。请在此精化式中加入合适的时间增量式，并分别根据以下变量，写一个合适的程序 P，来表示执行时间。

(a) 存储空间变量的初始值。根据你选择的 P 证明此精化式。

(b) 存储空间变量的终结值。根据你选择的 P 证明此精化式。

157 (求总和) 给定一个表变量 L 以及其他一些必要的变量，写一个程序，将表 L 转换成一个累加和的表。形式地有：

(a) $\forall n: 0,..\#L \cdot L'\, n = \sum L\, [0;..n]$

(b) $\forall n: 0,..\#L \cdot L'\, n = \sum L\, [0;..n+1]$

158 (立方) 写一个程序，仅用加、减和测试是否为零三种操作实现求立方运算。

159 (立方测试) 写一个程序，不用求幂的方法，判断一个给定自然数是否为某个数的立方。

160 (模 2) 令 n 为自然数变量。求 n 模 2 的值的问题可以如下解决：

$n' = mod\ n\ 2 \Leftarrow$ **if** $n < 2$ **then** ok **else** $n := n–2\ .\ n' = mod\ n\ 2$ **fi**

利用递归时间度量，找出并证明它的一个时间上界。尽可能使它最小。

161 (模 4) 令 n 为自然数变量。这里有一个精化式。

$n' = mod\ n\ 4 \Leftarrow$ **if** $n < 4$ **then** ok **else** $n := n–4.\ n' = mod\ n\ 4$ **fi**

(a) 证明它。

(b) 根据递归时间度量插入时间增量，并写一个时间规范。

(c) 证明这个时间精化式。

162 形式化地表达规范 R 可被任意次数（包括 0）满足规范 S 的行为的重复所满足。

163 (快速模 2) 令 n 和 p 为自然数变量，求 n 模 2 的值的问题可以如下解决：

$n' = mod\ n\ 2 \Leftarrow$ **if** $n<2$ **then** ok **else** $even\ n' = even\ n.\ n' = mod\ n\ 2$ **fi**

$even\ n' = even\ n \Leftarrow p:=2.\ even\ p \Rightarrow even\ p' \wedge even\ n' = even\ n$

$even\ p \Rightarrow even\ p' \wedge even\ n' = even\ n \Leftarrow$

$n:=n–p.\ p:=p+p.$

if $n<p$ **then** ok **else** $even\ p \Rightarrow even\ p' \wedge even\ n' = even\ n$ **fi**

(a) 证明此精化式。

(b) 利用递归时间度量，找出并证明一个亚线性增长的时间上界。

164 令 n 和 d 为自然数变量。这里有个精化式。

$n' = n + d\times(d\text{-}1)/2 \Leftarrow$

if $d=0$ **then** ok **else** $d:= d\text{-}1.\ n:= n + d.\ n' = n + d\times(d\text{-}1)/2$ **fi**

(a) 证明此精化式。

(b) 根据递归时间度量插入时间增量，并写一个合适的时间规范。

(c) 证明这个时间精化式。

165 令 s 和 n 为自然数变量。这里有个精化式。

$$s' = s + 2^n\text{-}1 \Leftarrow \textbf{if } n{=}0 \textbf{ then } ok \textbf{ else } n := n\text{-}1.\ s := s + 2^n.\ s' = s + 2^n\text{-}1 \textbf{ fi}$$

(a) 证明此精化式。

(b) 根据递归时间度量插入时间增量，并写一个合适的时间规范。

(c) 证明这个时间精化式。

166 已知规范 P 以及预状态σ以 t 作为时间变量，定义“终止的精确前置条件”如下：

$$\exists n{:}\ nat\cdot\ \forall \sigma'\cdot\ t' \le t+n \Leftarrow P$$

令 x 为整数变量，求以下各式终止的精确前置条件，并评价是否合理。

(a) $x \ge 0 \Rightarrow t' \le t+x$

(b) $\exists n{:}\ nat\cdot\ t' \le t+n$

(c) $\exists f : int \to nat\cdot\ t' \le t+f\,x$

167√ (最大项) 写一个程序，求一个表中的最大项。

168 (表比较) 利用项比较而不是表比较，写一个程序，判断一个表是否依据表次序出现在在另一个表之前。

169√ (表求和) 写一个程序，求一个数表的项总和。

170 (交替和) 写一个程序，求数表 L 的交替和 $L0 - L1 + L2 - L3 + \ldots$。

171 令 L 为一个变量，L: [*int]。写一个程序，把 L 中的所有负数项都改为 0，其他项不变。

172 (组合)写一个程序找到将 $a+b$ 个事物划分成 a 个事物和 b 个事物这两部分的可能的方法。包含递归时间。

173 (多项式) 已知 n: nat, c: [n*rat]，x: rat 和变量 y: rat。其中 c 为一个 x 的(“n–1 次”)多项式的系数表。为下式写一个程序：

$$y' = \sum i{:}\ 0,..n.\ ci \times x^i$$

174 (乘法表) 已知 n: nat，和变量 M : [*[*nat]]，写一个程序，将 M 赋值为一个大小为 n 的乘法表，不能使用乘法。例如，当 n=4 时有，

$$\begin{aligned} M' = [\ & [0]; \\ & [0;\ 1]; \\ & [0;\ 2;\ 4]; \\ & [0;\ 3;\ 6;\ 9]\ ] \end{aligned}$$

175 (帕斯卡三角形) 已知 n: nat,和变量 P : [*[*nat]]，写一个程序，将 P 赋值为一个大小为

n 的帕斯卡三角形。例如，当 n=4 时有，

P' = [[1];
[1; 1];
[1; 2; 1];
[1; 3; 3; 1]]

左边和对角线上值都为 1；每个中间项的值为它上一层的顶上数值和左对角线上数值的和。

176√ (2 的指数运算) 已知自然数变量 x 和 y，写一个程序求 $y' = 2^x$，不许使用指数运算。

177√ (快速指数运算) 已知有理数变量 x 和 z，以及自然数变量 y，写一个程序快速求 $z' = x^y$，不许使用指数运算。

178 不使用指数运算，写一个程序找出大于或等于一个给定正整数的最小的 2 的幂次方。

179 (排序测试) 写一个程序，给一个二元变量赋值，以表示一个给定表是否已排序。

180√ (线性查找) 写一个程序，找出给定表中给定项的第一次出现，该程序的执行时间必须与表长成正比。

181√ (二分查找) 写一个程序，在一个已排序的非空表中找一个给定项。执行时间必须与表长呈对数关系。方法是：首先确定给定项如果存在，它会在表的哪二分之一中，接着哪四分之一中，接着哪八分之一中？依次类推。

182 (有相等测试的二分查找) 问题是二分查找（练习 181），但每个迭代步骤测试剩余段的中间项是否是我们所寻找的项。

(a) 写出程序，包括规范和证明。

(b) 根据递归时间度量求出执行时间。

(c) 如果每个测试需要花费时间 1，求执行时间。

(d) 与不进行相等测试的二分查找进行执行时间的比较。

183 (三分查找) 该问题与二分查找一样（练习 181）。这次方法是：首先确定如果给定项出现，它应该在表的哪三分之一中，接着哪九分之一中，接着哪二十七分之一中？依次类推。

184 (逼近查找) 已知一个已排序的非空数表和一个数，写一个程序，得出值与已知数最接近的项的索引。

185√ (二维查找) 写一个程序，在一个二维数组中找一个给定项，执行时间必须与维数的乘积呈线性关系。

186 (已排序二维查找) 写一个程序，在一个给定的行和列均已排序的二维数组中找一个给定项。执行时间必须与维数之和呈线性关系。

187 (已排序二维计数) 写一个程序，在一个给定的且行和列均已排序的二维数组中计算一个给定项的出现次数。执行时间必须与维数之和呈线性关系。

188 (模式查找) 令 *subject* 和 *pattern* 为两段正文。写一个程序作以下事情：如果 *pattern* 在 *subject* 内的某处出现，将它的第一次出现的起始位置赋给自然数变量 *h*。

(a) 使用节 2.3 中给出的任意字符串运算符。

(b) 仅使用字符串索引，但不使用其他字符串运算符。

189 (不动点) 令 *L* 为一个包含不同整数的已排序非空表。写一个程序，找 *L* 的不动点，即返回一个索引 *i* 满足 *Li=i*，或者报告不存在这样的索引。执行时间上限应该是 *log(#L)*。

190 (最早开会时间)写一个程序，找出对三个人都可接受的最早开会时间。每个人都通过一个函数说明他们可能的开会时间，此函数为：对每个时刻 *t*，返回他们在时刻 *t* 或是在 *t* 之后可以开会的最早时间。(不要将这个 *t* 与执行时间变量混淆，在此问题中可以不考虑执行时间。)

191 (全部出现) 已知一个自然数和一张有关自然数表，写一个程序判断是否小于这个自然数的所有自然数都是表中的项。

192 (缺少的数) 已知一个长度为 *n* 的未排序的表，其中的项为 0,..*n*+1 之间的数，但缺少了一个数。试写一个程序找出那个缺少的数。

193 (重复) 写一个程序，判断是否一个已知非空表有任何重复的项。

194 (项计数) 写一个程序，找出一个给定项在一个已知表中出现的次数。

195 (重复计数) 写一个程序，找出多少个项是其前面的项的重复

(a) 在一个已知排序的非空表中。

(b) 在一个已知表中。

196 (无 z 的段落) 已知一段文本，写一个程序，找到最长的不包含字母“z”的一段。

197 (合并) 已知两个已排序的表，写一个程序，把它们合并成一个已排序的表。

198 (文本长度) 已知一段文本(字母串)，它以零个或多个 “正常”字母开头，以零个或多个“修饰”字母结尾。修饰字母不为正常字母。写一段程序，计算文本中正常字母的个数。执行时间应该与文本的长度呈对数关系。

199 (排序对查找) 已知一个至少包含两项的表，表的首项小于或等于它的尾项。写一个程序，找一个邻近项对，满足该对的第一项小于或等于它的第二项。执行时间应该与表长呈对数关系。

200 (凸等对)如果一个数表的每一项(除了第一项和最后一项)都小于或等于其相邻两项的平均值，并且表的长度至少是 2，那么该表称为凸表。给定一个凸表，写一个程序，判断它是否存在一个连续的相等项对。执行时间应该与表长呈对数关系。

201 在整数对之间，定义一个如下的偏序关系《：

$$[a\,;\,b]《[c;\,d] = a<c \wedge b<d$$

已知 $n: nat$ 和 $L:[n*[int;\ int]\ ]$, 写一段程序找出 L 中最小项的索引。也就是，找一个 $j: 0,..\#L$, 满足$\neg \exists\, i \cdot Li《 Lj$。执行时间应该为 n 。

202 (n 排序) 已知一个表 L， $L(0,..\#L) = 0,..\#L$，写一个程序在线性时间和常量空间对 L 进行排序。对 L 所允许的唯一改变是交换两个项。

203√ (n^2 排序) 写一段程序，对一个表排序。执行时间至多为 n^2，其中 n 为表的长度。

204 ($n\times \log n$ 排序) 写一段程序，对一个表排序。执行时间至多为 $n\times \log n$，其中 n 为表的长度。

205 (倒序)写一段程序，将一张表的所有项倒序排列。

206 (下一个有序表) 已知一个自然数的有序表，写一段程序，找出表总和及长度不变的下一个(根据表的顺序)有序表。

207 (下一个组合) 给定一个包含 m 个 $0,..n$ 之间的不同数的有序表，写一段程序，找出按字典排序的下一个包含 m 个 $0,..n$ 之间不同数的有序表。

208 (下一个排列) 给定一个由 $0,..n$ 之间的数组成的某种顺序的表。写一段程序, 找出按字典排序的下一个包含 $0,..n$ 之间数的表。

209 (倒排列) 给定一个表变量 P, 它由 $0,..\#P$ 之间的不同项组成。写一段程序, 使得 $P\ P'=[0;..\#P]$。

210 (对角线) 有一些点被安排在一个圆周的附近。按顺时针方向从每一个点到它的下一个点的距离由一个表给出。写一段程序找出相距最长的两个点。

211 (幂等排列)给定一个由 $0,..\#L$ 之间的项(不一定全都不同)组成的表变量 L。写一段程序对该表进行排列, 最后满足 $L'L' = L'$。

212 (局部极小量) 给定一个由至少 3 个数组成的表 L, 满足 $L0 \geq L1$ 和 $L(\#L-2) \leq L(\#L-1)$。一个局部极小量是一个内部索引 i: 1,..$\#L-1$ 满足

$$L(i-1) \geq Li \leq L(i+1)$$

写一段程序, 找出 L 的一个局部极小量。

213 (自然除法)自然数 n 除以正整数 p 所得的商为满足下式的自然数 q：

$$q \leq n/p < q+1$$

写一段程序, 在 *log n* 时间内找出 n 除以 p 的自然商, 不使用函数 *div, mod, floor* 或 *ceil*。

214 (余数) 写一段程序找出整数相除（练习 213）后的余数, 可以使用比较，加法和减法(不可使用乘法, 除法或模运算)。

215 (以自然数 2 为底的对数)一个正整数 p 的以 2 为底的对数为满足下式的自然数 b：

$$2^b \leq p < 2^{b+1}$$

写一段程序, 在 *log p* 时间内找出一个给定正整数 p 的以 2 为底的对数。

216 (自然平方根)一个自然数 n 的自然平方根为满足下式的自然数 s：

$$s^2 \leq n < (s+1)^2$$

(a) 写一段程序, 在 *log n* 时间内找出一个给定自然数 n 的自然平方根。

(b) 写一段程序, 在 *log n* 时间内找出一个给定自然数 n 的自然平方根, 仅允许使用加法、减法、翻倍、减半和比较(不允许使用乘法或除法)。

217 (因式计数) 写一段程序, 找出一个给定自然数的因式(不一定要求是质因子)个数。

218 (费马最后程序) 给定自然数 c, 写一段程序, 找出满足 $a^2+b^2 = c^2$ 的无序自然数对 a 和 b 的个数, 该程序的时间与 c 的大小呈正比。(一个无序对实际上就是一个大小为 1 或 2 的束。如果已经对数对 a 和 b 计数, 我们就不再希望对数对 b 和 a 计数。) 程序可以使用加法、减法、乘法、除法和比较， 但不准使用指数或平方根运算。

219 (展开)写一段程序展开一张表，结果返回一张与原旧表相近的新表，但除去了旧表中的内部结构。例如，

L = [[3; 5]; 2; [5;[7]; [*nil*]]]

L' = [3; 5; 2; 5; 7]

程序可以使用一个测试 *Li: int*， 以检查一个表项是否为一个整数或为一张表。

220 (最小和段) 给定一个整数表, 其中可能包含负数，写一段程序

(a) √ 寻找任意段(连续项组成的子表)的最小和。

(b) 寻找和为最小的段（连续项组成的子表）。

221 (最大积段) 给定一个整数表, 其中可能包含负数，写一段程序

(b) √ 寻找任意段(连续项组成的子表)的最大积。

(c) 寻找积为最大的段（连续项组成的子表）。

222 (段和计数)

(a) 写一个程序, 找出在给定自然数表中，段之和等于一给定自然数的段的个数。

(b) 写一个程序, 找出在给定正自然数表中，段之和等于一给定自然数的段的个数。

223 (最长排序子表) 写一个程序，从一个已知表中找到一个最长排序子表的长度，其中

(a) 子表必须是连续的项（一段）。

(b) 子表中的项的顺序和它们在已知表中的顺序一致，但不必是连续的。

224 (近乎排序段)一个近乎排序表中，最多一对相邻的元素顺序不对。写一个程序，找到已知表中最长的近乎排序段的长度。

225 (最长平稳段) 给定一个非空已排序数表。一个平稳段为一个具有相等项的表段(由连续项构成的子表)。写一段程序，找出

(a) 最长平稳段的长度。

(b) 最长平稳段的个数。

226 (最长平滑段) 在一个整数表中，一个平滑段为一个连续项的子表，其中每两个相邻项的差值不超过 1。写一段程序找一个最长平滑段。

227 (最长平衡段) 给定一张二元值表，写一段程序, 找出具有相等个数的项 T 和项⊥组成的最长段(连续项组成的子表)。

228 (最长回文) 回文是一个其正向表和它的反向表相等的表。写一段程序找出一个给定表中的最长回文段。

229 (最大子序列) 给定一张表，根据表序列，找出它的最大子表。(一个子表中所包含的项一定属于原父表，且这些项出现的次序与它们在原父表中出现的次序相同，但不必一定为连续的项。)

230 给定一张由 0, 1, 2 组成的表，写一段程序，

(a) 找出以任意次序包含所有这三个数的最短段(连续项)的长度。

(b) 找出按序出现 0, 1, 2 这三个数的子表(项不必连续)的个数。

231 令 L 和 M 为已排序的数表，写一段程序，找出满足 $Li \le Mj$ 的索引对 i:0,..#L 和 j:0,..#M 的个数。

232 (头和尾) 令 L 为一个正整数表。写一段程序，找出满足以下条件的索引对 i 和 j 的个数：$\sum L[0;..i] = \sum L[j;..\#L]$

233 (支点) 给定一个非空正数表，写一段程序找出它的平衡点，即支点。支点两边的平衡量分别等于位于支点同侧的各表项的值（比重）与表项到支点的距离的乘积和，表项 i 被放置在点 i+1/2 处，而支点类似得可能是非整数。

234 (最小差值) 给定两个非空已排序的数表。写一段程序，分别从这两个表中找出一项，并满足它们的差值的绝对值最小。

235 (最早放弃者) 在一个非空表中寻找第一个后面不再重复的项。在表[13;14;15;14;15;13]中最早放弃者为 14，因为其他项 13 和 15 都在 14 的最后出现之后依然出现。

236 (区间合并) 在一条实数线上有一个区间集合，这些区间的左端点由表 L 表示，而右端点由相应的表 R 表示。表 L 已排序，且这些区间有些可能重叠，有些可能有间隔。写一段程序找出被这些区间覆盖的实数线段的总长度。

237 (字位和) 写一段程序，找出一个用二进制表示的给定自然数所有为 1 字位的个数。

238 (数字和) 写一段程序，找出一个用十进制表示的给定自然数各位数字的和。

239 (奇偶校验) 写一段程序，检查一个用二进制表示的自然数为 1 字位的个数是偶数还是奇数。

240 给定两个自然数 s 和 q，写一段程序，在 s^2时间内，找出四个自然数 a，b，c 和 d（如果存在的话） 满足：它们的和为 s，乘积为 p。

241 给定三个自然数 n，s，和 p，写一段程序，找出一个长度为 n 的自然数表 (如果存在的话)，满足它们的和为 s，乘积为 p。

242 (传递闭包) 一个关系 R: (0,..n)→ (0,..n)→ bin 可以表示成大小为 n 的平方二元数组。给定一个平方二元数组方式表示的关系，写一段程序找出

(a) 它的传递闭包(由给定关系所蕴含的最强的传递关系)。

(b) 它的自反传递闭包(由给定关系所蕴含的最强的自反传递关系)。

243 (可达性) 给定一个有限地点束，和一个关于地点的后继函数 S, 对每个地点，S 告诉我们从那个地点可以直接到达的其他地点。另外还给定一个特殊地点 h (代表家)。写一段程序，找出所有可从 h 到达(自反地、直接地或间接地)的地点。

244 (最短路径) 给定一个平方广义有理数组，项 i j 代表了从地点 i 到地点 j 的直接距离。如果不可能直接从地点 i 到地点 j, 那么项 i j 的值为 ∞。写一段程序找出一个平方广义有

理数组，满足项 *i j* 代表了从地点 *i* 到地点 *j* 的可能间接的最短距离。

245 (反序计数) 给定一张表，写一段程序找出有多少对项(不必连续)次序混乱，即大项位于小项之前。

246 (麦卡锡的 91 个问题) 令 *i* 为一个整数变量，令

M = **if** *i*>100 **then** *i*:=*i*–10 **else** *i*:=91 **fi**

(a) 证明 *M* ⇐ **if** *i*>100 **then** *i*:=*i*–10 **else** *i*:=*i*+11. *M*. *M* **fi**。

(b) 找出(a)中 *M* 精化式的执行时间。

247 (阿克曼) *ack* 是一个如下定义的关于两个自然数变量的函数，

ack 0 0 = 2
ack 1 0 = 0
ack (*m*+2) 0 = 1
ack 0 (*n*+1) = *ack* 0 *n* + 1
ack (*m*+1) (*n*+1) = *ack m* (*ack* (*m*+1) *n*)

(a) 假设函数和函数应用为没有实现的表达式； 在这种情形下，*n*:= *ack m n* 不是一个程序。试精化 *n*:= *ack m n* 来获得一个程序。

(b) 找出一个时间界。提示：可以在时间界中使用函数 *ack*。

(c) 找到空间界。

248 (交替阿克曼)对下面的函数 *f*，精化 *n*:= *f m n*，找出时间界（可能包含 *f*），并找出空间界。

(a) *f* 0 *n* = *n* +2
f 1 0 = 0
f (*m*+2) 0 = 1
f (*m*+1) (*n*+1) = *f m* (*f*(*m*+1) *n*)

(b) *f* 0 *n* = *n* ×2
f (*m*+1) 0 = 1
f (*m*+1) (*n*+1) = *f m* (*f*(*m*+1) *n*)

(c) *f* 0 *n* = *n* +1
f 1 0 = 2
f 2 0 = 0
f (*m*+3) 0 = 1
f (*m*+1) (*n*+1) = *f m* (*f*(*m*+1) *n*)

249√ (过山车)令 *n* 为一个自然数变量。很容易证明：

n′ =1 ⇐ **if** *n*=1 **then** *ok*
else if *even n* **then** *n*:=*n*/2. *n*′ =1
else *n*:=3*n*+1. *n*′ =1 **fi fi**

问题是如何找出它的执行时间。警告： 这一问题从未被解决过。

250√ (斐波那契数) 斐波那契数 *fib n* 定义如下：

$$fib\ 0 = 0$$

$$fib\ 1 = 1$$

$$fib\ (n+2) = fib\ n + fib\ (n+1)$$

写一段程序，在 *log n* 时间内找出 *fib n*。提示：参见练习 350。

251 (斐波卢契数) 令 *a* 和 *b* 为整数。那么 *a* 和 *b* 的斐波卢契数为：

$$flc\ 0 = 0$$

$$flc\ 1 = 1$$

$$flc\ (n+2) = a \times flc\ n + b \times flc\ (n+1)$$

(斐波那契数是 1 和 1 的斐波卢契数。)给定自然数 *k*, 不使用任何表变量，写一段程序计算：

$$\sum n: 0,..k \cdot flc\ n \times flc\ (k-n)$$

252 令 *n* 为一个自然数变量。根据递归时间度量，在下式中加入时间，并找出它的执行时间的一个有限上界。

$$P \Leftarrow \textbf{if } n \geq 2 \textbf{ then } n := n-2.\ P.\ n := n+1.\ P.\ n := n+1 \textbf{ else } ok \textbf{ fi}$$

253 (算术) 将一个自然数表示成一个自然数表，如果以 *b*>1 作为自然数的基，那么表中每一个数都处于 0,..*b* 之间，且倒序排列。例如，如果 *b*=10，那么[9；2；7]代表了 729。根据以下各式写一段程序。

(a) 找出一个代表给定自然数在一个给定基下的这种表。

(*b*) 给定一个基和两个代表自然数的表，找出代表它们的和的表。

(c) 给定一个基和两个代表自然数的表，找出代表它们差值的表。假设第一个表所代表的自然数大于或等于第二张表所代表的自然数。如果不是这样的话，结果怎样？

(d) 给定一个基和两个代表自然数的表，找出代表它们乘积的表。

(e) 给定一个基和两个代表自然数的表，找出代表它们的商和余数的表。

254 (机器乘法) 给定两个自然数，写一段程序，计算它们的乘积，只许使用加法、减法、加倍、减半、测试是否为偶数以及测试是否为零。

255 (机器除法) 给定两个自然数，写一段程序，计算它们的商, 只许使用加法、减法、加倍、减半、测试是否为偶数以及测试是否为零。

256 (机器平方) 给定一个自然数，写一段程序，计算它的平方，只许使用加法、减法、加倍、减半、测试是否为偶数以及测试是否为零。

257 给定一个多项式的根表，写一段程序找出该多项式的系数表。

258 (编辑距离) 给定两张表，写一段程序，找出在将一张表转变成另一张表的过程中，项插入、删除和替换的最少次数。

259 (终极周期序列) 已知 $f : int \rightarrow int$ 为这样一个函数，使得如下序列：

$$x_0 = 0$$

$$x_{n+1} = f(x_n)$$

由 f 产生从 0 开始，为一个终极周期：

$$\exists p: nat+1 \cdot \exists n: nat \cdot x_n = x_{n+p}$$

满足 $\exists n:nat \cdot x_n = x_{n+p}$ 的最小正数 p 称为周期。写一段程序找出这个周期。程序所使用的存储单元应该小于一个常数，并且不依赖于 f。

260 (最大真方阵) 写一段程序，在一个二元真值表中找一个全部项为 T 的最大子方阵。

261 (分割) 如果一张正整数表称为是一个自然数 n 的分割，那么该表所有项的和等于 n。写一段程序找出

(a) 由一个给定自然数 n 的所有分割组成的表。例如，如果 n=3, 那么一个可接受的回答为[[3]; [1;2]; [2;1]; [1;1;1]]。

(b) 由一个给定自然数 n 的所有已排序的分割组成的表。例如，如果 n=3, 那么一个可接受的回答为[[3] ; [1;2] ; [1;1;1]]。

(c) 由一个给定自然数 n 的所有分割组成的已排序的表。例如，如果 n=3, 那么一个可接受的回答为[[1;1;1]; [1;2]; [2;1]; [3]]。

(d) 由一个给定自然数 n 的所有已排序的分割组成的已排序表。例如，如果 n=3, 那么一个可接受的回答为[[1;1;1] ; [1;2] ; [3]]。

262 (P–表) 给定一个非空自然数表 S，定义 P–表为一个非空自然数表 P，并满足 P 的每一项都是 S 的一个索引，以及

$$\forall i: 1,..\#P \cdot P(i-1) < P\ i \leq S\ (P\ (i-1))$$

写一段程序，找出一个给定表 S 的最长 P–表的长度。

263 (J–表) 对于自然数 n，一个阶数为 n 的 J–表是一个由 $2n$ 个自然数组成的表，其中每个数 m: 0,..n 出现两次，且在数 m 的两次出现之间有 m 个项。

(a) 写一段程序，为一个给定自然数 n 创建(如果有的话)一个阶数为 n 的 J–表。

(b) 对什么样的 n 存在这样的 J–表？

264 (缩减 J–表) 对正整数 n，一个阶数为 n 的缩减 J–表是一个由 $2n-1$ 个自然数组成的表，其中 0 出现一次，其他的 m:1,..n 出现两次，且在 m 的两次出现之间有 m 个项。

(a) 写一段程序，为一个给定自然数 n 创建(如果有的话)一个阶数为 n 的缩减 J–表。

(b) 对什么样的 n 存在这样的缩减 J–表？

265 (最大公约数)写一段程序，根据以下描述，找出最大公约数。

(a) 两个正整数。

(b) 两个不都为 0 的整数（不一定是正数）。

(c) 三个正整数。一个方法是，找到其中两个数的最大公约数，然后找到所得数和余下那个数的最大公约数，但存在更好的方法。

266 (最小公倍数) 给定两个正整数，写一段程序找出它们的最小公倍数。

267 (公共项) 令 A 为一个由不同整数组成的已排序的表，B 为另一个这样的表。写一段程序，找出同时出现在两个表中的整数的个数。

268 (独特项) 令 A 为一个由不同整数组成的已排序的表，B 为另一个这样的表。写一段程序，找出一个已排序的整数表，满足其中每一个项仅出现在 A 和 B 之一的表中。

269 (最小公共项) 给定两个至少有一个公共项的已排序的表，写一段程序，找出同时出现在两个表中的最小公共项。

270 (最长公共前缀) 一个正整数可以写成以一个零开头的十进制数序列。给定两个自然数，写一段程序，找出作为它们的最长公共前缀的数。例如，给定 025621 和 02547，结果应为 025。提示：该问题是关于数的，而不是关于串或表的。

271 给定三个至少有一个公共项的已排序的表，写一段程序，找出同时出现在三个表中的最小公共项。

272 (博物馆) 给定一个自然数 n，有理数 s 和 f (开始和结束)，和表 A, D: [n*rat](到达和离开)，满足：

$$\forall i: s \le Ai \le Di \le f$$

它们代表了一个博物馆开放时刻为 s，参观人数为 n，其中人物 i 的到达时间为 Ai，离开时间为 Di，博物馆的关闭时间为 f。写一段程序，找出总共的参观时间，即至少有一个人在博物馆内，并找出它在开放时间内的平均参观人数，程序的执行时间要与 n 呈线性关系，且考虑以下情形：

(a) 表 A 为已排序。

(b) 表 D 为已排序

273 (移动测试) 给定两个无限长的表 A 和 B。表项可被对比以求顺序。两个表都有周期 n: nat+1。

$$\forall k: nat \cdot Ak = A(k+n) \wedge Bk = B(k+n)$$

写一个程序，判断是否表 A 和表 B 是一样的，不考虑索引的移动。

274 (旋转测试) 给定两个表，写一段程序，判断其中一个表是否为另一个表的一个旋转。

可以使用项比较，但不可使用表比较。执行时间应与表的长度呈线性关系。

275 (最小旋转) 给定一个文本变量 t，写一段程序，对它重新赋值，使它成为按字符(字典)排序的最小旋转。可以使用字符比较，但不可使用文本比较。

276 给定一个已赋值为一个非空表的表变量 L，所有对 L 的改变都必须通过过程 *swap*，它的定义如下：

$$swap = \langle i, j: 0,..\#L \rightarrow L:=i \rightarrow Lj \mid j \rightarrow Li \mid L \rangle$$

(a) 写一段程序，将原表向右旋转一个位置 (即将原表的最后一项变为新表的第一项) 形成一个新表，并将新表重新赋值给 L。

(b) (旋转) 给定一个整数 r，写一段程序，将原表向右旋转 r 个位置 (如果 $r<0$，向左旋转 $-r$ 个位置) 形成一个新表，并将新表重新赋值给 L。递归执行时间至多为 $\#L$。

(c) (段交换) 给定一个索引 p，交换表 L 在位置 p 处的前后两段，p 为后段的开头。

277 (荷兰国旗) 给定一个变量

$$flag: [*(red, white, blue)]$$

给它排序，使得所有 *red* 值都在前面，所有 *white* 值都在中间，所有 *blue* 值都在最后。唯一可用的改变 *flag* 的方法是

$$swap = \langle i, j: 0,..\#flag \rightarrow flag:= i \rightarrow flag\ j \mid j \rightarrow flag\ i \mid flag \rangle$$

278 (挤压) 令L为一个已赋值为一个非空表的表变量。对它重新赋值，满足两个或两个以上的相同项都被挤压成一个单独的项。

279 令 n 和 p 为自然数变量。写一段程序求解

$$n \geq 2 \Rightarrow p': 2^{2^{nat}} \ \wedge \ n \leq p' < n^2$$

包含一个执行时间的有限上界，不论多小都无关系。

280 (直方图的最大方阵列) 以自然数表 H 的形式给定一个直方图。写一段程序，找出 H 的最长段，满足它的高度(每一项)至少与段长一样大。

281 (长文本) 一个特定的计算机对字符个数小于或等于某个特定的常数 n 的文本有专门的硬件表示。超过 n 的长文本就必须以软件形式表示成一个由短文本组成的表串。(长文本是短文本的连接。) 如果除了最后一项的所有其他项都具有长度 n，那么这个长文本称为是“压缩的”。写一段程序，压缩一个短文本表串，但不改变它所表示的长文本。

282 (克努斯,莫里斯,普拉特)

(a) 给定表 P，找一个表 L，满足对表 P 的每一个索引 n，Ln 代表了同时为 $P[0;..n+1]$的真前缀和真后缀的最长表的长度。这里有一个程序可以找出 L。

$$A \Leftarrow i:=0.\ L:=[\#P*0].\ j:=1.\ B$$

$$B \Leftarrow \textbf{if}\ j \geq \#P\ \textbf{then}\ ok\ \textbf{else}\ C.\ L:=j \rightarrow i \mid L.\ j:=j+1.\ B\ \textbf{fi}$$

$$C \Leftarrow \textbf{if}\ Pi=Pj\ \textbf{then}\ i:=i+1$$

else if *i*=0 **then** *ok*

else *i*:=*L* (*i*–1). *C* **fi fi**

找出规范 *A, B* 和 *C*，使得 *A* 为问题，且三个精化为定理。

(b) 给定表 *S* (主题)，表 *P* (模式)，和表 *L* (同(a)中一样)，判断 *P* 是否为 *S* 的一个段，如果是，它出现在哪里？以下是一个程序。

D ⇐ *m*:= 0. *n*:= 0. *E*

E ⇐ **if** *m*=#*P* **then** *h*:= *n*–#*P* **else** *F* **fi**

F ⇐ **if** *n*=#*S* **then** *h*:= ∞

else if *Pm*=*Sn* **then** *m*:=*m*+1. *n*:=*n*+1. *E*

else *G* **fifi**

G ⇐ **if** *m*=0 **then** *n*:=*n*+1. *F* **else** *m*:=*L* (*m*–1). *G* **fi**

找出规范 *D, E, F* 和 *G*，使得 *D* 为问题，且四个精化为定理。

283 令 *x* 为一个自然数变量。在精化式中

P ⇐ **if** *x*=1 **then** *ok* **else** *x*:= *div x 2*. *P*. *x*:= *x*×2 **fi**

每个调用把一个返回地址推到堆栈，并且每个返回从堆栈上拿下来一个地址。添加一个空间变量 *s* 和一个最大空间变量 *m*，在程序中对它们进行合适的赋值。找到并证明所用最大空间的上限。

284 (阶乘空间) 可以如下计算 *x*:=*n*!（阶乘）。

x:=*n*! ⇐ **if** *x*=0 **then** *x*:= 1 **else** *n*:= *n*-1. *x*:= *n*!. *n*:= *n*+1. *x*:= *x*×*n* **fi**

x:=*n*!的每一次调用把一个返回地址推到堆栈，并且每个返回从堆栈上拿下来一个地址。添加一个空间变量 *s* 和一个最大空间变量 *m*，在程序中对它们进行合适的赋值。找到并证明所用最大空间的上限。

285 令 *k* 为一个自然常数，令 *x* 和 *n* 为自然数变量。假设每个递归调用（放返回地址）之前，一个单位的空间已经被分配，并在调用之后被释放。找到并证明精化式的最大空间界限

P ⇐ **if** *n*=0 **then** *x*:= 0 **else** *n*:= *n*-1. *P*. *x*:= *x*+*k* **fi**

286 (汉诺塔) 有 3 个塔和 n 个盘子。盘子是有大小的；盘子 0 最小，盘子 $n-1$ 最大。初始时塔 A 上有所有的 n 个盘子，盘子按照大小顺序放在塔上，最大的盘子在最下面，最小的盘子在最上面。任务是将所有的盘子从塔 A 移动到塔 B，但每次只能移动一个盘子，而且大的盘子永远不能放在小的盘子上面。在这个过程中，允许使用塔 C 作为过渡。

(a)√ 使用命令 *MoveDisk from to* 让机器手臂将塔 *from* 上最顶端的盘子移动到塔 *to* 上，写一个程序将塔 A 上的所有盘子移动到塔 B 上。

(b)√ *MoveDisk* 的计算时间为 1，其他动作不消耗时间，求执行时间。

(c) 假设塔的放置呈等边三角形，因此手臂每次移动的距离是常数（到达想去的位置的三角形的一条边加上移动盘子的一条边），而不依赖于被移动的盘子。假设移动盘子的

时间随被移动的盘子的重量而变化，而重量随它的面积变化，而面积随它的半径的平方而变化，而半径随盘子的编号而变化。求执行时间。

(d)√ 假设一个递归调用所需空间计为 1（放返回地址），其他不需空间，求程序所需的最大存储空间。

(e)√ 假设一个递归调用所需空间计为 1（放返回地址），其他不需空间，求程序所需的平均存储空间。

(f) 假设一个递归调用所需空间计为 1（放返回地址），其他不需空间，求程序所需的平均存储空间的简单上界。

287 (硬币重量) 给定一些硬币，除了其中一个硬币可能比标准重量轻或者重，其他所有硬币都是同一个标准重量。给定一个天平标尺，以及需要的任意多的标准硬币。写一个程序，判断是否存在一个不标准的硬币，如果存在是哪一个，它是比标准更重还是更轻，在最少次数称重的前提下。

----------程序理论结束

10.5 程序设计语言

288 (不确定性赋值) 将赋值记号 *x:=e* 普遍化，使得表达式 *e* 可以为一个束，该赋值语句表示 *x* 被赋值为这个束中的任意一个元素。例如，*x:=nat* 表示将 *x* 赋值为任意一个自然数。试用标准的二元记号表示这种形式的赋值语句，说明它对置换定律有何影响。

289 假设有变量说明定义如下：

var *x*: *T*· *P* = ∃*x*: *undefined*· ∃*x'*: *T*· *P*

试问这种形式的变量说明有何特点？考虑例子

var *x:int*· *ok*

290 假设带初始化的变量说明定义如下：

var *x*: *T* := *e*· *P* = **var** *x*: *T*· *x*:=*e*. *P*

这种定义与第 5.5.0 节中给出的定义在哪些方面不同？

291 下面有两个带初始化的变量声明的不同定义：

var $x : T := e \cdot P = \exists x, x' : T \cdot x = e \wedge P$

var *x*: *T* := *e*· *P* = ∃*x'*: *T*·（将 P 中的 x 置换为 e ）

用一个例子说明它们有何不同。

292 如下定义局部变量有何错误：

var *x*: *T*· *P* = ∀*x*: *T*· ∃*x'*: *T*· *P*

293 规范

var *x*: *nat*· *x*:= –1

说明了一个局部变量，并对它赋了一个超出其定义域范围的值。这个程序可实现吗? (要求证明)

294 (框架问题) 假设有一个非局部变量 x，定义 $P = x' = 0$。是否可证明：

$$P \Leftarrow \textbf{var}\ y{:}\ nat \cdot y := 0.\ P.\ x := y$$

问题是 y 不是定义 P 的状态空间的一部分，所以 P 是否应该保持 y 不变？提示：考虑相关组合的定义。它的使用合适吗？

295 令 x, y, z 为状态变量，不用 **frame** 重写 **frame** $x \cdot \mathsf{T}$。用语言说明 x 的最后值。

296 令 x, y, n 为自然数变量。令 $f{:}\ nat \to nat$ 为一个函数。简化

$$\textbf{frame}\ x \cdot \textbf{var}\ y, m{:}\ nat \cdot m := n.\ x' = fm \wedge y' = f(m+1)$$

297 在一个可以对数组元素进行赋值的语言中，程序

$$x := i.\ i := A\ i.\ A\ i := x$$

表示交换 i 和 $A\ i$ 的值。假设所有的变量和数组元素都为类型 nat，并且 i 的值等于 A 的一个索引。

(a) 使用变量 x，i 和 A，详细表述 i 与 Ai 应该交换， A 的其他部分不变，但 x 可能改变。

(b) 找出能精化(a)中规范的程序的精确前置条件。

(c) 找出能精化(a)中规范的程序的精确后置条件。

298 在一个可以对数组元素进行赋值的语言中，$(A(A\ i) := 0.\ A\ i := 1.\ \ i := 2)$精化 $A'\ i' = 1$ 的精确前置条件是什么？

299 令 n 为自然数常量，令 f 和 i 为自然数状态变量。定义

$$n! = \prod i{:}\ 0,..n \cdot i+1 = 1 \times 2 \times 3 \times ... \times n$$

证明

$$f' = n! \Leftarrow f := 1.\ i := 0.\ \textbf{while}\ i < n\ \textbf{do}\ i := i+1.\ f := f \times i\ \textbf{od}$$

300√ (无界的界) 为下面包含自然数变量 x 和 y 的程序寻找一个时间界。

$$\begin{array}{l}\textbf{while}\ \ \neg x = y = 0\\ \textbf{do}\ \ \textbf{if}\ \ y > 0\ \ \textbf{then}\ \ y := y - 1\\ \quad \textbf{else}\ x := x\text{-}1.\ \textbf{var}\ n{:}\ nat \cdot y := n\ \textbf{fi}\ \textbf{od}\end{array}$$

301 令 $W \Leftarrow \textbf{while}\ b\ \textbf{do}\ P\ \ \textbf{od}$ 为 $W \Leftarrow \textbf{if}\ b\ \textbf{then}\ \ P.W\ \textbf{else}\ ok\ \ \textbf{fi}$ 的另一种写法，令 $R \Leftarrow \textbf{do}\ P\ \textbf{until}\ b\ \textbf{od}$ 为 $R \Leftarrow P.\ \textbf{if}\ b\ \textbf{then}\ ok\ \textbf{else}\ R\ \ \textbf{fi}$ 的另一种写法。现在证明

$$\begin{array}{ll}& (R \Leftarrow \textbf{do}\ P\ \textbf{until}\ b\ \textbf{od}) \wedge (W \Leftarrow \textbf{while}\ \neg b\ \textbf{do}\ P\ \textbf{od})\\ \Leftarrow & \quad (R \Leftarrow P.W) \wedge (W \Leftarrow \textbf{if}\ \ b\ \textbf{then}\ \ ok\ \textbf{else}\ R\ \textbf{fi})\end{array}$$

302 令 $P{:}\ nat \to bin$。

(a) 定义量词 *FIRST* 使得 *FIRST m*: *nat*· *Pm* 是 *Pm* 成立的最小自然数 *m*，并且如果这样的数不存在则值为∞。

(b) 证明 *n*:= *FIRST m*: *nat*· *Pm* ⇐ *n*:= 0. **while** ¬*Pn* **do** *n*:= *n*+1 **od** 。

303 给定自然数表变量*L*，索引变量*i*，和时间变量*t*，把每个表项都增加1直到产生项100。时间上界是*#L* 。程序是：

```
i:= 0.
do   exit when i=#L.
     Li:= L i + 1.
     exit when L i = 100.
     i:= i+1 od
```

写一个形式化的规范，并证明它被这段程序精化。

304 这里有一个循环的嵌套。所有 **exit** 都显示出来了。为了证明这个循环嵌套精化了规范 *S*，什么精化式需要被证明？

(a)
```
do  A.
    do  B.
        exit 2 when u.
        C.
        exit 1 when v.
        D od.
    E.
    exit 1 when w.
    F.
    do  G.
        do  H.
            exit 1 when x.
            I.
            exit 2 when y.
            J od.
        K.
        exit 1 when z.
        L od.
    M od
```

(b)
```
do  A.
    do  B.
        do  C.
            exit 1 when u.
            exit 2 when v.
            exit 3 when w.
            D od.
        E od.
    F od
```

305√ 使用 **for**-循环写一个程序在表的每个项上加 1。

306 (平方) 令 *n* 为一自然数，令 *s* 为一自然数变量。用一个 **for**-循环为 *s*≒n^2 写一段程序，

不许用乘法或求幂。

307 令 L 为一变量，L: [*int]。这里有一段程序，把 L 中所有为负的项都变为 0，其他项不变。

for n:= 0;..#L **do if** Ln<0 **then** L:= n→0 | L **else** ok **fi od**

写出所有所需的的规范和精化式，以证明这个程序是按照其原本意图执行的。无需证明精化式。

308 以下是定义 **for**–循环的一种可能的方法。令 j, n, k 和 m 为整数表达式，并令 i 为一个新名。

for i:=nil **do** P **od** = ok

for i:=j **do** P **od** = (在 P 中置换 i 为 j)

for i:=n;..k ; k;..m **do** P **od** = **for** i:=n;..k **do** P **od**. **for** i:=k;..m **do** P **od**

(a) 从这个定义中，关于 **for** i:=0;..n **do** n:=n+1 **od** 可以证明些什么？其中 n 是一个整数变量。

(b) 在你所知道的程序设计语言中有哪几种 **for**–循环？

309 (多数投票) 这个问题是在一个给定表中找出一个多数项(即它出现在一半以上的位置中)如果这个项存在。令 L 为给定表，m 为一个变量，其最终值为所求的那个多数项，证明以下程序可求解该问题。

(a)
```
var e: nat:= 0·
for i:=0;..#L
do  if m = L i then e:= e+1
    else if i = 2×e then m:=Li. e:=e+1
         else ok fi fi od
```

(b)
```
var s: nat:= 0·
for i:=0;..#L
do  if m = L i then ok
    else if i = 2×s then m:=L i
         else s:=s+1 fi fi od
```

310 令 a 和 b 为前状态（前置条件）的二元表达式，令 A，B，C，P，Q，R，S，T 和 U 为可实现的规范，使得精化式

A ⇐ **if** a **then** ok **else if** b **then** P. B **else** Q. C**fi fi**

B ⇐ **if** a **then** ok **else if** b **then** R. C **else** S. A **fi fi**

C ⇐ **if** a **then** ok **else if** b **then** T. A **else** U. B **fi fi**

都为定理。于是A可执行如下（用冒号作标记）：

A: **if** a **then go to** D **else if** b **then** P. **go to** B **else** Q. **go to** C **fi fi**.
B: **if** a **then go to** D **else if** b **then** R. **go to** C **else** S. **go to** A **fi fi**.
C: **if** a **then go to** D **else if** b **then** T. **go to** A **else** U. **go to** B **fi fi**.
D: ok

我们已经替换了精化式并使用标记和**go to**进行调用。

(a) 表明不能够在不引入新变量的情况下，替换精化式并使用**while**-循环进行调用（在这个例子里。

(b) 表明可以在引入新变量的情况下，替换精化式并使用**while**-循环进行调用（在这个例子里。

311 规范 **wait** *w* 表示执行时间 *w* 的延迟，其中 *w* 为一个时间长度，而不是一个时刻。试用以下变量形式化这个规范并实现它。

(a) 递归时间度量。

(b) 实数时间度量（可假设所需的任意正的运算时间）。

312 定义 **wait until** *w* = *t*:= *max t w*，其中 *t* 是个广义自然数时间变量，*w* 是个广义自然数表达式。

(a)√ 证明 **wait until** *w* ⇐ **if** $t \geq w$ **then** *ok* **else** *t*:=*t*+1. **wait until** *w* **fi**

(b) 现在假设 *t* 是个广义非负实数时间变量，*w* 是个广义非负实数表达式。重新恰当地定义 **wait until** *w*，并用实数时间度量（可假设所需的任意正的运算时间）来精化它。

313 我们是否可以用以下公理定义已编程表达式 *P* **result** *e*：

(a) $x' = (P \textbf{ result } e) = P.\ x' = e$

(b) $x' = (P \textbf{ result } e) \Longrightarrow P.\ x' = e$

(c) $P \Rightarrow (P \textbf{ result } e) = e'$

(d) $x' = (P \textbf{ result } e) \wedge P \Rightarrow x' = e'$

314 令 *a* 和 *b* 为有理数变量，定义过程 *P* 为

$$P = \langle x, y : rat \rightarrow \textbf{if } x=0 \textbf{ then } a:=x \textbf{ else } a:=x \times y.\ a:=a \times y \textbf{ fi} \rangle$$

(a) *P a*(1/*b*) 精化 *a′ =b′*的精确前置条件是什么？

(b) 对参变量是 “活性”(eager)求值还是 “惰性”(lazy)求值会同时影响到程序设计理论和程序设计语言的实现。试讨论它们之间的差别。

315 “变参调用” (Call-by-value-result) 描述了这样一个参数，它从一个参数变量处获取一个初始值，然后作为一个局部变量参与运算，最后将它的终止值返回给原来的参数变量，因此它本身一定是个变量。形式化定义 “变参调用”。讨论它的优点和缺点。

316 (按名调用) 以下是一个过程对一个参变量的应用。

$$\langle x: int \cdot a:=x.\ b:=x \rangle\ (a+1)$$

假设，由于失误，我们将过程体中 *x* 的两次出现都置换成了参变量。这时我们得到什么？我们应该得到什么？(该错误称为 “按名调用”)

317 (卫式命令) 在 “Dijkstra 的小型语言”中有一个条件程序，语法如下：

$$\textbf{if } b \rightarrow P \ [] \ c \rightarrow Q \textbf{ fi}$$

其中 *b* 和 *c* 为二元式，*P* 和 *Q* 为程序。它可以如下执行：如果一开始 *b* 和 *c* 中只有一个为真，那么执行相应的程序；如果 *b* 和 *c* 一开始同时为真，那么执行 *P* 和 *Q* 中的任意

一个(任意选择)；如果 b 和 c 一开始都不为真，那么执行完全是任意的。

(a) 用本书中介绍的记号把这个程序表示为一个规范。

(b) 用本书中介绍的记号精化这个规范为一个程序。

318 (布尔的布尔) 如果 T=1，⊥=0，表示

(a) $\neg a$

(b) $a \wedge b$

(c) $a \vee b$

(d) $a \Rightarrow b$

(e) $a \Leftarrow b$

(f) $a = b$

(g) $a \neq b$

仅使用以下符号（任意数量）

(i) 0 1 a b ( ) + - ×

(ii) 0 1 a b ( ) – *max min*

这是 7×2=14 个问题。

319 令 n 为一个数，令 P，Q 和 R 为概率规范。证明

(a) $n{\times}P.\ Q\ =\ n{\times}(P.\ Q)\ =\ P.\ n{\times}Q$

(b) $P{+}Q.\ R\ =\ (P.\ R) + (Q.\ R)$

(c) $P.\ Q{+}R\ =\ (P.\ Q) + (P.\ R)$

(d) $x{:=}\ e.\ P\ =\ \langle x{\to}P\rangle e$

320 证明

(a) 当 n 以概率 2^{-n} 在 $nat+1$ 上变化时 n^2 的平均值为 6。

(b) 当 n 以概率 $(5/6)^n \times 1/6$ 在 nat 上变化时平均值为 5。

321 (硬币) 重复投掷硬币直至得到头像。证明投掷 n 次的概率为 2^{-n} 。如果适当定义 R，程序为

$$R \Leftarrow t := t+1.\ \textbf{if}\ rand\,2\ \textbf{then}\ ok\ \textbf{else}\ R\ \textbf{fi}$$

322 一个硬币被重复投掷，在每次投掷时，一个自然数变量不是减少 1 就是不变。需要多少次投掷直到变量值为 0?

323 (纸牌) 你从台面上发到一张牌。它的值是 1 到 13 中的一个。你可以只有一张牌就停止，也可以要第二张牌。你的目标是得到尽可能接近 14 的总值，但不能超过 14。你的策略是如果第一张牌小于 7 就要第二张牌。假设每张牌等概率发取。

(a)√ 求你的每一种总值的概率。

(b) 求你的总值的平均值。

324 (醉酒) 一个醉汉在试图走回家。每个时间单元，醉汉可能向前走一个距离单元，站在原地，或往回走一个距离单元。在 n 个时间单元后，醉汉在哪里？

(a) 每个时间单元，向前走的概率是 2/3，呆在原地的概率是 1/3。这个醉汉没有往回走。

(b) 每个时间单元，向前走的概率是 1/4，呆在原地的概率是 1/2，往回走的概率是 1/4。

(c) 每个时间单元，向前走的概率是 1/2，呆在原地的概率是 1/4，往回走的概率是 1/4。

325 (豆先生的袜子) 豆先生想从一个有无限红色和蓝色袜子的抽屉里拿一双匹配的袜子。他先随机拿两只袜子。如果匹配则完成。否则，他随机扔掉一只，再随机从抽屉拿一只，如此重复。那么他拿到两只匹配的袜子需要多少时间？假设从抽屉取任何一种颜色袜子的概率是 1/2，扔掉任何一种颜色袜子的概率也是 1/2。

326 (翻动开关) 一个双向开关被翻动了一些次数。每一次（包括一开始，第一次翻动之前）有 1/2 的几率继续翻动，1/2 的几率停止。开关停在最初的状态的概率是 2/3，它停在另一边的概率是 1/3。

(a) 用概率分布来表示最终状态。

(b) 使这个分布等于一个描述这些翻动的程序。

(c) 证明这个等式。

327√ (色子) 如果你重复投掷两个六面的色子直到它们相等，需要多时间？

328 (构造 1/2) 假设投掷一个硬币，但怀疑这个硬币可能有偏重误差。假设以头像着陆的概率为 p。理想的是硬币以头像着陆的概率为 1/2。这里有一个达到理想状态的方法。投掷硬币两次。如果结果不同，用第一个结果。如果结果相同，重复实验，直到两次结果不同，然后采用第一对结果不同的第一个结果。证明这个过程有效，并求出花费多少时间。

329 (蒙提霍尔问题) 蒙提霍尔是一个游戏节目的主持人，在这个游戏里有三扇门。奖励藏在其中一扇门内。参赛者选择一扇门。然后蒙提打开其中一扇门，但不打开藏有奖励的那扇门，也不打开参赛者选择的那扇门。蒙提问参赛者是否乐意更换所选的门，还是坚持原来的选择。参赛者应该怎么做？

330 (变色龙) 有 c 条变色龙，其中 r 条是红色的，其余是蓝色的。在每次钟的滴答声响起，任意选择一条变色龙，并且

(a) 它改变了颜色。直到所有变色龙都有相同颜色时，需要多长时间？

(b) 其中一条拥有另一种颜色的变色龙改变了颜色，以匹配随机选的那条变色龙的颜色。直到所有变色龙颜色相同，需要多长时间？

331 (神奇的平均值) 考虑以下看起来不正确的程序，其中 p 是正自然数变量。

loop = **if** *rand* 2 **then** p:= 2×p. t:= t+1. *loop* **else** *ok* **fi**

重复得投掷一枚硬币；每次都看到头像，使 p 翻倍，到第一次看到尾部时停下来。

(a) *loop* 的分布是什么？
(b) *t* 的最终值的平均值是什么？
(c) *p* 的最终值的平均值是什么？

332 程序设计语言的什么特征干扰了程序设计理论的使用？在什么方面进行了干扰？

333 我们提议定义一个新的程序设计连接符 $P \blacklozenge Q$。♦有何基本特性？为什么？

----------程序设计语言结束

10.6 递归定义

334 证明$\neg -1 : nat$。提示：需要利用归纳。

335 证明: $\forall n : nat \cdot Pn = \forall n : nat \cdot \forall m : 0,..n \cdot Pm$ 。

336√ 证明一个奇数自然数的平方为 $8m+1$，其中 *m* 为某个自然数。

337 证明每一个正整数都是质数的乘积。这里“乘积”表示将任意自然数数目的数(不一定不同)全部相乘。“质数”表示只有两个因子的自然数。

338 这里有个论点，即可以“证明”在任意一个人群组中，每个人都具有相同的年龄。“证明”是通过对组的大小进行归纳得到。归纳的基础是在任意一个大小为 1 的组中，所有的人都具有相同的年龄。或者我们也可以等价地使用大小为零的组作为归纳的基础。归纳假设是：假设在所有大小为 *n* 的组中，所有的人都具有相同的年龄。现在考虑大小为 $n+1$ 的组, 令其中的人分别 $p_0, p_1, ..., p_n$，由归纳假设可以得到：在大小为 *n* 的子组 $p_0, p_1, ..., p_{n-1}$ 中，每一个人都具有相同的年龄；明确地，它们都和 p_1 具有相同的年龄。同时，在大小为 *n* 的子组 $p_1, p_2, ..., p_n$ 中，每一个人都具有相同的年龄；明确地，它们也都和 p_1 具有相同的年龄。因此，所有 $n+1$ 个人都具有相同的年龄。形式化这个论点并找出它的错误之处。

339 这里有一个可能的 *nat* 的另一种构造公理。

$$0, 1, nat + nat : nat$$

(a) 与之相随的归纳公理是什么？
(b) 该构造公理和(a)部分的归纳公理是否满足成为 *nat* 的定义？

340 第六章给出了四个谓词形式的 *nat* 归纳。证明它们全都等价。

341 证明 $nat = 0,..\infty$ 。

342 以下是关于束 *bad* 的一个构造公理和一个归纳公理：

$$(\S\, n{:}\ nat\cdot \neg\, n{:}\ bad){:}\ bad$$

$$(\S\, n{:}\ nat\cdot \neg\, n{:}\ B){:}\ B \Longrightarrow bad{:}\ B$$

(a) 这些公理一致吗？

(b) 利用以上公理可以证明以下不动点等式吗？

$$bad = \S\, n{:}\ nat\cdot \neg\, n{:}\ bad$$

343 证明以下各式，其中量词的作用范围是 *nat*。

(a) $\neg\exists i, j\cdot j\neq 0 \wedge 2^{1/2} = i/j$ 2 的平方根是无理数。

(b) $\forall n\cdot (\sum i{:}\ 0,..n\cdot 1) = n$

(c) $\forall n\cdot (\sum i{:}\ 0,..n\cdot i) = n \times (n-1) / 2$

(d) $\forall n\cdot (\sum i{:}\ 0,..n\cdot i^3) = (\sum i{:}\ 0,..n\cdot i)^2$

(e) $\forall n\cdot (\sum i{:}\ 0,..n\cdot 2^i) = 2^n\text{-}1$

(f) $\forall n\cdot (\sum i : 0,..n\cdot i\times 2^i) = (n-2)\times 2^n + 2$

(g) $\forall n\cdot (\sum i{:}\ 0,..n\cdot (-2)^i) = (1-(-2)^n) / 3$

(h) $\forall n\cdot n\geq 10 \Rightarrow 2^n > n^3$

(i) $\forall n\cdot n \geq 4 \Rightarrow 3^n > n^3$

(j) $\forall n\cdot n \geq 3 \Rightarrow 2\times n^3 > 3\times n^2 + 3\times n$

(k) $\forall a, d\cdot \exists q, r\cdot d \neq 0 \Rightarrow r<d \wedge a = q\times d + r$

(l) $\forall a,b\cdot a \leq b \Rightarrow (\sum i : a,..b\cdot 3^i) = (3^b - 3^a)/2$

(m) $\forall n\cdot (n+1)^{nat} : nat\times n + 1$

344 说明我们可以使用不动点构造式和以下式子定义 *nat*

(a) $\forall n : nat\cdot 0 \leq n < n+1$

(b) $\exists m : nat\cdot \forall n : nat\cdot m \leq n < n+1$

345 令 R 是自然数上的关系 $R{:}\ nat\rightarrow nat\rightarrow bin$，它对于第二个参数是单调的

$$\forall i, j\cdot Rij \Rightarrow Ri(j+1)$$

证明 $\exists i\cdot \forall j\cdot Rij = \forall j\cdot \exists i\cdot Rij$ 。

346√ 假设我们用普通的构造式和归纳式定义了 *nat* 。

$$0, nat+1 : nat$$

$$0, B+1 : B \Rightarrow nat : B$$

证明不动点构造式和归纳式为定理。

$$nat = 0, nat+1$$

$$B = 0, B+1 \Rightarrow nat : B$$

347 (不动点定理) 假设我们用不动点构造式和归纳式定义了 *nat* 。

$$nat = 0, nat+1$$

$$B = 0, B+1 \Rightarrow nat : B$$

证明普通的构造式和归纳式为定理。警告：这很难，需要用到极限。

$$0, nat+1 : nat$$

$$0, B+1 : B \Rightarrow nat : B$$

348 (标尺) 标尺的形成如下：一个垂直短线 | 是一个标尺。如果加上一段水平短线一，再加上一个垂直短线丨，就得到了另一个标尺。于是前几个标尺为：丨，|—|，|—|—|，|—|—|—|，依次类推。以这种方式形成的任意两个标尺都不相同，并且不存在其他形式的标尺。试问需要哪些公理才能定义一个标尺束，并满足它包含所有且仅包含标尺？

349 如果函数 f 满足：$\forall i, j \cdot i \le j \Rightarrow f\,i \le f\,j$， 则它是单调的。

(a) 证明 f 是单调的当且仅当 $\forall i, j \cdot f\,i < f\,j \Rightarrow i < j$

(b) 令 $f: int \to int$ 。证明 f 是单调的当且仅当 $\forall i \cdot f\,i \le f\,(i+1)$。

(c) 令 $f: int \to int$，并满足 $\forall n \cdot ffn < f\,(n+1)$。证明 f 是个恒等函数。提示：首先证明 $\forall n \cdot n \le fn$。然后证明 f 是单调的，然后再证明 $\forall n \cdot f\,n \le n$ 。

350 斐波那契数 $fib\ n$ 定义如下：

$$fib\ 0 = 0$$

$$fib\ 1 = 1$$

$$fib\ (n+2) = fib\ n + fib\ (n+1)$$

证明

(a) $fib\ (gcd\ n\ m) = gcd\ (fib\ n)\ (fib\ m)$
其中 gcd 为最大公约数。

(b) $fib\ n \times fib\ (n+2) = (fib\ (n+1))^2 - (-1)^n$

(c) $fib\ (n+m+1) = fib\ n \times fib\ m + fib\ (n+1) \times fib\ (m+1)$

(d) $fib\ (n+m+2) = fib\ n \times fib\ (m+1) + fib\ (n+1) \times fib\ m + fib\ (n+1) \times fib\ (m+1)$

(e) $fib\ (2 \times n+1) = (fib\ n)^2 + (fib\ (n+1))^2$

(f) $fib\ (2 \times n+2) = 2 \times fib\ n \times fib\ (n+1) + (fib\ (n+1))^2$

(g) $fib\ (k \times n) : nat \times fib\ n$

351 从公理 $P = 1, x, -P, P+P, P \times P$ 中可以证明 P 中含有什么样的元素？证明

$$2 \times x^2 - 1 : P$$

352 表示一个包含 2 的整数次方的束，即 2^{int}，不许使用指数运算。可以引入辅助名。

353 求分别满足以下条件的最小束：

(a) $B = 0, 2 \times B + 1$

(b) $B = 2, B \times B$

354 束 $this$ 由以下构造和归纳公理定义：

2, $2\times this$: $this$

2, $2\times B$: $B \Rightarrow this$: B

束 *that* 由以下构造和归纳公理定义：

2, $that\times that$: $that$

2, $B\times B$: $B \Rightarrow that$: B

证明 $this=that$。

355 令 n 为一个自然数，从不动点等式：

$ply=n, ply+ply$

可通过递归构造获得一个束序列 ply_i。

(a) 形式化表述 ply_i (不需证明)。

(b) 用语言表述 ply_i 。

(c) 什么是 ply_∞?

(d) ply_∞是一个解吗？如果是，它是唯一解吗？

356 令 $A\backslash B$ 为束 A 和束 B 的差。运算符\的优先级为 4，它由以下公理定义：

$x: A \backslash B = x: A \wedge \neg x: B$

对以下每一个不动点等式，由递归构造可以生成什么？它满足这些不动点等式吗？

(a) $Q = nat\backslash(Q+3)$

(b) $D = 0, (D+1)\backslash(D-1)$

(c) $E = nat \setminus (E+1)$

(d) $F = 0,(nat \setminus F)+1$

357 对以下每一个不动点等式，由递归构造可以生成什么？它满足这些不动点等式吗？

(a) $P = \S n: nat\cdot\ n=0 \wedge P=null \vee n: P+1$

(b) $Q = \S x: xnat\cdot\ x=0 \wedge Q=null \vee x: Q+1$

358 以下是一对相互递归的等式。

$even = 0$，$odd+1$

$odd = even + 1$

(a) 由递归构造可以生成什么？说明构造过程。

(b) 还需要进一步的公理以保证 *even* 仅包含偶数，*odd* 仅包含奇数吗？如果需要，需要哪些公理？

359(a) E 为未知的，找到 $E, E+1 = nat$ 的三个解。

(b) 现在加入归纳公理 $B, B+1 = nat \Rightarrow E : B$。问 E 是什么？

360 由构造公理 0 , $1-few: few$

(a) 可以构造哪些元素？

(b) 给出三个解（*few* 是未知的）。

(c) 给出相应的归纳公理。

(d) 说明哪个解是构造公理和归纳公理表示的。

361 探讨以下的不动点等式

$$strange = \S\, n{:}\ nat \cdot \forall m{:}\ strange \cdot \neg\, m{+}1{:}\ n{\times}nat$$

362 令 *truer* 为一个由二元字符串组成的束，它由以下构造和归纳公理定义：

$$\mathsf{T}, \bot;\ truer;\ truer{:}\ truer$$

$$\mathsf{T}, \bot;\ B;\ B{:}\ B \Rightarrow truer{:}\ B$$

给定一个二元字符串，写一段程序判断该字符串是否在 *truer* 中。

363 (字符串) 如果 S 是一个字符串的束，那么 $*S$ 是由 S 中的任意多个任意串以任意秩序连接成的所有串的束。

(a) 使用构造和归纳定义 $*S$ 。

(b) 证明 $**S = *S$ 。

364 以下是由类型为 *T* 的项组成的表的构造和归纳公理：

$$[nil]\ ,\ [T]\ ,\ list{+}list{:}\ list$$

$$[nil]\ ,\ [T]\ ,\ L{+}L\ :\ L \Longrightarrow list{:}\ L$$

证明 *list* = [*T]。

365 (巴科斯–诺范式) 巴科斯–诺范式是一个语法形式，它的语法规则写成以下形式：

$$\langle exp\rangle ::= \langle exp\rangle + \langle exp\rangle \mid \langle exp\rangle \times \langle exp\rangle \mid 0 \mid 1$$

用我们的语法形式应该写成：

$$exp\ =\ exp;\ "+";\ exp,\ \ exp;\ "\times";\ exp,\ \ "0",\ \ "1"$$

试用类似的公理形式定义以下各式。

(a) 回文：正向读和反向读都相同的文本。使用一个双符号字母表。

(b) 长度为奇数的回文。

(c) 所有在字符 *“a”* 后跟有同样数目的字符 *“b”* 的正文。

(d) 所有在字符 *“a”* 后跟有至少同样数目的字符 *“b”* 的正文。

366 用不动点构造公理和关联的不动点归纳公理定义语言 *lang*：

$$lang\ =\ nil,\ \text{“(”};\ lang;\ \text{“)”},\ lang;lang$$

(a) 非形式化的，这个语言描述了什么？

(b) 写一个等价的，不递归的此语言的定义。提示：以 § 开头，用一个对文本中字符出现次数进行计数的谓词。

367 (二元自然数) 令 0 和 1 为两个新值。定义一个新空序 nil 和一个新的连接符号; （优先

级为 5），这个符号制造了一些 0 和 1 的序，并定义了一个新的加号＋（优先级为 4）作用于以下的序。

$nil; a = a = a; nil$

$(a; b); c = a; (b; c)$

$b＋0$

$(b; 0) ＋ 1 = b; 1$

$(b; 1) ＋ 1 = (b＋1); 0$

$a＋b = b＋a$

$(a＋b) ＋c = a＋(b＋c)$

定义二元自然数 *binat* 如下。

$0, binat＋1: binat$

$0, B＋1: B \Rightarrow binat: B$

对于 n：0，1 和 b: *binat* 证明

(a) $b＋b = (b; 0)$

(b) $(b; n) = b＋b＋n$

368 (十进制小数) 利用递归数据定义，定义一个包含所有十进制小数的束。它们是可以用带有一个小数点的有限十进制数字串表示的有理数。注意：你是在定义一个数束而不是文本束。

369 第 6.1 节利用以下不动点等式定义了程序 *zap*：

zap ＝ **if** x=0 **then** y:=0 **else** x:=x–1. t:= t+1. *zap* **fi**

(a)√ 需要什么样的公理才能使 *zap* 成为最弱不动点？

(b) 需要什么样的公理才能使 *zap* 成为最强不动点？

(c) 6.1 节为这个等式提供了六个解。试找出更多的解。提示：在 ∞ 时刻会发生奇怪的事情。

370 令以下所有变量都为整数。在以下各式中加入递归时间，使用递归构造式，找出它们的一个不动点。

(a) *skip* ＝ **if** i≥0 **then** i:=i–1. *skip*. i:=i+1 **else** *ok* **fi**

(b) *inc* ＝ *ok* ∨ (i:= i+1. *inc*)

(c) *sqr* ＝ **if** i=0 **then** *ok* **else** s:=s + 2× i – 1. i:=i–1. *sqr* **fi**

(d) *fac* ＝ **if** i=0 **then** f:=1 **else** i:=i–1. *fac*. i:=i+1. f :=f×i **fi**

(e) *chs* ＝ **if** a=b **then** c:=1 **else** a:=a–1. *chs*. a:=a+1. c:=c×a / (a-b) **fi**

371 令以下所有变量都为整数。在以下各式中加入递归时间，使用任何方法，找出它们的一个不动点。

(a) *walk* ＝ **if** i≥0 **then** i:=i–2. *walk*. i:=i+1. *walk*. i:=i+1 **else** *ok* **fi**

(b) *crawl* ＝ **if** i≥0 **then** i:=i–1. *crawl*. i:=i+2. *crawl*. i:=i–1 **else** *ok* **fi**

(c) run = **if** *even i* **then** $i:=i/2$ **else** $i:=i+1$ **fi**. **if** $i=1$ **then** *ok* **else** *run* **fi**

372 探讨若以以下时刻作为构造的起点会如何影响递归构造：

(a) $t' = \infty$

(b) $t := \infty$

373 令 x 为一个整数变量。利用递归时间度量，在下式中加入时间，并找出最强的可实现的规范 P 和 Q：

$$P \Leftarrow x' \geq 0.\ Q$$

$$Q \Leftarrow \textbf{if } x=0 \textbf{ then } ok \textbf{ else } x:=x-1.\ Q \textbf{ fi}$$

假设 $x' \geq 0$ 不占用任何时间。

374 令 x 为一个整数变量。

(a) 利用递归时间度量，在下式中加入时间，并找出最强的可实现的规范 S:

$$S \Leftarrow \textbf{if } x=0 \textbf{ then } ok \textbf{ else if } x>0 \textbf{ then } x:=x-1.\ S \textbf{ else } x' \geq 0.\ S \textbf{ fi fi}$$

假设 $x' \geq 0$ 不占用任何时间。

(b) 如果从 $t' \geq t$ 开始递归构造会得到什么？

375 证明以下三种定义 R 的方式是等价的。

$$R = ok \vee (R.S)$$

$$R = ok \vee (S.R)$$

$$R = ok \vee S \vee (R.R)$$

376 证明 **while–**循环的逐步精化定律和部分精化定律。

377 证明

$$\forall \sigma, \sigma' \cdot \textbf{if } b \textbf{ then } P.\ t:=t+1.\ W \textbf{ else } ok \textbf{ fi} \Leftarrow W$$

$$\Leftarrow \forall \sigma, \sigma' \cdot \textbf{while } b \textbf{ do } P \textbf{ od} \Leftarrow W$$

与 **while 构造**公理是等价的，并且构造公理和归纳公理合在一起可以表示为

$$\forall \sigma, \sigma' \cdot t' \geq t \textbf{ if } b \textbf{ then } P.\ t:=t+1.\ W \textbf{ else } ok \textbf{ fi} \Leftarrow W$$

$$= \forall \sigma, \sigma' \cdot \textbf{while } b \textbf{ do } P \textbf{ od} \Leftarrow W$$

378 **记号 do** P **while** b **od** 表示一个如下执行的循环结构。首先 P 执行；然后计算 b，如果为 T 则重复执行，如果为⊥则执行结束。使用构造公理和归纳公理定义 **do** P **while** b **od**。

379 令状态包括二元变量 b 和 c。令

W = **if** b **then** P. W **else** ok **fi**

X = **if** $b \vee c$ **then** P. X **else** ok **fi**

(a) 找出 $W.\ X = X$ 的一个反例。

(b) 现在令 W 和 X 为以上等式的最弱解，证明 $W.\ X = X$。

380 对于自然数变量 n，忽略时间，找到三个满足下式的规范 P。

$P = P.\ n = 2 \times n'$

381 x 为实数变量，考虑等式：

$P = P.\ x := x^2$

(a) 找出 P 的 7 个不同解。

(b) 从 T 开始，利用递归构造可以得到什么样的解？这个解是最弱解吗？

(c) 如果加入时间变量，从 $t' \geq t$ 开始，利用递归构造可以得到什么样的解？它是最强可实现的解吗？

(d) 现在令 x 为整数变量，重做以上问题。

382 假设我们利用以下的一般构造和归纳式定义 **while** b **do** P **od**，**忽略时间。**

if b **then** P. **while** b **do** P **od else** ok **fi** $\Leftarrow$ **while** b **do** P **od**

$\forall \sigma, \sigma' \cdot$ **if** b **then** P. W **else** ok **fi** $\Leftarrow W \implies \forall \sigma, \sigma' \cdot$ **while** b **do** P **od** $\Leftarrow W$

证明以下不动点构造式和归纳式为定理。

while b **do** P **od** = **if** b **then** P. **while** b **do** P **od else** ok **fi**

$\forall \sigma, \sigma' \cdot W =$ **if** b **then** P. W **else** ok **fi** $\implies \forall \sigma, \sigma' \cdot$ **while** b **do** P **od** $\Leftarrow W$

383 假设我们利用以下不动点构造和归纳式定义 **while** b **do** P **od**，**忽略时间。**

while b **do** P **od** = **if** b **then** P. **while** b **do** P **od else** ok **fi**

$\forall \sigma, \sigma' \cdot W =$ **if** b **then** P. W **else** ok **fi** $\implies \forall \sigma, \sigma' \cdot$ **while** b **do** P **od** $\Leftarrow W$

证明以下的一般构造式和归纳式为定理。

if b **then** P. **while** b **do** P **od else** ok **fi** $\Leftarrow$ **while** b **do** P **od**

$\forall \sigma, \sigma' \cdot$ **if** b **then** P. W **else** ok **fi** $\Leftarrow W \implies \forall \sigma, \sigma' \cdot$ **while** b **do** P **od** $\Leftarrow W$

警告：这个问题很难，需要用到极限。

384 忽略时间，使用练习 378 中的定义证明:

(a) **do** P **while** b **od** = P. **while** b **do** P **od**

(b) **while** b **do** P **od** = **if** b **then do** P **while** b **od else** ok **fi**

(c) $(\forall \sigma, \sigma' \cdot D =$ **do** P **while** b **od**$) \wedge (\forall \sigma, \sigma' \cdot W =$ **while** b **do** P **od**$)$

= $(\forall \sigma, \sigma' \cdot (D = P.\ W)) \wedge (\forall \sigma, \sigma' \cdot W =$ **if** b **then** D **else** ok **fi**$)$

----------递归定义结束

10.7 理论设计与实现

385 (小器件) 小器件的理论以某种新的语法和公理的形式提出。一个小器件的实现已经写

出。

(a) 我们如何知道小器件的理论是否一致？

(b) 我们如何知道小器件的理论是否完备？

(c) 我们如何知道小器件理论的实现是否是正确的？

386 √ 实现数据－堆栈理论使得以下两个二元表达式成为反定理：

$$pop\ empty = empty$$
$$top\ empty = 0$$

387 证明以下定义实现了简单的数据－堆栈理论。

$$stack = [nil],\ [stack;\ X]$$
$$push = \langle s{:}\ stack \rightarrow \langle x{:}\ X \rightarrow [s;\ x]\rangle\rangle$$
$$pop = \langle s{:}\ stack \rightarrow s\ 0\rangle$$
$$top = \langle s{:}\ \mathrm{stack} \rightarrow \mathrm{s}\ 1\rangle$$

388 (弱 数据－堆栈) 在 7.1.3 节中我们设计了一个程序－堆栈理论，它很弱，使得我们可以加入公理对压入和弹出操作进行计数而不引起不一致性。设计一个类似的弱数据－堆栈理论。

389 (数据－队列实现) 实现 7.0 节中提出的数据－队列理论。

390 (滑动) 滑动数据结构利用以下公理说明了名词 *slip*：

$$slip = [X;\ slip]$$
$$B = [X;\ B] \Rightarrow B{:}\ slip$$

其中 X 是一个给定的数据类型。它可以实现吗？

391 证明 7.1.1 节中给出的程序－栈实现满足 7.1.0 节的程序－栈公理。

392 如下实现弱程序－栈理论：实现者变量是一张只增不减的表。一个弹出的项必须被表注为废料。

393 (线性代数) 设计一个线性代数理论。它应该包含数量、向量、矩阵和、矩阵积和矩阵内积。实现这个理论并给出证明。

394 (普通树) 普遍来讲，树的每一个节点可能有任意数量的子树。

(a) 为普通树设计一个数据定理。

(b) 实现你的定理。

(c) 证明你的实现。

395 (叶计数) 写一段程序，计算一个二叉树的叶个数。

396 (叶状树) 叶状树是这样的一个树，其信息仅驻留在叶子上。试为一个二叉叶状数据树设计合适的公理。

397 (括号) 给定一段字符文本 *t*，它由以下字母组成：“*x*”，“(”，“)”，“[”，“]”。写一段程序，判断 *t* 是否是一个括号配对和嵌套都良好的文本。

398 (有界栈) 根据我们的公理，一个栈具有无限能力以容纳我们压入的项。一个有界的栈与此类似，所不同的是只能容纳有限个数的项。

(a) 为一个有界数据栈设计公理。

(b) 为一个有界程序栈设计公理。

(c) *limit* 可以是 0 吗？

399 (有界队列) 根据我们的公理，一个队列具有无限能力以容纳加入的项。一个有界的队列与此类似，所不同的是只能加入有限个数的项。

(a) 为一个有界数据队列设计公理。

(b) 为一个有界程序队列设计公理。

(c) *limit* 可以是 0 吗？

400 (可重置变量) 一个可重置变量定义如下。引入 3 个新名词：*value*(类型为 *X*)，*set*(带有一个类型为 *X* 的参数的过程)，和 *reset*(一段程序)。以下是它们的公理：

$$value' = x \Leftarrow set\ x$$

$$value' = value \Leftarrow set\ x.\ reset$$

$$reset.\ reset = reset$$

实现这种数据结构并给出证明。

401 (循环表) 设计循环表的公理。它应该具有如下操作：创建一个空表，将表向下移一个位置(沿循环方向表的第一项在最后一项后面)，在当前位置插入一项，删除当前的项，并返回当前的项。

402 一个特别的程序表具有如下的操作：

- 操作 *mkempty* 使得表为空
- 操作 *extend x* 将项 *x* 并接到表后
- 操作 *swap i j* 交换表索引 *i* 和 *j* 处的项
- 表达式 *length* 返回表的长度
- 表达式 *item i* 返回索引 *i* 处的项

(a) 写出定义这种程序表的公理。

(b) 实现这种程序表并给出证明。

403 树可通过以广度优先次序列出它的项来实现。

(a) 用一个由其项组成的表来表示一个二叉树，并满足：根的索引为 0，在索引 n 处的项的左子树和右子树的根分别为 $2\times n+1$ 和 $2\times n+2$。

(b) 证明你的实现。

(c) 将这个实现普遍化到至多具有 k 个分支的树，其中 k 可以是任意数（但必须是常数）。

404 (混合树) 第 7 章提出了数据树和程序树理论。设计一种只有一个树结构的混合树理论，所以这个树结构可以是一个带有程序操作的实现者变量，但可能有许多指针指向这个树，因此它们是数据指针(它们也可能是数据堆栈)。

405 (堆) 堆是具有如下性质的树：其根是最大项，其子树还是堆。

(a) 形式化表述这个堆的性质。

(b) 写一个函数 *heapgraft*，从两个给定堆和一个新项生成一个新堆。它可能用到函数 *graft*，而且可能根据生成新堆的需要重排原来堆的项。

406 (二分查找树) 一个二分查找树是一个具有如下属性的二叉树：其左子树中的所有项都比根项小，其右子树中的所有项都比根项大，并且这两个子树也都分别是二分查找树。

(a) 形式化表述这个二分查找树的性质。

(b) 有多少个二分查找树是由三个项组成？

(c) 写一段程序，在一个二分查找树中找出一项。

(d) 写一段程序，在一个二分查找树中加入一项作为一个新叶节点。

(e) 写一段程序，将一个二分查找树中的一列表项排序。

(f) 写一段程序，判断两个二分查找树是否有相同的项。

407 (聚会) 一个公司的结构是一棵树，雇员是树的节点。每个雇员，除了根节点，都有树中的父节点为其老板。每个雇员都一个欢乐度（一个数）表示他们在聚会上获得多少快乐。但没有人会与老板一起参加聚会。写一个程序邀请一些雇员来参加聚会，使总的欢乐度为最大。

408 (插入表) 插入表是一个与表相似的数据结构，但是带有一个相关的插入点。

```
[...; 4 ; 7; 1 ; 0 ; 3 ; 8 ; 9 ; 2 ; 5 ;...]
                ↑
             插入点
```

insert 将一个项放在插入点的左边(位于原有两项之间)。*erase* 将插入点左边一项删除，并关闭该表。*item* 返回插入点左边一项。*forward* 将插入点向右移一项。*back* 将插入点向左移一项。

(a) 设计一个双向无穷数据插入表公理。

(b) 设计一个双向无穷程序插入表公理。

(c) 设计一个有限数据插入表公理。

(d) 设计一个有限程序插入表公理。

409 (程序表) 一个程序表具有相关索引，以及以下操作：*item* 给出索引所在项的值；*set x*

把索引所在项的值变为 *x*；*goLeft* 把索引左移一项；*goRight* 把索引右移一项。

(a) 为双重无限程序表设计公理。

(b) 运用(a)中你的理论，证明

$$goLeft.\ set\ 3.\ goRight.\ set\ 4.\ goLeft \implies item' = 3$$

410 实现节 7.1.5 中的程序－树理论，其中树在所有方向都是无限的。在任何时间，只有被访问的树的部分需要表示。

411√ (语法分析) 定义 *E* 为一个字符表串束，它满足以下等式：

$$E = [\text{“}x\text{”}],\ [\text{“if”}]\ ;\ E\ ;\ [\text{“then”}]\ ;\ E\ ;\ [\text{“else”}]\ ;\ E\ ;\ [\text{“fi”}]$$

给定一个字符表串，写一段程序判断该串是否在束 *E* 中。

412 每一个程序理论都提供了一种数据结构的一个单一、匿名的实例。如何使得一个程序理论提供一种数据结构的多个实例，就像数据理论一样？

413 一个理论引入了三个名词：*zero, increase, inquire*。它们可通过一个实现表述。令 *u*: *bin* 为使用者变量，令 *v*: *nat* 为实现者变量。公理为

$$zero = v{:=}0$$
$$increase = v{:=}v{+}1$$
$$inquire = u{:=}even\ v$$

根据以下数据转换式将 *v* 替换成 *w*: *bin* 。

(a)√ $w = even\ v$

(b) ⊤

(c) ⊥ (这不是一个数据转换式，因为 $\forall w\cdot \exists v\cdot \bot$不是一个定理，但是照常应用它，看看会发生什么)

414 使用者变量为二元变量 *b*。实现者变量是自然数变量 *x* 和 *y*。运算有：

$$done = b{:=}\ x{=}y{=}0$$
$$step = \textbf{if}\ y{>}0\ \textbf{then}\ y{:=}\ y{-}1\ \textbf{else}\ x{:=}\ x{-}1.\ \textbf{var}\ n{:}\ nat{\cdot}\ y{:=}\ n\ \textbf{fi}$$

使用一个新的实现者自然数变量 *z* 替代两个实现者变量 *x* 和 *y*。

415 一个理论引入了三个名词：*set, flip,* 和 *ask* 。它们可通过一个实现表述。令 *u*: *bin* 为使用者变量，令 *v*: *bin* 为实现者变量，公理为

$$set = v{:=}\ \top$$
$$flip = v{:=}\ \neg v$$
$$ask = u{:=}\ v$$

(a)√ 根据数据转换式 $v = even\ w$, 用 *w*: *nat* 替换 *v*。

(b) 根据数据转换式 $(w{=}0 \Rightarrow v) \wedge (w{=}1 \Rightarrow \neg v)$, 用 *w*: *nat* 替换 *v*, 有何错误？

(c) 根据数据转换式 $(v \Rightarrow w{=}0) \wedge (\neg v \Rightarrow w{=}1)$, 用 *w*: *nat* 替换 *v*, 有何错误？

416 (稀疏数组) 如果一个数组 *A*: [*[**rat*]]的许多项都为 0，那么该数组称为是稀疏的。我们

可以用一个全都由非零项 $A\ i\ j = x \neq 0$ 组成的三元组表$[i\ ;\ j\ ;\ x]$，简洁地表示这样的数组。利用这个想法，找出一个数据转换式，并转换以下程序：

(a) A:= [100*[100*0]]

(b) $i := A\ i\ j$

(c) $A := (i; j) \rightarrow x \mid A$

417 令 a，b 和 c 为二元变量。变量 a 和 b 为实现者变量，c 为用户变量。有以下运算：

$$seta = a := \mathsf{T}$$
$$reseta = a := \bot$$
$$flipa = a := \neg a$$
$$setb = b := \mathsf{T}$$
$$resetb = b := \bot$$
$$flipb = b := \neg b$$
$$and = c := a \wedge b$$
$$or = c := a \vee b$$

该理论必须使用整数变量来重新实现，0 表示 $\bot$，其它所有整数表示 T。

(a) 数据转换式是什么？

(b) 转换 *seta* 。

(c) 转换 *flipa* 。

(d) 转换 *and* 。

418√ (安全开关) 一个安全开关有三个二元用户变量 a，b 和 c。用户对 a，b 赋值作为开关的输入。开关的输出赋给 c。当两个输入都变化时输出也变化。更精确地说，当两个输入与上次输出时它们的值都不同时，输出的值改变。概念是一个用户可能希望输出的值改变，因而改变其输入值，但在另一个用户也同意输出改变而改变其输入的值之前输出并不会改变。如果第一个用户在第二个用户改变前又改变回去了，那么输出也不改变。

(a) 根据前面的非形式化描述尽可能直接地实现一个安全开关。

(b) 转换(a)的实现，获得一个高效的实现。

419 令 p 为使用者的二元变量，令 m 为实现者的自然数变量。运算允许使用者给实现者变量赋一个值 n，并测试是否实现者变量为质数。

$$assign\ n = m := n$$
$$check = p := prime\ m$$

假设 *prime* 被合理地定义。如果 *prime* 是昂贵的函数，并且 *check* 操作比 *assign* 操作更频繁，可以用以下方法改进问题的解，使 *check* 变得没那么昂贵尽管有可能促使 *assign* 更昂贵。运用数据转换，完成这个改进。

420√ (取一个数) 维持一个代表 “正在使用”的自然数表。有三个操作：

- 使该表为空(初始化)

- 将一个当前不在使用的数赋给变量 n，并将该数加入到该表中(现在它是正在使用的数)
- 已知一个正在使用的数 n，将它从该表中删除(现在它不再是正在使用的数，以后可以被重新使用)。

(a) 使用束的术语实现上述操作。

(b) 使用数据转换将所有束变量替换为自然数变量。

(c) 使用数据转换获得一个分布式的解。

421√ 一个有限队列是一个只有有限的位置存放项的队列。令其界为正自然数 n，令 $Q:[n*X]$ 和 $p:nat$ 为实现者变量。下面是一个实现。

$$makemptyq = p := 0$$
$$isemptyq = p = 0$$
$$isfullq = p = n$$
$$joinx = Qp := x.p := p+1$$
$$leave = \textbf{for } i := 1;..p \textbf{ do } Q(i\text{-}1) := Qi \textbf{ od}.\ p := p\text{-}1$$
$$front = Q0$$

删除队列最前面的一项需要 $p-1$ 的时间将后面的项前移。请转换该队列使所有运算都是常数时间的。

422 (转换的不完备性) 使用者变量为 i，实现者变量为 j，都为 0，1，2 类型。运算为:

$$initialize = i' = 0$$
$$step = \textbf{if } j > 0 \textbf{ then } i := i+1.\ j := j\text{-}1 \textbf{ else } ok \textbf{ fi}$$

用户可以看 i 但不能看 j。用户可以使用 *initialize*，它使 i 从 0 开始，而使 j 从三个值中任意之一开始。然后用户可以重复使用 *step* 并观察到 i 增加 0 次或 1 次或 2 次，然后停止增加，这就有效告诉了用户 j 以哪个值开始。

(a) 说明没有数据转换式可以用二元变量 b 代替 j，使得

$$initialize \quad \text{转换为} \quad i' = 0$$
$$step \quad \text{转换为} \quad \textbf{if } b \wedge i < 2 \textbf{ then } i' = i+1 \textbf{ else } ok \textbf{ fi}$$

转换后的 *initialize* 或将 b 初始化为 T，表示 i 将被增加，或初始化为 $\perp$，表示 i 不会被增加。转换后的 *step* 每次使用测试 b 看 i 是否增加，并检查 $i < 2$ 保证增加后的 i 始终不超过 2。如果 i 增加了，b 重新被赋为它的两个值之一。用户会看到 i 从 0 开始并增加了 0 次或 1 次或 2 次然后停止增加，正如原来的规范一样。

(b) 使用数据转换式 $b = (j > 0)$ 转换 *initialize* 和 $i + j = k \Rightarrow step$，其中 k 是一个常数，$k:0,1,2$。

423 一棵二叉树可以以宽度优先顺序存储在一个节点表中。传统地，根存放在索引 1 的位置，索引为 n 的节点的左子节点的索引为 $2\times n$，右子节点的索引为 $2\times n+1$。假设使用者变量为 $x:X$，实现者变量为 $s:[*X]$ 和 $p:nat+1$，运算有

$$goHome = p := 1$$
$$goLeft = p := 2\times p$$
$$goRight = p := 2\times p+1$$

$goUp = p := div\ p\ 2$

$put = s := p \rightarrow x \mid s$

$get = x := s\ p$

现在假设我们决定将整个表向下移动一个索引，这样就不会浪费索引 0。根存放在索引 0 的位置，其子节点为索引 1 和 2，依此类推。寻找必要的数据转换，并使用它转换以上的运算。

424 节 7.1.5 引入了程序－树理论。

(a) 实现这个理论，把所有节点值储存在一个数据结构中。

(b) 转换这个实现使得对 *node* 的赋值不会更新储存节点值的主要数据结构；这种更新出现于当你从一个节点进行 *go* 时。

425 (弱有限程序束) 已知自然数 n，一个维持子束 $0,..n$ 的理论。操作是：*mkempty*，使束为空；*insert x*，把 *x* 插入束中；*remove x*，如果 *x* 在那里就把它去除，以及 *check x*，用对使用者变量 *u* 赋值的方法得出是否 *x* 在那里。

(a) 设计一个足够弱的公理，使得允许其他操作可以加到理论中。

(b) 将你在(a)部分的理论实现为一个二元值表。

(c) 将部分(b)中的实现转换为可以维持自然数表的实现。

426 (横排为主) 计算机存储中的二维数组的通常表示方法是以横排为主的顺序，把横排都串在一起。例如，一个 3×4 的数组

[[5; 2; 7; 3] ;
[8; 4; 2; 0] ;
[9; 2; 7; 7]]

被表示为5; 2; 7; 3; 8; 4; 2; 0; 9; 2; 7; 7。

(a) 已知自然数n和m，找到一个数据转换式，把一个$n\times m$的数组A转换成它的横排为主的表示B。

(b) 用你的转换式，转换$x := A\ y\ z$，其中x，y和z为使用者变量。

427 令u为二元使用者变量。令a和b为原本的二元实现者变量。使用0代表⊥，非零整数代表T的协议（来自C语言）把a和b替换成新的整数实现者变量x和y。

(a) 转换式是什么？

(b) 转换$a := \neg a$。

(c) 转换$u := a \wedge b$。

428 (霍夫曼编码) 已知一个有限集合的信息，以及每个信息出现的概率。

(a) 写一段程序，找到每个信息的二元编码。它必须有可能清楚地把任何顺序的一些0和1解码成一连串的信息，并且平均代码长度（根据信息频率得出）必须最短。

(b) 写一段伴随程序，产出(a)中产生的代码的解码器。

429 使用者变量为二元值b和自然数x。使用实现者变量L: [*nat]和i: *nat*，实现了以下操作。

$init = L := [nil]$

$start = i := 0$

insert ＝ *L*:= *L*[0;…*i*]+[*x*]+*L*[*i*;…#*L*]

delete ＝ *L*:= *L*[0;…*i* ; *i*+1;…#*L*]

next ＝ *i*:= *i*+1

end ＝ *b*:= *i*=#*L*

value ＝ *x*:= *Li*

set ＝ *L*:= *i*→*x* | *L*

转换这些操作，提供一个堆实现。插入的节点来自一个空闲的表，删除的节点返回到空闲的表。

430 一个旧的实现者变量 *c*: -1, 0, 1 被新的实现者变量 *a, b*: *bin* 替换使得 *c*=-1 被 *a* 和 *b* 都为⊥替换，*c*=1 被 *a* 和 *b* 都为 T 替换，*c*=0 被 *a* 和 *b* 的值不相同替换。

(a) 转换式是什么？

(b) 用你的转换式转换 *c*:= 0 。

431 令 *b*: *bin* 为使用者变量，令 *n*: *nat* 为实现者变量，令运算为：

step ＝ **if** *n*>0 **then** *n*:= *n*-1 **else** *ok* **fi**

done ＝ *b*:= *n*=0

请说明没有转换式能摆脱 *n* ，使得

step 被转换为 *ok*

done 被转换为 *b*:= ⊥

尽管使用者不能发现其中的区别。

432 找到将练习 309(a)的程序转换为练习 309(b)的程序的数据转换式。

----------理论设计和实现结束

10.8 并发

433 创造一个相关组合的例子，其中有两种合理的方法来分割赋予组合不同意思的变量。提示：相关组合的操作不一定为程序。

434 令 *a*，*b* 和 *c* 为整数变量。简化

a:= *a*+*b*. (*b*:= *a*–*b* ‖ *a*:= *a*–*b*)

435 令 *x* 和 *y* 为自然数变量，不使用 ‖ ，重新表示下式：

(a) *x*:=*x*+1 ‖ **if** *x*=0 **then** *y*:=1 **else** *ok* **fi**

(b) **if** $x > 0$ **then** $y := x - 1$ **else** *ok* **fi** ‖ **if** $x = 0$ **then** $x := y + 1$ **else** *ok* **fi**

436 如果忽略时间，那么

x:=3. *y*:=4 ＝ *x*:=3 ‖ *y*:=4

如果忽略时间，某些相关组合可以并行执行。但是 *P*. *Q* 的执行时间是分别执行 *P* 和 *Q* 的时间和，这迫使它们按顺序执行。

$t{:=}t{+}1.\ t{:=}t{+}2 \ = \ t{:=}t{+}3$

类似地，如果忽略时间，某些独立组合可以顺序执行。但是 $P \parallel Q$ 的执行时间为分别执行 P 和 Q 的时间中的最大者，这迫使它们并行执行。

$t{:=}t{+}1 \ \parallel \ t{:=}t{+}2 \ = \ t{:=}t{+}2$

设计另一种组合形式，在相关组合和独立组合之间作个折中，使得它的执行除了必要的顺序执行以外，尽可能地并行执行。警告：这是一个研究问题。

437 (筛法) 给定变量 p : $[n*bin]{:=}[\bot;\bot;\ (n{-}2)*\mathsf{T}]$，以下程序是一个厄拉多塞筛法以判断一个数是否为质数。

for $i{:=}2;..ceil(n^{1/2})$
do if $p\ i$ **then for** $j{:=}i;..ceil(n/i)$ **do** $p{:=}(j{\times}i){\rightarrow}\bot \mid p$ **od**
else *ok* **fi od**

(a) 说明这个程序可被如何转换来体现出并发性。画出执行模式说明你的回答。

(b) 用一个 n 的函数表示它的具有最大并发性的执行时间，这个执行时间是什么？

438 练习 157 要求一个程序计算累加和（求总数）。写一个程序，这个程序可以在 $\log n$ 时间内将顺序执行转换为并行执行，其中 n 是表的长度。

439 (互斥组合) 独立组合 $P \| Q$ 要求 P 和 Q 没有共同的变量，尽管可以通过进行私有拷贝使用另一个进程的变量的初始值。一个可替换的方法，我们说的互斥组合，允许 P 和 Q 无限制地使用所有变量，然后选取变量 v 和 w 的互斥的集合，定义

$$P \mid v \mid w \mid Q = (P.v' = v) \wedge (Q.w' = w)$$

(a) 说明 $P \mid v \mid w \mid Q$ 如何执行。

(b) 证明如果 P 和 Q 是可实现的规范，则 $P \mid v \mid w \mid Q$ 也是可实现的。

440 将互斥组合 $P \mid v \mid w \mid Q$（练习 439）的定义从变量扩展到表的项上。

441 (半相关组合) 独立组合 $P \| Q$ 要求 P 和 Q 没有公共的变量，尽管可以通过进行私有拷贝使用另一个进程的变量的初始值。在本问题中我们开发另一种组合，称为半相关组合 $P \| Q$。与相关组合一样，它要求 P 和 Q 有相同的状态变量。与独立组合一样，它可以通过进程并行执行来执行，但每个进程只对状态变量的本地拷贝赋值。然后，当两个进程都完成后，变量的最终值是这样决定的：如果两个进程都没改变它的值，则其值不变；如果一个进程改变了它的值，另一个进程没有，那么它的值是改变后的值；如果两个进程都改变它的值，那么它的终结值是任意的。这个状态变量的最终重写不要求进程间的协调与通信；每个进程重写它所改变的变量。在两个进程都改变同一个变量的情况下，我们甚至不要求最终值是两个改变的值中的一个。重写可能将字位搞乱。

(a) 形式化定义半相关组合，包括时间。

(b) 哪些定律应用于半相关组合？

(c) 在什么情况下进程对状态变量进行私有拷贝是不必要的？

(d) 使用变量 x，y 和 z，不使用|||，表示

$$x := z \,|||\, y := z$$

(e) 使用变量 x，y 和 z，不使用|||，表示

$$x := y \,|||\, y := x$$

(f) 使用变量 x，y 和 z，不使用|||，表示

$$x := y \,|||\, x := z$$

(g) 使用变量 x，y 和 z，证明

$$x := y \,|||\, x := z \;=\; \textbf{if } x = y \textbf{ then } x := z \textbf{ else if } x = z \textbf{ then } x := y \textbf{ else } x := y \,|||\, x := z \textbf{ fi fi}$$

(h) 使用二元变量 x，y 和 z，不使用|||，表示

$$x := x \wedge z \,|||\, y := y \wedge \neg z \,|||\, x := x \wedge \neg z \,|||\, y := y \wedge z$$

(i) 令 $w : 0,..4$ 和 $z : 0,1$ 为变量。不使用|||，表示

$$w := 2 \times \max(div\ w\ 2)z + \max(\text{mod } w\ 2)(1 - z)$$
$$|||\quad w := 2 \times \max(div\ w\ 2)(1 - z) + \max(\text{mod } w\ 2)z$$

442 将半相关组合 $P|||Q$（练习 441）的定义从变量扩展到表的项上。

443 重新定义半相关组合 $P|||Q$（练习 441），使得如果 P 和 Q 同意变量其中一个改变后的值，则它以之为最终值，如果不同意，则它的终结值为

(a) 任意值。

(b) 两个改变后的值中之一。

444 我们想找到满足性质 p 的 $0,..n$ 中的最小数。线性查找解决了这个问题。但是对 p 的求值是昂贵的，假设其需要时间 1，其它不消耗时间。最快的解决方法是并发地在 n 个数上计算 p，然后找到满足性质 p 的最小数。不使用并发写一个程序，进行顺序向并行的转换以得到想要的计算。

445√ (哲学家就餐) 五个哲学家坐在圆桌前。在桌子中间有一碗无限量的面条。在每对相邻的哲学家之间有一只筷子。当一个哲学家饥饿时，他就去拿他左边和右边的筷子，因为吃面条需要一双筷子。如果相邻的哲学家在使用筷子，那么就至少有一只筷子不可用，那么饥饿的哲学家必须等待。当两只筷子都可用时，哲学家就可以吃一会儿面条，然后放下两只筷子，继续思考，直到再次饥饿。问题是写一个程序，其执行模拟哲学家的生活，要求最大的并发，并不导致死锁。

446 在一个具有数组元素赋值和表并发的语言中，什么是如下的精确前置条件

if $\#A \geq 1$ **then** *ok*
else if $A\ 0 > A\ (\#A - 1)$ **then** *ok*
else $A\ 0 := A\ (\#A - 1) \,\|\, A\ (\#A - 1) := A\ 0$ **fi fi**

使其可以精化

$$\forall i, j : 0,..\#A \cdot i \leq j \Rightarrow A'i \leq A'j$$

447 (同步等式) 已知字符串变量 X，它的 n 个项是有理数，以及一个有 n 个函数 f_i 的的字符串，每个函数都接收 n 个有理参数并产出一个有理数结果。向 X 赋一个值满足

$$\forall i{:}\ 0,..n \cdot X_i = f_i@X$$

或者，把它展开，

$$\begin{aligned} X_0 &= f_0\ X_0 X_1 \cdots X_{n-1} \\ X_1 &= f_1\ X_0 X_1 \cdots X_{n-1} \\ &\vdots \\ X_{n-1} &= f_{n-1}\ X_0 X_1 \cdots X_{n-1} \end{aligned}$$

也就是说，找到n个同步的固定点。假设一个对形式$X_i = f_i\ X_0 X_1 \cdots X_{n-1}$的赋值会导致一连串改进的估算值，直到$X$的值“足够接近”，在一定的偏差内。函数计算是整个计算中消耗时间的部分，所以尽可能地，函数计算应该平行进行。

----------并发结束

10.9 交互

448√ 假设 a 和 b 为整数边界变量，x 和 y 为整数交互变量，t 为广义自然数时间变量。假设每个赋值需要时间 1。不用任何程序设计记号（不用赋值，不用相关组合，不用独立组合），表述下面的式子。

$$(x := 2.x := x + y.x := x + y) \| (y := 3.y := x + y)$$

449√ (缓慢增长) 假设 *alloc* 分配一个内存空间，需要花费一个时间。然后下面的计算缓慢地分配内存。

$$GrowSlow \Leftarrow \textbf{if}\ t = 2 \times x\ \textbf{then}\ alloc \parallel x := t\ \textbf{else}\ t := t+1\ \textbf{fi}.\ GrowSlow$$

如果时间等于 $2 \times x$，那么分配一个空间，同时 x 成为分配时的时间戳，否则时钟继续嘀嗒。该进程永远重复。证明如果空间初始时小于时间的对数，且 x 是适当初始化的，那么在所有时间内空间都小于时间的对数。

450 不用任何编程记号（不用赋值，不用相关组合，不用独立组合）表达这段程序

$$(x := 1.\ x := x+y) \parallel (y := 2.\ y := x+y)$$

其中x和y是

(a) 边界变量和用时为0的赋值。

(b) 交互变量和用时为1的赋值。

451 令 a 和 b 为二元交互变量。定义

$$loop\ =\ \textbf{if}\ b\ \textbf{then}\ loop\ \textbf{else}\ ok\ \textbf{fi}$$

根据任何合理的度量加入时间变量，然后不使用 $\|$ 表示

$$b := \bot \| loop$$

作为一段相等的程序。

452√ (自动调温器) 定义一个燃气控制装置的自动调温器 (Thermostat) 。自动调温器与其它进

程并行操作

$$thermometer \| control \| thermostat \| burner$$

温度计和控制器一般位置在一起，但在逻辑上是区分开的。自动调温器的输入为：

- 实数 *temperature*，它来自温度计，表示实际温度。
- 实数 *desired* , 它来自控制器，表示想要的温度。
- 二元值 *flame*, 它来自于燃气装置内的一个火焰传感器，表示燃气是否在燃烧。

自动调温器的输出为：

- 二元值 *gas*, 如果开启燃气则对其赋值 T，如果关闭燃气则对其赋值 ⊥ 。
- 二元值 *spark*，对其赋值 T 会引发火花从而点燃燃气。

当实际温度比所希望的温度低于 ε 时需要加热，而如果实际温度比所希望的温度高 ε 时就不再需要加热， ε 足够小以至不被注意，又足够大以防迅速震荡。要加热时，使用火花于燃气至少需要一秒钟才能点燃燃气并使火焰成为稳定状态。而一个安全规程要求燃气不可以处于开启但非点燃状态超过 3 秒钟。另一个规程要求燃气装置在关闭之后，至少必须等待 20 秒钟再开启，以清除上次聚集而尚未烧完的燃气。最后，燃气装置必须对它的输入在 1 秒钟内作出响应。

453 置换定律对交互变量不起作用。

(a) 举一个该定律对交互变量失败的例子。

(b) 为交互变量开发一个新的置换定律。

454 根据交互变量赋值的定义，写变量需要一定的时间，而在这段时间内变量的值是未知的。但在赋值的一开始表达式中被赋值的变量就立即被读了。修改交互变量赋值的定义使得

(a) 在赋值结束时写立即发生。

(b) 对表达式中被赋值的变量的读也需要完全的赋值时间，正如写一样。

455 (交互数据转换) 7.2 节描述了边界变量的数据转换。当存在交互变量时我们应该如何进行数据转换？警告：这是一个研究类问题。

456 (电话) 详细说明一个简单的电话控制。其输入为那些可以执行的动作：拿起电话，拨号，和放下电话（挂电话）。其输出为一列数字（所拨的号）。如果间隔 5 秒还没有拨下一个号表示拨号结束。如果尚未等待 5 秒就挂电话，那么就没有输出。但是，挂电话后又在 2 秒内重新拿起电话，这表明刚才只是一个事故，不影响输出。

457 (一致) 有一些并行进程环状相连。每一个进程都有一个带有初始值的局部整数变量。这些初始值可能不相同，其他方面进程是相同的。所有进程的执行必须在与进程数呈线性的时间内终止，并且在终止时刻，所有的局部变量值都相同，且等于其中一个初始值。试写出这样的进程。

458 许多程序设计语言在输入时都需要一个变量，语法通常是 **read** *x*。形式化定义这种输

入形式。什么时候它比节 9.1 中描述的输入要更方便？什么时候不如后者方便？

459 写一个程序打印一个自然数序列，每个时间单元一个。

460 写一个程序重复地打印当前时间，直到某个给定的时间为止。

461 给定一个由不同字符按升序排列的有限串 S，写一个程序以下面的顺序打印串$*(S_{0..} \leftrightarrow S)$：短的串在长的串前面，相等长度的串以串（字母序，字典序）的顺序打印。

462 (T–串) 我们称串 S : [*(“a”, “b”, “c”)]为一个 T–串，如果它满足不存在相同的两个相邻非空段。

$$\neg\exists i, j, k \cdot 0 \le i < j < k \le \leftrightarrow S \wedge S_{i;..j} = S_{j;..k}$$

写一段程序，以字母次序输出所有的 T–串。(数学家苏证明了有无穷多个 T–串。)

463 根据节 5.5.0 给出的 **result** 表达式定义，当输出出现在编程数据的程序部分时会发生什么？可以读和使用输入吗？如果这样做会发生什么？

464 (重排) 写一段程序读、重排和写一个字符序列。输入包含可在任意位置出现的换行符；输出应包含只在每一个分号后出现的换行符。每当输入包含两个相连的星号，或是两个只由换行符分隔的星号，那么输出就应将这两个星号换成一个向上的箭头。除此以外，其他的输出应与输入相同。输入和输出都由一个特殊的符号作为结束标志。

465 (矩阵相乘) 写一段程序，使用 n^2 个进程，$2\times n^2$ 个局部信道，将两个 $n\times n$ 维矩阵相乘，执行时间要求为 n。

466 (输入实现) 令 W 表示 “等待信道 c 上的输入，然后读取它”。

(a) √ $W = t{:=}\ max\ t\ (T_r + 1).\ c?$

证明 $W \Leftarrow$ **if** $\sqrt{c}$ **then** $c?$ **else** $t{:=}\ t{+}1.\ W$ **fi** 假设时间是一个广义自然数。

(b) 现在令时间为一个广义非负实数，重新适当定义 W，并重新证明这个精化。

467 (时间耗尽的输入) 如练习 466，令 W 表示 “等待信道 c 上的输入，然后读取它”，区别是，如果输入在一个截止时间还不可行，则启动警报。

$$W \Leftarrow \textbf{if}\ t \le deadline\ \textbf{then if}\ \sqrt{c}\ \textbf{then}\ c?\ \textbf{else}\ t{:=}\ t{+}1.\ W\ \textbf{fi else}\ alarm\ \textbf{fi}$$

适当定义W，并证明精化式。

468 利用递归构造找出以下不动点等式的

$$twos = c!2.\ t{:=}t{+}1.\ twos$$

(a) 最弱不动点。

(b) 最强可实现不动点。

(c) 最强不动点。

469 下面是两个定义

$A =$ **if** $\surd c \wedge \surd d$ **then** $c? \vee d?$
else if $\surd c$ **then** $c?$
else if $\surd d$ **then** $d?$
else if $Tc_{rc} < Td_{rd}$ **then** $t := Tc_{rc} + 1.\ c?$
else if $Td_{rd} < Tc_{rc}$ **then** $t := Td_{rd} + 1.\ d?$
else $t := Tc_{rc} + 1.\ c? \vee d?$ **fi fi fi fi fi**

$B =$ **if** $\surd c \wedge \surd d$ **then** $c? \vee d?$
else if $\surd c$ **then** $c?$
else if $\surd d$ **then** $d?$
else $t := t+1.\ B$ **fi fi fi**

令时间为一个广义自然数，证明 $A=B$。

470 如下定义关系 $partmerge: nat \to nat \to bin$：

$partmerge\ 0\ 0$
$partmerge\ (m+1)\ 0 = partmerge\ m\ 0 \wedge Mc_{wc+m} = Ma_{ra+m}$
$partmerge\ 0\ (n+1) = partmerge\ 0\ n \wedge Mc_{wc+n} = Mb_{rb+n}$
$partmerge\ (m+1)\ (n+1) = partmerge\ m\ (n+1) \wedge Mc_{wc+m+n+1} = Ma_{ra+m}$
$\vee\ partmerge\ (m+1)\ n \wedge Mc_{wc+m+n+1} = Mb_{rb+n}$

现在 $partmerge\ m\ n$ 表示信道 c 上的头 $m+n$ 个输出是信道 a 上的 m 个输入和信道 b 上的 n 个输入的合并。定义 $merge$ 为

$merge = (a?.\ c!\ a) \vee (b?.\ c!\ b).\ merge$

证明 $merge = (\forall m \cdot \exists n \cdot partmerge\ m\ n) \vee (\forall n \cdot \exists m \cdot partmerge\ m\ n)$

471 (完美交替) 写一个规范，表示重复在信道 c 或 d 上读取一个输入的计算。规范表明计算可能从其中任意一个信道开始，然后交替进行。

472 (时间合并) 我们希望重复地从信道 c 或信道 d 上读取一个输入，谁的信息先到就读谁的，并将它写到信道 e 上。在每次读的时候，如果两个信道上的信息都已到达，那么任意读其中一个；如果只有一个信道上的信息已到达，就读那个到达的信息；如果两个信道上的信息都未到达，则等待第一个到达者并读取它(若两者同时到达，读取任意一个)。

(a)√ 形式化写出这个规范，然后写一个它的程序。

(b) 证明

$Te_{we} = max\ t\ (min\ (Tc_{rc})\ (Td_{rd})+1)$
$\forall m, n \cdot Te_{we+m+n+1} \le max\ (max\ (Tc_{rc+m})\ (Td_{rd+n}))\ (Te_{we+m+n})+1$

473 (公平时间合并) 本问题与时间合并（练习 472）一样，但如果当两个信道上都有输入时，此次读取的选择应与前一次读取的选择相反。另外，如果在等待输入到达之后，两个

信道上的输入是同时到达的，读取的选择也应与前一次读取的选择相反。

474 令 t 为广义自然数时间变量。以下的规范是否可实现？

(a) $\forall n: nat \cdot M_n=n \wedge T_n=t$

(b) $\forall n: nat \cdot M_{w+n}=n-t \wedge T_{w+n}=t-n$

(c) $\forall n: nat \cdot M_{r+n}=n \wedge T_{r+n}=t$

(d) $M_w=t-1 \wedge T_w=t-1$

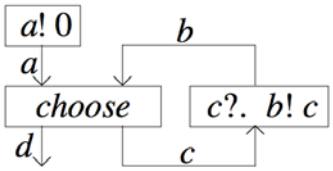
475 在节 9.1.6 的反应控制器中，同步装置接收从数字化装置传来的数字数据要比接收从控制装置传来的请求要快。现在假设控制器有时比数字化装置运行快。修改这个同步装置使得：如果有两个或两个以上的请求连续排队到达(在新的数字数据到达之前)，同样的数字数据将被送去答复每一个请求。

476√ 证明以下程序的执行会导致死锁。

(a) **chan** $c: int \cdot c?.\ c!\ 5$

(b) **chan** $c, d: int \cdot (c?.\ d!\ 6\ ) \parallel (d?.\ c!\ 7\ )$

477 (布诺克–阿克曼)下图显示了一个通信进程网络。



该网络的形式化描述为

chan $a, b, c \cdot a!\ 0 \parallel choose \parallel (c?.\ b!\ c)$

形式化定义 *choose*，加入传递时间，并描述输出信息和时间，如果

(a) *choose* 或者从 a 读取数据然后在 c 和 d 上输出一个 0，或者从 b 上读取数据然后在 c 和 d 上输出一个 1。两者可以自由选择。

(b) 如同(a)中，*choose* 或者从 a 读取数据然后在 c 和 d 上输出 0，或者从 b 上读取数据然后在 c 和 d 上输出 1。但这次选择不再是任意地，*choose* 选择输入先到达的信道读取信息(若两者同时到达，任意选择其一)。

478√ (幂级数相乘) 写一段程序，从信道 a 读取一个幂级数 $a_0+a_1\times x+a_2\times x^2+a_3\times x^3$...的无穷系数序列 a_0, a_1, a_2, a_3 ..., 并且并行地从信道 b 读取一个幂级数 $b_0+b_1\times x+b_2\times x^2+b_3\times x^3$... 的无穷系数序列 b_0, b_1, b_2, b_3 ..., 并且并行地将它们二者的积的幂级数 $c_0+c_1\times x+c_2\times x^2+c_3\times x^3$... 的无穷系数序列 c_0, c_1, c_2, c_3 ...写到信道 c 上。假设所有输入都已到达，无任何输入延迟。要求每一个时间单位生成一个输出。

479 (文件更新) 要读一个由记录组成的主文件和一个由记录组成的交易文件，一次读一个记录，另外再写一个由记录组成的新文件，一次写一个记录。一个记录由两个正文段

组成：一个“关键字”段和一个“信息”段。主文件按关键字排序，没有重复的关键字，并且最后一个记录有一个哨符关键字 “zzzz”，保证它大于所有其他的关键字。交易文件也按照其关键字排序，也具有一样的结束哨符关键字，但是它允许出现重复关键字。新文件与主文件相似，但根据交易文件的内容有所改变。如果交易文件包含一个关键字在主文件中没有出现过的记录，那么这个记录就应被加入到新文件中。如果交易文件包含了一个关键字在主文件中出现过的记录，若其 “信息”字段不为空，表示该记录是对原记录的相应信息字段的更新，若其 “信息”字段为空，表示删除该记录。每当交易文件包含重复的关键字时，具有该关键字的最后的那个记录决定结果。

480 (重复) 写一段程序读取一个无穷序列，并且在每次偶数数量的输入之后，输出一个二元值，表示迄今为止输入的后一半是否为其前一半的重复。

481 (互斥) 进程 *P* 是一个 “非临界段”*PN* 和一个 “临界段”*PC* 的无休止循环，进程 *Q* 与它类似。

$$P = PN.\ PC.\ P$$

$$Q = QN.\ QC.\ Q$$

它们并行执行$(P \parallel Q)$。形式化说明这两个临界段永远不会在同一时刻执行。

(a) 通过插入已赋值的但永不会被用的变量。

(b) 通过插入永远不会被读取的信道输出。

482 (同步通信) 当发信者准备开始发信，同时收信者准备开始接收时，通信会发生。已准备好的一方必须等待其它方也准备好后才可通信。

(a) 设计一个同步通信理论。对每一个信道而言，只需要一个游标，但需要两个(或更多)的时间脚本。输出与输入一样，不断增加时间直到当前信息时间脚本的最大值。

(b) 用一些例子说明它是如何工作的，包括死锁例子。

(c) 举一个例子说明当采用异步通信时，该例不发生死锁，而当采用同步通信时，该例发生死锁。

483 节 5.3 中定义并实现了程序 **wait until** *w* 其中 *w* 是一个时间。请定义并实现程序 **wait until** *c* 其中 *c* 是一个条件。例如，**wait until** *x*=*y* 应当延迟执行直到变量 *x* 和 *y* 相等。条件中的至少一个变量应当是属于另一个进程的交互变量。

----------交互结束

----------练习结束

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

每当在标准的记号和新的完美的记号之间进行选择，我会选择标准的记号。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, 所以最好引入一个像↑的对称符号作为中缀操作符：*x*↑*y*。我个人私底下总是这样做, 但在本书中我所选择的符号尽量保持数量少和符合合理的传统。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。在第一版中，我用λ记号表示函数，认为它是标准的。十年中学生们说服了我它不是标准的，于是在后面的版本中我用了更合适的记号表示函数。

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。最后我们做一个选择并一直使用它。额外的括号总是可以使用, 特别是在优先级结构不清晰时更应当使用。为了结构清晰，最好给 ∧ 和 ∨ 相同的优先级，但我还是保持了传统。本书采用的优先级比我预想的要多。我可以用单运算符–代替¬、×代替∧、双运算符+代替∨、并且=和≠代替⇒和⇐。虽然节约了四个优先级, 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 = ⇐ ⇒ 是一个创新；我希望它既容易阅读理解也容易书写。请用过一阵后再对此进行评判, 它毕竟节省了许多括号。人们可以很快看出来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 基本理论

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，逻辑的主题从它混淆的哲学的过去中显现出来。我认为命题就是布尔（二元）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、函数一样的等号来应用于布尔表达式。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证明它们。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 1 章所述，否定操作符和一致性规则的引入使我们不需要这两个新词。我们可以用 ¬*expression* 是定理来代替说明 *expression* 是反定理。另外有何必要引入⊥？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来引入证明规则（和公理）。本书中没有形式化的元语言，元语言就是自然语言。形式化的元语言有助于（尽管不必要）提出理论并与其它竞争的各种形式体系作比较，同时也有助于证明形式体系的定理。但在本书中，仅提出了一种形式体系。如果为了引入这种形式体系而先去学习另一种形式体系，增加这种负担是没有必要的。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：

$\langle v \to b \rangle\, a = b[\, a/v\,]$

但接着就必须说明 $b[\, a/v\,]$表示“将 b 中的 v 置换成 a ”。因此可以直接说：

$\langle v \to b \rangle\, a$ = (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“公理模式”，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。另外我还把“定律”作为“定理”的同义语（我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，但目前尚未设计出一个合适的理论”。

在选择某些公理和定律的名称时我有点随性。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 基本数据结构

为什么要引入束? 集合不也一样使用吗? 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：$A,B = B,A$ （逗号是对称的或可交换的）。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创造束, 只是称它们为集合的“内容”。请注意, 上述语法恰恰等同于束；字符串连接（用并置表示）分布作用于它们操作元的所有元素, 而交替（用竖直短线表示）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。束论向我们介绍了聚合, 而集合论向我们介绍了打包。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（正如多少年来一直如此）, 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）, 并且我愿意保留它。同样, 束也是一个优美的数据结构, 也引入了一个概念（聚合）, 我也愿意保留它。我总是倾向于使用足够达到目的的最简单的结构。

函数程序设计的主题不能方便地表达非确定性。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。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。而所需要的正是束。一个束总是可以被看作是一个“非确定值”。这个问题在程序设计之前已经存在于数学计算里。例如，有两个平方根，于是 $1^{1/2} = 1,-1$ 。并且$(1^{1/2})^2 = (1,-1)^2 = 1^2,(-1)^2 = 1,1 = 1$ 。

类型论复制了它的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。通过使用束作为类型, 这样的重复就被消除了。

很多数学家认为弧形括号和逗号只是语法符号。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, 我们看到弧形括号有逆运算）。这是一个由来已久的历史趋势。例如, = 最初只是一个语法符号, 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形式化运算符。

在许多文章中, 作者会对有时把表连接记号错用作表和项的连接而道歉。或者也许有三种连接记号：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。为了排序，可怜的作者不得不与表所提供的包装符号作斗争。我向这些作者提供小建议：不用包装进行排序。（当然, 它们可以在需要时包装到表中。我并不是要去除表。）

令 x 为实数，p 为正实数。于是 $0 < 0;x < p$ 。因此，$0;x$ 是一个极小量，比 0 大但是比任何正实数都要小。并且 $0 < 0;0;x < 0;p < p$ ，于是 $0;0;x$ 是一个比 $0;p$ 小的极小量。依此类推。类似地，$x < \infty < \infty;x < \infty;\infty < \infty;\infty;x$ ，依此类推。但本书不讨论极小量和无穷量。

----------基本数据结构结束

11.0.3 函数理论

我使用了词语“局部的”和“非局部的”，而其它人可能使用的是词语“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⟨ ⟩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, $\langle x{:}\ int \rightarrow x+y \rangle$ 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 形成的量词仍然称为 *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”。

我忽略了极限的“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，但它仍可能告诉我们一些有用的东西。

----------函数理论结束

11.0.4 程序理论

赋值语句可能定义为

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

其中 *defined* 排除了像 1/0 这样的表达式，而 T 是变量 x 的类型。我不考虑 *defined* 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 为自然数变量时 $n := n - 1$ 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。更糟糕的是，我们会失去替换定律；我们希望(n:= -1. n≥0)为⊥ 。

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, 所以我想句号会是一个可接受的符号。我考虑过交换这两个符号, 用分号表示相关组合, 而用句号表示字符串连接, 但是后者不可行。

在自然语言中, 词语“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“必要且充分的前置条件”, 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：

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

我们应该知道在这个规范中 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 X 和 S 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涉及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, 其中同样的规范为

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

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 增加的规范是

$$x' > x$$

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, 最好能使它尽可能简单。

有人也许会推测任何类型的数学表达式都可以用作规范：无论什么都可行。某事物的规范, 不管是汽车还是计算, 都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困难。

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：无论怎么对应都可行。在 Z 中, 表达式 T 用来刻画（描述）会终止的计算, 而⊥用来刻画（描述）不会终止的计算。理由类似于：⊥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 个时间单位, 观察到它满足 $t' > t+10$, 而不满足 $t' \leq t+10$。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 $t' \leq t+10$ 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, 即使是不会终止的计算也不例外。因为 T 对任何事物都为真, 所以它描述了一切事物, 即使是不会终止的计算也不例外。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并非随心所欲的。

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任何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）时需要在无限循环内进行辨别。

----------程序理论结束

11.0.5 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，那么

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

从实现简单和执行快速角度来看，这种方法比初试化成一个“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 309 （多数表决））。然而，如果我们不证明所有的精化，那么用 *undefined* 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 fi**）应用于未定义的值 *undefined*，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 *undefined* 的任何等式。有些程序设计语言为了消除由于使用未初始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，又只消除了错误检测而不是错误本身。

另一种定义变量说明的方法是

$$\textbf{var}\ x{:}\ T \ =\ x'{:}\ T \wedge ok$$

此式是 x 的范围的开始，并且

$$\textbf{end}\ x \ =\ ok$$

此式结束了 x 的范围。在每一个这些程序中，ok 维持了其他变量。这种声明不要求范围嵌套；它们可以重叠。

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：

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

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

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

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

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

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

和

$$s + \sum L\ [n;..\#L] = \sum L \wedge n{\neq}\#L \Rightarrow\ s' + \sum L\ [n';..\#L] = \sum L \wedge 0 \le \#L - n' {<} \#L - n$$

$$\Leftarrow \qquad s{:=}\ s + Ln.\ n{:=}n{+}1$$

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）。有时不变式和变式方法需要引入额外的第五章的证明方法不需要的常量（数学类变量）。例如，把表 L 中每项加 1 需要引入一个新的表常量来表示 L 的初始值。

节5.2.0中表示$W \Leftarrow \textbf{while}\ b\ \textbf{do}\ P\ \textbf{od}$是另一种记号，但它有很危险的误导性。看起来$W$被一个只涉及$b$和$P$的程序精化；实际上，$W$被一个涉及$b$，$P$，和$W$的程序精化。

如果所有实数为概率，而不仅是 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 $\top = \infty$ 和 $\bot = -\infty$ 。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表达的概率分布。如果 x 是类型 T 的变量，它变成一个类型 $T \rightarrow prob$ 的变量使得 $\sum x = \sum x' = 1$。然后所有的运算符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，而不是通过它们的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“缩小”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 / , 其左操作元为一个双目操作符，其右操作元为一张表。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。本书的目的就是要教授必要的程序设计技巧。

正如我们的例子所阐明的那样，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而非递归精化，使得证明复杂化；而函数式程序员坚持使用等式而非精化，这使得非确定性问题的解决更为困难。

----------程序设计语言结束

11.0.6 递归定义

递归结构总是可以通过取一个近似值序列的极限完成。我的创新是用 ∞ 代替序列的索引；这比寻找极限要容易得多。替换 ∞ 并不总能保证产生一个理想的不动点，但寻找极限也是一样。替换 ∞ 在除了设法求极限的例子以外效果还是很好的。

----------递归定义结束

11.0.7 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”。

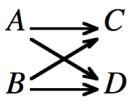
我们使用了一个小心构造的例子，而不是一个在实际中会发生的例子，说明了数据转换的不完备性。我更倾向于采用简单的规则，这些规则对任何会真正发生的问题的转换而言是充分的，不仅是为说明理论上的不完备性的问题，而不倾向于切换到一个具有完备性的更复杂的规则或规则的组合。为了重新获得完备性，我们需要的只是引入局部变量的正常数学实践。这种目的的变量被不同的作者称为“边界变量”，“逻辑常量”，“规范变量”，“灵魂变量”，“抽象变量”和“预言变量”。

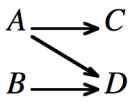
----------理论设计和实现结束

11.0.8 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非常通用的方法：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十分勉强。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。

这里有两个执行模式。





正如我们在第八章看到的，第一个执行模式可以被表示为 $((A \| B).(C \| D))$ 而不需任何同步原语。但第二个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 441）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更简单的定义（独立组合）。

----------并发结束

11.0.9 交互

如果 x 是一个交互变量，($t'=\infty$. $x:=2$. $x:=3$)不幸地为⊥；那么交互变量理论有一点点太强了。类似地，($wc'=\infty$. $c!2$. $c!3$) 不幸地为⊥；那么通信理论有一点点太强了。为了除掉这些不理想的结果，不得不弱化对有前件 $t'<\infty$的交互变量的赋值的定义，并弱化有前件 $wc'<\infty$的输出。但我认为这些莫名其妙的情况不值得把理论复杂化。

在可实现性公式中，不存在合取式 $r' \le w'$ 以保证读游标不能向前超过写游标。而在 9.1.8 死锁这一节中，我们看到这种情况确实会发生。当然它要花费无穷时间才会发生。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：

$$\begin{aligned} & \textbf{chan}\ c \bullet\ t := max\ t\ (T_c + 1).\ c? \\ = \quad & \exists M, T, r, r', w, w' \bullet\ t' = max\ t\ (T_0 + 1) \wedge r' = 1 \wedge w' = 0 \\ = \quad & t' \ge t \end{aligned}$$

我们希望证明 $t' = \infty$。为得到这个答案，必须强化对局部信道声明的定义，通过加入合取式 $T_{w'} \ge t'$。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，尽管他们带来的的影响可能很大。我可能没有感谢那些在我失意的日子里给我出谋划策的人，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，或许也有兴致所至，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 vonWright 所著的《精化演算》（*Refinement Calculus*）

一书(1998)。

同一时期，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)。很快就又发现同种描述方式，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，这一点在一开始就应该很显然了。这些结果发表在一系列论文中(1984, 1986, 1988, 1989, 1990, 1994, 1998, 1999, 2011), 最终导致本书的面世。

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 建议用$\mathcal{C}$和$\$$分别表示束和集合的基数。Robert Will 和 Lev Naiman 说服我添加结尾符 **fi** 和 **od** 。“conflation”一词的使用是由 Doug McIlroy 提议的。练习 29（括号代数）来自 Philip Meguire，他是从 George Spencer-Brown 那里得来的，而他是从 Charles Sanders Peirce 处得来的。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”一词及其用法来自于 Lambert Meertens(1986)。Peter Kanareitsev 对高阶函数方面进行了帮助。Alan Rosenthal 建议我不要担心极限的 “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，尽管在自然语言中还存在一些。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 的工作中得到启发(1987); 在他们的语言 LUSTRE 中，循环的每一次执行占用 1 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, 他在访问圭尔夫大学时学会这个解。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改编自 Greg Nelson 的关于实现天使般的非确定性的一个技巧(1989)。Lev Naiman 发现了早期版本（现在的练习 313(a)）中关于 **result** 公理的不一致。Carroll Morgan 和 Annabelle McIver 建议将概率看作可观察的量词(1996)，练习 325（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 合作完成的(1992)。Theo 还将时间加入 **while** 循环的递归定义中(1997)。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)。程序-树理论经历了几个连续的版本归功于 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 Theo Norvell 和我检查过它们与 Wei Chen 和 Jan Tijmen Udding(1989)的公式的等价性。Theo 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 的资源分配例子(1990)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 原创的(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）组合的各种各样的公式(1981, 1984, 1990, 1994)；第一版中的应归功于 Theo Norvell，在本版本中作为练习 441（半-相关组合）出现，并在 Hoare 和 He 的工作中使用到(1998)；在本版本中 Leslie Lamport 说服我回到早先的(1984, 1990)版本：简单合取。8.1 顺序到并行转换一节是与 Chris Lengauer 合作完成的(1981); 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。输入检测是 Alain Martin 的一个发明(1985)，他把它称为“试验值（probe）”。监控器（Monitors）由 Per Brinch Hansen(1973)和 Tony Hoare (1974)创造。幂级数相乘来自 Doug McIlroy(1990)，他把其归功于 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 进一步发展作为埃因霍温理工大学的考试题；它们自此出现在 Edsger Dijkstra 和 Wim Feijen 合写的一本书中(1988)。有些练习来自 Martin Rem 所写的一系列杂志文章(1983,..1991)。其他的练习来源十分广泛，这里无法一一提及。这本书里的理论被应用到 Yannis Kassios 的博士论文（2006）中的面向对象程序设计，以及 Albert Lai 的博士论文（2013）中的延迟执行理论；延迟执行理论也出现在我的论文里（2018）。Albert 的理论也验证了本书中理论的可靠性。

----------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.vonWright: *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

P.BrinchHansen: “Concurrent Programming Concepts”, *ACM Computing Surveys*, volume 5, pages 223,..246, 1973 December

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, 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*, volume 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: “T ermination Conventions and Comparative Semantics”, *Acta Informatica*, volume 25, number 1, pages 1,..15, 1988 January

E.C.R.Hehner: “Termination is Timing”, 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: “Abstractions of Time”, *a Classical Mind*, chapter 12, Prentice-Hall, 1994

E.C.R.Hehner: “Formalization of Time and Space”, *Formal Aspects of Computing,* volume 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: “a Probability Perspective”, *Formal Aspects of Computing* volume 23, number 4, pages 391,..420, 2011

E.C.R.Hehner: “a Theory of Lazy Imperative Timing”, Workshop on Refinement, Oxford U.K., 2018 July

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.Sørensen, 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, 1990

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

I.T.Kassios: *a Theory of Object-Oriented Refinement*, PhD thesis, University of Toronto, 2006

A.Y.C.Lai: *Eager, Lazy, and Other Executions for Predicative Programming*, PhD thesis, University of Toronto, 2013

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

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

11 Reference 222 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

J.M.Morris, A.Bunkenburg: “a Theory of Bunches”, *Acta Informatica*, volume 37, number 8, pages 541,..563, 2001 May

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*, Chapman-Hall, 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.Sørensen, H.Rischel: “Control Program for a Gas Burner”, Technical University of Denmark, Department of Computer Scence, 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, Cambridge University Press, 1998

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

K.Seidel, C.Morgan, A.K.McIver: “an Introduction to Probabilistic Predicate Transformers”, technical report PRG-TR-6-96, Oxford University, 1996

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

(此中文翻译版忽略了索引一节)

11.3 词语对照
（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
二元表达式（布尔表达式）　binary expression
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（文本）　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
长文本　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）
文本（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
归谬法 Reductio ad Absurdum
归并 conflation

（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
因式分解 factoring
并置 juxtaposition

（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
否定后件律 Modus Tollens

（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（范围） 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 parameter
连接 catenation
析取因子 disjunct
析取式 disjunction
析取三段论 Disjunctive Syllogism
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（操作元） operand
运算符（操作符） 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
肯定前件论式 Modus Ponens

（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

----------词语对照结束

11.4 公理和定律

11.4.0 二元 (布尔)

令 a , b , c , d 和 e 为二元值。

镜面定律

$$a \Leftarrow b = b \Rightarrow a$$

布尔定律

$$\top$$

$$\neg \bot$$

双重否定定律

$$\neg \neg a = a$$

排中定律 (Tertium non Datur)

$$a \vee \neg a$$

对偶定律 (德・摩根定律)

$$\neg (a \wedge b) = \neg a \vee \neg b$$

$$\neg (a \vee b) = \neg a \wedge \neg b$$

非矛盾定律

$$\neg (a \wedge \neg a)$$

互斥定律

$$a \Rightarrow \neg b = b \Rightarrow \neg a$$

$$a = \neg b = b = \neg a$$

基定律

$$\neg (a \wedge \bot)$$

$$a \vee \top$$

$$a \Rightarrow \top$$

$$\bot \Rightarrow a$$

包含定律

$$a \Rightarrow b = \neg a \vee b$$ (实质蕴涵)

$$a \Rightarrow b = (a \wedge b = a)$$

$$a \Rightarrow b = (a \vee b = b)$$

同等定律

$$\top \wedge a = a$$

$$\bot \vee a = a$$

$$\top \Rightarrow a = a$$

$$\top = a = a$$

吸收定律

$$a \wedge (a \vee b) = a$$

$$a \vee (a \wedge b) = a$$

幂等定律

$$a \wedge a = a$$

$$a \vee a = a$$

直接证明定律

$(a \Rightarrow b) \wedge a \Rightarrow b$ (肯定前件论式)

$(a \Rightarrow b) \wedge \neg b \Rightarrow \neg a$ (否定后件律)

$(a \vee b) \wedge \neg a \Rightarrow b$ (析取三段论)

自反定律

$a \Rightarrow a$

$a = a$

传递定律

$(a \wedge b) \wedge (b \wedge c) \Rightarrow (a \wedge c)$

$(a \Rightarrow b) \wedge (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$

$(a = b) \wedge (b = c) \Rightarrow (a = c)$

$(a \Rightarrow b) \wedge (b = c) \Rightarrow (a \Rightarrow c)$

$(a = b) \wedge (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$

间接证明定律

$\neg a \Rightarrow \bot = a$ (归谬法)

$\neg a \Rightarrow a = a$

特定化定律

$a \wedge b \Rightarrow a$

结合定律

$a \wedge (b \wedge c) = (a \wedge b) \wedge c$

$a \vee (b \vee c) = (a \vee b) \vee c$

$a = (b = c) = (a = b) = c$

$a \neq (b \neq c) = (a \neq b) \neq c$

$a = (b \neq c) = (a = b) \neq c$

分配定律 (因式分解)

$a \wedge (b \wedge c) = (a \wedge b) \wedge (a \wedge c)$

$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$

$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$

$a \vee (b \vee c) = (a \vee b) \vee (a \vee c)$

$a \vee (b \Rightarrow c) = (a \vee b) \Rightarrow (a \vee c)$

$a \vee (b = c) = (a \vee b) = (a \vee c)$

$a \Rightarrow (b \wedge c) = (a \Rightarrow b) \wedge (a \Rightarrow c)$

$a \Rightarrow (b \vee c) = (a \Rightarrow b) \vee (a \Rightarrow c)$

$a \Rightarrow (b \Rightarrow c) = (a \Rightarrow b) \Rightarrow (a \Rightarrow c)$

$a \Rightarrow (b = c) = (a \Rightarrow b) = (a \Rightarrow c)$

对称定律 (可交换定律)

$a \wedge b = b \wedge a$

$a \vee b = b \vee a$

$a = b = b = a$

$a \neq b = b \neq a$

普遍化定律

$a \Rightarrow a \vee b$

反分配定律

$a \wedge b \Rightarrow c = (a \Rightarrow c) \vee (b \Rightarrow c)$

$a \vee b \Rightarrow c = (a \Rightarrow c) \wedge (b \Rightarrow c)$

反对称定律 (双重蕴涵)

$(a \Rightarrow b) \wedge (b \Rightarrow a) = a = b$

移动定律

$a \wedge b \Rightarrow c = a \Rightarrow (b \Rightarrow c)$

$a \wedge b \Rightarrow c = a \Rightarrow \neg b \vee c$

抛弃定律

$a \wedge (a \Rightarrow b) = a \wedge b$

$a \Rightarrow (a \wedge b) = a \Rightarrow b$

归并定律

$(a \Rightarrow b) \wedge (c \Rightarrow d) \Longrightarrow a \wedge c \Rightarrow b \wedge d$

$(a \Rightarrow b) \wedge (c \Rightarrow d) \Longrightarrow a \vee c \Rightarrow b \vee d$

反单调定律

$a \Rightarrow b \Longrightarrow (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$

单调定律

$a \Rightarrow b \Longrightarrow c \wedge a \Rightarrow c \wedge b$

$a \Rightarrow b \Longrightarrow c \vee a \Rightarrow c \vee b$

$a \Rightarrow b \Longrightarrow (c \Rightarrow a) \Rightarrow (c \Rightarrow b)$

换位定律

$a \Rightarrow b = \neg b \Rightarrow \neg a$

相等和不等定律

$a=b = (a \wedge b) \vee (\neg a \wedge \neg b)$

$a \neq b = (a \wedge \neg b) \vee (\neg a \wedge b)$

分解定律

$a \wedge c \Longrightarrow (a \vee b) \wedge (\neg b \vee c) = (a \wedge \neg b) \vee (b \wedge c) \Longrightarrow a \vee c$

情况创建定律

$a =$ **if** b **then** $b \Rightarrow a$ **else** $\neg b \Rightarrow a$ **fi**

$a =$ **if** b **then** $b \wedge a$ **else** $\neg b \wedge a$ **fi**

$a =$ **if** b **then** $b = a$ **else** $b \neq a$ **fi**

情况分析定律

if a **then** b **else** c **fi** $= (a \wedge b) \vee (\neg a \wedge c)$

if a **then** b **else** c **fi** $= (a \Rightarrow b) \wedge (\neg a \Rightarrow c)$

单一情况定律

if a **then** $\top$ **else** b **fi** $= a \vee b$

if a **then** $\bot$ **else** b **fi** $= \neg a \wedge b$

if a **then** b **else** $\top$ **fi** $= a \Rightarrow b$

if a **then** b **else** $\bot$ **fi** $= a \wedge b$

情况吸收定律

if a **then** b **else** c **fi** $=$ **if** a **then** $a \wedge b$ **else** c **fi**

if a **then** b **else** c **fi** $=$ **if** a **then** $a \Rightarrow b$ **else** c **fi**

if a **then** b **else** c **fi** $=$ **if** a **then** $a = b$ **else** c **fi**

if a **then** b **else** c **fi** $=$ **if** a **then** b **else** $\neg a \wedge c$ **fi**

if a **then** b **else** c **fi** $=$ **if** a **then** b **else** $a \vee c$ **fi**

if *a* **then** *b* **else** *c* **fi** ＝ **if** *a* **then** *b* **else** *a* ≠ *c* **fi**

情况分配定律 (情况因式分解)

¬ **if** *a* **then** *b* **else** *c* **fi** ＝ **if** *a* **then** ¬*b* **else** ¬*c* **fi**

if *a* **then** *b* **else** *c* **fi** ∧ *d* ＝ **if** *a* **then** *b* ∧ *d* **else** *c* ∧ *d* **fi**

并且类似地可替换 ∧为∨ = ≠ ⇒ ⇐中任意一个

if *a* **then** *b* ∧ *c* **else** *d* ∧ *e* **fi** ＝ **if** *a* **then** *b* **else** *d* **fi** ∧ **if** *a* **then** *c* **else** *e* **fi**

并且类似地可替换 ∧为∨ = ≠ ⇒ ⇐中任意一个

----------二元结束

11.4.1 通用符号

操作符 = + **if then else fi** 适用于任何类型的表达式（但 **if then else fi** 的第一个操作元必须为二元值），并有如下定律：

$x = x$	自反性
$x=y \ ＝ \ y=x$	对称性
$x=y \wedge y=z \Rightarrow x=z$	传递性
$x=y \Rightarrow fx = fy$	透明性
$x \neq y \ ＝ \ \neg(x = y)$	不等性
if ⊤ **then** *x* **else** *y* **fi** = *x*	基情况
if ⊥ **then** *x* **else** *y* **fi** = *y*	基情况
if *a* **then** *x* **else** *x* **fi** = *x*	幂等情况
if *a* **then** *x* **else** *y* **fi** = **if** ¬*a* **then** *y* **else** *x* **fi**	逆转情况

操作符 < ≤ > ≥ 可适用于数字, 字符, 字符串和表, 有如下定律：

$\neg x < x$	反自反性
$\neg(x < y \wedge x > y)$	排他性
$\neg(x < y \wedge x = y)$	排他性
$x \leq y \wedge y \leq x \ ＝ \ x = y$	反对称性
$x<y \wedge y<z \Rightarrow x<z$	传递性
$x \leq y \ ＝ \ x<y \vee x=y$	包含性
$x > y \ ＝ \ y < x$	镜像
$x \geq y \ ＝ \ y \leq x$	镜像
$x<y \vee x=y \vee x>y$	完全性，三分法

----------通用符号结束

11.4.2 数

令 *d* 为一个数字序列（零个或多个数字），令 *x* , *y* , 和 *z* 为数。

*d*0+1 = *d*1	计数
*d*1+1 = *d*2	计数
*d*2+1 = *d*3	计数
*d*3+1 = *d*4	计数
*d*4+1 = *d*5	计数
*d*5+1 = *d*6	计数

$d6+1 = d7$	计数
$d7+1 = d8$	计数
$d8+1 = d9$	计数
$d9+1 = (d+1)0$	计数 (参见练习 32)
$x+0 = x$	恒等性
$x+y = y+x$	对称性
$x+(y+z) = (x+y)+z$	结合性
$-\infty < x < \infty \Rightarrow (x+y = x+z = y=z)$	消除性
$-\infty < x \Rightarrow \infty + x = \infty$	吸收性
$x < \infty \Rightarrow -\infty + x = -\infty$	吸收性
$-x = 0 - x$	否定
$--x = x$	自反
$-(x+y) = -x + -y$	分配性
$-(x \times y) = -x \times y$	半分配性
$-(x/y) = -x / y$	半分配性
$x-0 = x$	恒等性
$x-y = x + -y$	减法
$x + (y - z) = (x + y) - z$	结合性
$-\infty < x < \infty \Rightarrow (x - y = x - z = y = z)$	消除性
$-\infty < x < \infty \Rightarrow x - x = 0$	逆转
$x < \infty \Rightarrow \infty - x = \infty$	吸收性
$-\infty < x \Rightarrow -\infty - x = -\infty$	吸收性
$-\infty < x < \infty \Rightarrow x \times 0 = 0$	基
$x \times 1 = x$	恒等性
$x \times y = y \times x$	对称性
$x \times (y+z) = x \times y + x \times z$	分配性
$x \times (y \times z) = (x \times y) \times z$	结合性
$-\infty < x < \infty \wedge x \neq 0 \Rightarrow (x \times y = x \times z = y=z)$	消除性
$0 < x \Rightarrow x \times \infty = \infty$	吸收性
$0 < x \Rightarrow x \times -\infty = -\infty$	吸收性
$x / 1 = x$	恒等性
$-\infty < x < \infty \wedge x \neq 0 \Rightarrow x / x = 1$	逆转
$x \times (y / z) = (x \times y) / z = x / (z / y)$	乘法-除法
$y \neq 0 \Rightarrow (x / y) / z = x / (y \times z)$	乘法-除法
$-\infty < x < \infty \Rightarrow x / \infty = 0 = x / -\infty$	湮没性
$-\infty < x < \infty \Rightarrow x^0 = 1$	基
$x^1 = x$	恒等性
$x^{y+z} = x^y \times x^z$	指数
$x^{y \times z} = (x^y)^z$	指数
$-\infty < 0 < 1 < \infty$	方向性
$x < y = -y < -x$	反射性
$-\infty < x < \infty \Rightarrow (x+y < x+z = y<z)$	消除性，转换
$0 < x < \infty \Rightarrow (x \times y < x \times z = y<z)$	消除性，比例
$x < y \vee x = y \vee x > y$	三分法

$-\infty \le x \le \infty$ 极值

---数结束

11.4.3 束

令 x 和 y 为元素（二元值，数，字符，集合，字符串和元素的表）。

$x: y = x=y$	初等公理
$x: A,B = x: A \vee x: B$	复合公理
$A,A = A$	幂等性
$A,B = B,A$	对称性
$A,(B,C) = (A,B),C$	结合性
$A'A = A$	幂等性
$A'B = B'A$	对称性
$A'(B'C) = (A'B)'C$	结合性
$A,B: C = A: C \wedge B: C$	反分配性
$A: B'C = A: B \wedge A: C$	分配性
$A: A,B$	普遍化
$A'B: A$	特定化
$A: A$	自反性
$A: B \wedge B: A = A=B$	反对称性
$A: B \wedge B: C \Rightarrow A: C$	传递性
$\not\subset null = 0$	大小
$\not\subset x = 1$	大小
$\not\subset (A, B) + \not\subset (A'B) = \not\subset A + \not\subset B$	大小
$\neg x: A \Rightarrow \not\subset (A'x) = 0$	大小
$A: B \Rightarrow \not\subset A \le \not\subset B$	大小
$A,(A'B) = A$	吸收性
$A'(A,B) = A$	吸收性
$A: B = A,B = B = A = A'B$	包含性
$A,(B,C) = (A,B),(A,C)$	分配性
$A,(B'C) = (A,B)'(A,C)$	分配性
$A'(B,C) = (A'B), (A'C)$	分配性
$A'(B'C) = (A'B)'(A'C)$	分配性
$A: B \wedge C: D \Rightarrow A,C: B,D$	合并，单调性
$A: B \wedge C: D \Rightarrow A'C: B'D$	合并，单调性
$null: A$	归纳
$A, null = A$	恒等性
$A ' null = null$	基
$\not\subset A = 0 = A = null$	大小
$x:int \wedge y:int \wedge x \le y \Rightarrow (i: x,..y = i:int \wedge x \le i<y)$	
$x:int \wedge y:int \wedge x \le y \Rightarrow \not\subset (x,..y) = y-x$	
$-null = null$	分配性
$-(A, B) = -A, -B$	分配性
$A + null = null + A = null$	分配性

$(A, B)+(C, D) = A+C, A+D, B+C, B+D$ 分配性

对许多其它运算符也类似（见本书最后页）

----------束结束

11.4.4 集合

$\{\sim A\} = A$

$\sim \{A\} = A$

$\{A\} \neq A$

$A \in \{B\} = A : B$

$\{A\} \subseteq \{B\} = A : B$

$\{A\} \in \not\vdash B = A : B$

$\$ \{A\} = \not\subset A$

$\{A\} \cup \{B\} = \{A, B\}$

$\{A\} \cap \{B\} = \{A \text{ ‘ } B\}$

$\{A\} = \{B\} = A = B$

$\{A\} \neq \{B\} = A \neq B$

----------集合结束

11.4.5 字符串

令 S，T,和 U 是串；令 i 和 j 是项（二元值，数，字符，集合，表，函数）；令 n 为广义自然数；令 x，y，和 z 为整数。

$nil; S = S; nil = S$

$S; (T; U) = (S; T); U$

$\leftrightarrow nil = 0$

$\leftrightarrow i = 1$

$\leftrightarrow(S; T) = \leftrightarrow S + \leftrightarrow T$

$S_{nil} = nil$

$\leftrightarrow S < \infty \Rightarrow (S; i; T)_{\leftrightarrow S} = i$

$S_{T; U} = S_T; S_U$

$S_{(T_U)} = (S_T)_U$

$S_{\{A\}} = \{S_A\}$

$\leftrightarrow S < \infty \Rightarrow nil \leq S < S; i; T$

$\leftrightarrow S < \infty \Rightarrow (i < j \Rightarrow S; i; T < S; j; U)$

$\leftrightarrow S < \infty \Rightarrow (i = j = S; i; T = S; j; T)$

$0 * S = nil$

$(n+1) * S = n * S; S$

$\leftrightarrow S < \infty \Rightarrow S;i;T \triangleleft \leftrightarrow S \triangleright j = S;j;T$

$x;..x = nil$

$x;..x+1 = x$

$(x;..y) ; (y;..z) = x;..z$

$\leftrightarrow(x;..y) = y–x$

----------字符串结束

11.4.6 表

令S和T为字符串；令 i和 j 为项（二元值，数，字符，集合，表，函数）；令 L , M , 和 N为表。

$[S] \neq S$
$\#[S] = \leftrightarrow S$
$\sim[S] = S$
$S_{[T]} = [S_T]$
$[\sim L] = L$
$[S]\ [T] = [S_T]$
$[S]\ T = S_T$
$L\ \{A\} = \{L\ A\}$
$[S]+[T] = [S;\ T]$
$L\ [S] = [L\ S]$
$[S] = [T] = S = T$
$(L\ M)\ N = L\ (M\ N)$

$[S] < [T] = S < T$
$L@nil = L$
$nil \rightarrow i \mid L = i$
$L@i = L\ i$
$n \rightarrow i \mid [S] = [S \lhd n \rhd i]$
$L@(S;\ T) = L@S@T$
$(S;T) \rightarrow i \mid L = S \rightarrow (T \rightarrow i \mid L@S) \mid L$

----------表结束

11.4.7 函数

换名公理 — 若 v和w不出现在D中，且w不出现在b中
$\langle v{:}\ D \rightarrow b\rangle = \langle w{:}\ D \rightarrow \langle v{:}\ D \rightarrow b\rangle w\rangle$

应用公理：若元素$x{:}\ D$
$\langle v{:}\ D \rightarrow b\rangle x =$ (在b中以x置换v)

扩展定律
$f = \langle v{:}\ \Box f \rightarrow f\ v\rangle$

定义域公理
$\Box\langle v{:}\ D \rightarrow b\rangle = D$

函数组合公理：若 $\neg\ f{:}\ \Box g$
$\Box(g\ f) = \S x{:}\ \Box f{\cdot}\ fx{:}\ \Box g$
$(g\ f)\ x = g\ (f\ x)$
$f\ (g\ h) = (f\ g)\ h$

函数包含定律
$f{:}\ g = \Box g{:}\ \Box f \wedge \forall x{:}\ \Box g{\cdot}\ fx{:}\ gx$

基数公理
$\not\subset A = \Sigma\ (A \rightarrow 1)$

函数相等定律

$f = g = \Box f = \Box g \wedge \forall x{:}\ \Box f \cdot fx = gx$

函数交公理

$\Box(f \text{ ‘ } g) = \Box f, \Box g$

$(f \text{ ‘ } g)\ x = (f \mid g)\ x \text{ ‘ } (g \mid f)\ x$

函数并公理

$\Box(f, g) = \Box f \text{ ‘} \Box g$

$(f, g)\ x = f\ x, g\ x$

选择合并定律

$f \mid f = f$

$(g \mid h)\ f = g\ f \mid h\ f$

$\langle v{:}\ A \rightarrow x \rangle \mid \langle v{:}\ B \rightarrow y \rangle = \langle v{:}\ A, B \rightarrow \textbf{if } v{:}\ A \textbf{ then } x \textbf{ else } y \textbf{ fi} \rangle$

选择合并定律

$\Box(f \mid g) = \Box f, \Box g$

$(f \mid g)\ x = \textbf{if } x{:}\ \Box f \textbf{ then } f\ x \textbf{ else } g\ x \textbf{ fi}$

$f \mid (g \mid h) = (f \mid g) \mid h$

分配定律

$f\ null = null$

$f\ (A, B) = f\ A, f\ B$

$f\ (\S g) = \S y{:}\ f\ (\Box g) \cdot \exists x{:}\ \Box g \cdot fx = y \wedge gx$

$f \textbf{ if } b \textbf{ then } x \textbf{ else } y \textbf{ fi} = \textbf{if } b \textbf{ then } f\ x \textbf{ else } f\ y \textbf{ fi}$

$\textbf{if } b \textbf{ then } f \textbf{ else } g \textbf{ fi}\ x = \textbf{if } b \textbf{ then } f\ x \textbf{ else } g\ x \textbf{ fi}$

箭头定律

$f{:}\ null \rightarrow A$

$A \rightarrow B{:}\ (A \text{‘} C) \rightarrow (B, D)$

$f{:}\ A \rightarrow B = A{:}\ \Box f \wedge \forall a{:}\ A \cdot fa{:}\ B$

----------函数结束

11.4.8 量词

令 x为元素，令 a , b和 c为二元值，令 n和m为数，令 f和g为函数，令P为谓词。

$\forall v{:}\ null \cdot b = \top$

$\forall v{:}\ A, B \cdot b = (\forall v{:}\ A \cdot b) \wedge (\forall v{:}\ B \cdot b)$

$\forall v{:}\ x \cdot b = \langle v{:}\ x \rightarrow b \rangle\ x$

$\forall v{:}\ (\S v{:}\ D \cdot b) \cdot c = \forall v{:}\ D \cdot b \Rightarrow c$

$\exists v{:}\ null \cdot b = \bot$

$\exists v{:}\ A, B \cdot b = (\exists v{:}\ A \cdot b) \vee (\exists v{:}\ B \cdot b)$

$\exists v{:}\ x \cdot b = \langle v{:}\ x \rightarrow b \rangle\ x$

$\exists v{:}\ (\S v{:}\ D \cdot b) \cdot c = \exists v{:}\ D \cdot b \wedge c$

$\Sigma v{:}\ null \cdot n = 0$

$(\Sigma v{:}\ A, B \cdot n) + (\Sigma v{:}\ A \text{‘} B \cdot n) = (\Sigma v{:}\ A \cdot n) + (\Sigma v{:}\ B \cdot n)$

$\Sigma v{:}\ x \cdot n = \langle v{:}\ x \rightarrow n \rangle\ x$

$\Sigma v\colon (\S v\colon D\cdot b)\cdot n = \Sigma v\colon D\cdot$ **if** b **then** n **else** 0 **fi**

$\Pi v\colon null\cdot n = 1$
$(\Pi v\colon A,B\cdot n) \times (\Pi v\colon A`B\cdot n) = (\Pi v\colon A\cdot n) \times (\Pi v\colon B\cdot n)$
$\Pi v\colon x\cdot n = \langle v\colon x \rightarrow n\rangle\, x$
$\Pi v\colon (\S v\colon D\cdot b)\cdot n = \Pi v\colon D\cdot$ **if** b **then** n **else** 1 **fi**

$MIN\ v\colon null\cdot n = \infty$
$MIN\ v\colon A,B\cdot n = min\ (MIN\ v\colon A\cdot n)\ (MIN\ v\colon B\cdot n)$
$MIN\ v\colon x\cdot n = \langle v\colon x \rightarrow n\rangle\, x$
$MIN\ v\colon (\S v\colon D\cdot b)\cdot n = MIN\ v\colon D\cdot$ **if** b **then** n **else** ∞ **fi**

$MAX\ v\colon null\cdot n = -\infty$
$MAX\ v\colon A,B\cdot n = max\ (MAX\ v\colon A\cdot n)\ (MAX\ v\colon B\cdot n)$
$MAX\ v\colon x\cdot n = \langle v\colon x \rightarrow n\rangle\, x$
$MAX\ v\colon (\S v\colon D\cdot b)\cdot n = MAX\ v\colon D\cdot$ **if** b **then** n **else** $-\infty$ **fi**

$(MIN\ i\colon int\cdot i) = -\infty$
$(MAX\ i\colon int\cdot i) = \infty$

$\S v\colon null\cdot b = null$
$\S v\colon x\cdot b =$ **if** $\langle v\colon x \rightarrow b\rangle\, x$ **then** x **else** $null$ **fi**
$\S v\colon A,B\cdot b = (\S v\colon A\cdot b), (\S v\colon B\cdot b)$
$\S v\colon A`B\cdot b = (\S v\colon A\cdot b)\ `\ (\S v\colon B\cdot b)$
$\S v\colon (\S v\colon D\cdot b)\cdot c = \S v\colon D\cdot b \wedge c$

变量改变定律 — 若 d不出现在b中
$\forall r\colon fD\cdot b = \forall d\colon D\cdot \langle r\colon fD \rightarrow b\rangle\ (fd)$
$\exists r\colon fD\cdot b = \exists d\colon D\cdot \langle r\colon fD \rightarrow b\rangle\ (fd)$
$\Sigma r\colon fD\cdot n = \Sigma d\colon D\cdot$ (在n中以fd置换r)
$MIN\ r\colon fD\cdot n = MIN\ d\colon D\cdot \langle r\colon fD \rightarrow b\rangle\ (fd)$
$MAX\ r\colon fD\cdot n = MAX\ d\colon D\cdot \langle r\colon fD \rightarrow b\rangle\ (fd)$

束-元素转换定律
$V\colon W = \forall v\colon V\cdot \exists w\colon W\cdot v{=}w$
$fV\colon gW = \forall v\colon V\cdot \exists w\colon W\cdot fv{=}gw$

恒等定律
$\forall v\cdot \top$
$\neg\exists v\cdot \bot$

幂等定律 — 若 $D \neq null$且v不出现在b 中
$\forall v\colon D\cdot b = b$
$\exists v\colon D\cdot b = b$

吸收定律 — 若 $x\colon D$

$\langle v{:}\ D \to b \rangle\ x \wedge \exists v{:}\ D \cdot b = \langle v{:}\ D \to b \rangle\ x$

$\langle v{:}\ D \to b \rangle\ x \vee \forall v{:}\ D \cdot b = \langle v{:}\ D \to b \rangle\ x$

$\langle v{:}\ D \to b \rangle\ x \wedge \forall v{:}\ D \cdot b = \forall v{:}\ D \cdot b$

$\langle v{:}\ D \to b \rangle\ x \vee \exists v{:}\ D \cdot b = \exists v{:}\ D \cdot b$

分配定律 — 若 $D \neq null$且v不出现在a 中

$a \wedge \forall v{:}\ D \cdot b = \forall v{:}\ D \cdot a \wedge b$

$a \wedge \exists v{:}\ D \cdot b = \exists v{:}\ D \cdot a \wedge b$

$a \vee \forall v{:}\ D \cdot b = \forall v{:}\ D \cdot a \vee b$

$a \vee \exists v{:}\ D \cdot b = \exists v{:}\ D \cdot a \vee b$

$a \Rightarrow \forall v{:}\ D \cdot b = \forall v{:}\ D \cdot a \Rightarrow b$

$a \Rightarrow \exists v{:}\ D \cdot b = \exists v{:}\ D \cdot a \Rightarrow b$

反分配定律 — 若 $D \neq null$且v不出现在a 中

$a \Leftarrow \exists v{:}\ D \cdot b = \forall v{:}\ D \cdot a \Leftarrow b$

$a \Leftarrow \forall v{:}\ D \cdot b = \exists v{:}\ D \cdot a \Leftarrow b$

特定化定律 — 若 $x{:}\ D$

$\forall v{:}\ D \cdot b \Longrightarrow \langle v{:}\ D \to b \rangle\ x$

普遍化定律 — 若 $x{:}\ D$

$\langle v{:}\ D \to b \rangle\ x \Longrightarrow \exists v{:}\ D \cdot b$

单点定律 — 若 $x{:}\ D$且v不出现在x中

$\forall v{:}\ D \cdot v{=}x \Rightarrow b = \langle v{:}\ D \to b \rangle\ x$

$\exists v{:}\ D \cdot v{=}x \wedge b = \langle v{:}\ D \to b \rangle\ x$

分裂定律 — 对任何固定定义域

$\forall v \cdot a \wedge b = (\forall v \cdot a) \wedge (\forall v \cdot b)$

$\exists v \cdot a \wedge b \Longrightarrow (\exists v \cdot a) \wedge (\exists v \cdot b)$

$\forall v \cdot a \vee b \Longleftarrow (\forall v \cdot a) \vee (\forall v \cdot b)$

$\exists v \cdot a \vee b = (\exists v \cdot a) \vee (\exists v \cdot b)$

$\forall v \cdot a \Rightarrow b \Longrightarrow (\forall v \cdot a) \Rightarrow (\forall v \cdot b)$

$\forall v \cdot a \Rightarrow b \Longrightarrow (\exists v \cdot a) \Rightarrow (\exists v \cdot b)$

$\forall v \cdot a = b \Longrightarrow (\forall v \cdot a) = (\forall v \cdot b)$

$\forall v \cdot a = b \Longrightarrow (\exists v \cdot a) = (\exists v \cdot b)$

二元定律

$\neg \forall v \cdot b = \exists v \cdot \neg b$ (德摩根)

$\neg \exists v \cdot b = \forall v \cdot \neg b$ (德摩根)

$-\mathit{MAX}\ v \cdot n = \mathit{MIN}\ v \cdot -n$

$-\mathit{MIN}\ v \cdot n = \mathit{MAX}\ v \cdot -n$

交换定律

$\forall v \cdot \forall w \cdot b = \forall w \cdot \forall v \cdot b$

$\exists v \cdot \exists w \cdot b = \exists w \cdot \exists v \cdot b$

解定律

$\S v{:}\ D{\cdot}\top = D$

$(\S v{:}\ D{\cdot}\ b){:}\ D$

$\S v{:}\ D{\cdot}\ \bot = null$

$(\S v{\cdot}\ b){:}\ (\S v{\cdot}\ c) \ = \forall v{\cdot}\ b \Rightarrow c$

$(\S v{\cdot}\ b),\ (\S v{\cdot}\ c) = \S v{\cdot}\ b \vee c$

$(\S v{\cdot}\ b)\ `\ (\S v{\cdot}\ c) = \S v{\cdot}\ b \wedge c$

$x{:}\ \S p = x{:}\ \Box p \wedge px$

$\forall f = (\S f) = (\Box f)$

$\exists f = (\S f) \neq null$

半交换定律(Skolem)

$\exists v{\cdot}\ \forall w{\cdot}\ b \Longrightarrow \forall w{\cdot}\ \exists v{\cdot}\ b$

$\forall x{\cdot}\ \exists y{\cdot}\ pxy = \exists f{\cdot}\ \forall x{\cdot}\ px(fx)$

定义域改变定律

$A{:}\ B \Longrightarrow (\forall v{:}\ A{\cdot}\ b) \Leftarrow \ (\forall v{:}\ B{\cdot}\ b)$

$A{:}B \Longrightarrow (\exists v{:}\ A{\cdot}\ b) \Rightarrow \ (\exists v{:}\ B{\cdot}\ b)$

$\forall v{:}\ A{\cdot}\ v{:}\ B \Rightarrow p = \forall v{:}\ A`B{\cdot}\ p$

$\exists v{:}\ A{\cdot}\ v{:}\ B \wedge p = \exists v{:}\ A`B{\cdot}\ p$

约束定律—若 v不出现在 n中

$n > (MAX\ v{:}\ D{\cdot}\ m) \Rightarrow (\forall v{:}\ D{\cdot}\ n{>}m)$

$n < (MIN\ v{:}\ D{\cdot}\ m) \Rightarrow (\forall v{:}\ D{\cdot}\ n{<}m)$

$n \geq (MAX\ v{:}\ D{\cdot}\ m) = (\forall v{:}\ D{\cdot}\ n{\geq}m)$

$n \leq (MIN\ v{:}\ D{\cdot}\ m) = (\forall v{:}\ D{\cdot}\ n{\leq}m)$

$n \geq (MIN\ v{:}\ D{\cdot}\ m) \Leftarrow (\exists v{:}\ D{\cdot}\ n{\geq}m)$

$n \leq (MAX\ v{:}\ D{\cdot}\ m) \Leftarrow (\exists v{:}\ D{\cdot}\ n{\leq}m)$

$n > (MIN\ v{:}\ D{\cdot}\ m) = (\exists v{:}\ D{\cdot}\ n{>}m)$

$n < (MAX\ v{:}\ D{\cdot}\ m) = (\exists v{:}\ D{\cdot}\ n{<}m)$

极值定律

$(MIN\ v{\cdot}\ n) \leq n \leq (MAX\ v{\cdot}\ n)$

连接定律 (伽罗瓦)

$n{\leq}m = \forall k{\cdot}\ k{\leq}n \Rightarrow k{\leq}m$

$n{\leq}m = \forall k{\cdot}\ k{<}n \Rightarrow k{<}m$

$n{\leq}m = \forall k{\cdot}\ m{\leq}k \Rightarrow n{\leq}k$

$n{\leq}m = \forall k{\cdot}\ m{<}k \Rightarrow n{<}k$

分配定律— if $D \neq null$ 且 v不出现在 n中

$max\ n\ (MAX\ v{:}\ D{\cdot}\ m) = (MAX\ v{:}\ D{\cdot}\ max\ n\ m)$

$max\ n\ (MIN\ v{:}\ D{\cdot}\ m) = (MIN\ v{:}\ D{\cdot}\ max\ n\ m)$

$min\ n\ (MAX\ v{:}\ D{\cdot}\ m) = (MAX\ v{:}\ D{\cdot}\ min\ n\ m)$

$min\ n\ (MIN\ v{:}\ D{\cdot}\ m) = (MIN\ v{:}\ D{\cdot}\ min\ n\ m)$

$n + (MAX\ v{:}\ D{\cdot}\ m) = (MAX\ v{:}\ D{\cdot}\ n{+}m)$

$n + (MIN\ v{:}\ D{\cdot}\ m) = (MIN\ v{:}\ D{\cdot}\ n{+}m)$

$n - (MAX\ v{:}\ D{\cdot}\ m) = (MIN\ v{:}\ D{\cdot}\ n-m)$
$n - (MIN\ v{:}\ D{\cdot}\ m) = (MAX\ v{:}\ D{\cdot}\ n-m)$
$(MAX\ v{:}\ D{\cdot}\ m) - n = (MAX\ v{:}\ D{\cdot}\ m-n)$
$(MIN\ v{:}\ D{\cdot}\ m) - n = (MIN\ v{:}\ D{\cdot}\ m-n)$
$n{\geq}0 \Rightarrow n \times (MAX\ v{:}\ D{\cdot}\ m) = (MAX\ v{:}\ D{\cdot}\ n{\times}m)$
$n{\geq}0 \Rightarrow n \times (MIN\ v{:}\ D{\cdot}\ m) = (MIN\ v{:}\ D{\cdot}\ n{\times}m)$
$n{\leq}0 \Rightarrow n \times (MAX\ v{:}\ D{\cdot}\ m) = (MIN\ v{:}\ D{\cdot}\ n{\times}m)$
$n{\leq}0 \Rightarrow n \times (MIN\ v{:}\ D{\cdot}\ m) = (MAX\ v{:}\ D{\cdot}\ n{\times}m)$
$n \times (\Sigma v{:}\ D{\cdot}\ m) = (\Sigma v{:}\ D{\cdot}\ n{\times}m)$
$(\Pi v{:}\ D{\cdot}\ m)^n = (\Pi v{:}\ D{\cdot}\ m^n)$

----------量词结束

11.4.9 极限

$(MAX\ m{\cdot}\ MIN\ n{\cdot}\ f(m+n)) \leq (LIM\ f) \leq (MIN\ m{\cdot}\ MAX\ n{\cdot}\ f(m+n))$

$\exists m{\cdot}\ \forall n{\cdot}\ p(m+n) \Longrightarrow LIM\ p \Longrightarrow \forall m{\cdot}\ \exists n{\cdot}\ p(m+n)$

$(LIM\ n{\cdot}\ n) = \infty$

----------极限结束

11.4.10 规范与程序

对于规范P , Q , R 和 S ， 以及二元值b，

$ok = x'{=}x \wedge y'{=}y \wedge \ldots$

$x{:=}\ e = x'{=}e \wedge y'{=}y \wedge \ldots$

$P.\ Q = \exists x'', y'', \ldots{\cdot}\ \langle x', y', \ldots \rightarrow P\rangle x''y''\ldots \wedge \langle x, y, \ldots \rightarrow Q\rangle x''y''\ldots$

$P \| Q = \exists t_P, t_Q{\cdot}\ \langle t' \rightarrow P\rangle t_P \wedge \langle t' \rightarrow Q\rangle t_Q \wedge\ t' = max\ t_P\ t_Q$

if b **then** P **else** Q **fi** $= b \wedge P \vee \neg b \wedge Q$

var $x{:}\ T{\cdot}\ P = \exists x, x'{:}\ T{\cdot}\ P$

frame $x{\cdot}\ P = P \wedge y'{=}y \wedge \ldots$

while b **do** P **od** $= t'{\geq}t \wedge$ **if** b **then** $P.\ t{:=}\ t{+}1.$ **while** b **do** P **od else** ok **fi**

$\forall \sigma, \sigma'{\cdot}$ **if** b **then** $P.\ W$ **else** ok **fi** $\Leftarrow W \Longrightarrow \forall \sigma, \sigma'{\cdot}$ **while** b **do** P **od** $\Leftarrow W$

$(Fmn \Leftarrow m{=}n \wedge ok) \wedge (Fik \Leftarrow m{\leq}i{<}j{<}k{\leq}n \wedge (Fij.\ Fjk))$

$\Longrightarrow$ $Fmn \Leftarrow$ **for** $i{:=}\ m;..n$ **do** $m{\leq}i{<}n \Rightarrow Fi(i{+}1)$ **od**

$Im \Rightarrow I'n \Leftarrow$ **for** $i{:=}\ m;..n$ **do** $m{\leq}i{<}n \wedge Ii \Rightarrow I'(i{+}1)$ **od**

wait until $w = t{:=}\ max\ t\ w$

assert $b =$ **if** b **then** ok **else** $print$ “error”. **wait until** ∞ **fi**

ensure $b = b \wedge ok$

$P.\ (P$ **result** $e) = e$ 但是不要在(P **result** e)里加双撇号或进行替换

$c? = r{:=}\ r{+}1$

$c = Mc_{\ rc\text{-}1}$

$c!\ e = Mc_{\ wc} = e \wedge Tc_{\ wc} = t \wedge (wc{:=}\ wc{+}1)$

$\surd c = Tc_{\ rc} +$ (运输时间) $\leq t$

ivar $x{:}\ T{\cdot}\ S = \exists x{:}\ time \rightarrow T{\cdot}\ S$

chan $c{:}\ T{\cdot}\ P = \exists Mc{:}\ \infty{*}T{\cdot}\ \exists Tc{:}\ \infty{*}xreal{\cdot}$ **var** rc , $wc{:}\ xnat := 0{\cdot}\ P$

$ok.\ P = P.\ ok = P$ 恒等性

$P.\ (Q.\ R) = (P.\ Q).\ R$ 结合性

$P \vee Q.\ R \vee S = (P.\ R) \vee (P.\ S) \vee (Q.\ R) \vee (Q.\ S)$ 分配性

if b **then** P **else** Q **fi**. R = **if** b **then** P. R **else** Q. R **fi**　分配性 (去掉b上撇号)

P. **if** b **then** Q **else** R **fi** = **if** P. b **then** P. Q **else** P. R **fi**分配性 (去掉b上撇号)

$P \parallel Q = Q \parallel P$　对称性

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

$P \parallel t' \leftarrow t = P = t' \leftarrow t \parallel P$　恒等性

$P \parallel Q \vee R = (P \parallel Q) \vee (P \parallel R)$　分配性

$P \parallel$ **if** b **then** Q **else** R **fi** = **if** b **then** $P \parallel Q$ **else** $P \parallel R$ **fi** 分配性

if b **then** $P \| Q$ **else** $R \| S$ **fi** = **if** b **then** P **else** R **fi** $\parallel$ **if** b **then** Q **else** S **fi**　分配性

x:= **if** b **then** e **else** f **fi** = **if** b **then** x:= e **else** x:= f **fi**　函数-命令

----------规范与程序结束

11.4.11 置换

令x和 y为不同的边界状态变量，令e和f为前置状态表达式，令P为一个规范。

x:= e. P = (在P中置换x为e)

(x:= $e \parallel y$:= f). P = (在P中置换x为e，且独立地置换y为 f)

----------置换结束

11.4.12 条件

令 P和Q为任意规范，令C为一个前置条件，令C'为相应的后置条件（换句话说，除了在所有状态变量上加一撇外，C'与C其他方面都相同）。

$C \wedge (P.\ Q) \Leftarrow C \wedge P.\ Q$

$C \Rightarrow (P.Q) \Leftarrow C \Rightarrow P.\ Q$

$(P.Q) \wedge C' \Leftarrow P.\ Q \wedge C'$

$(P.Q) \Leftarrow C' \Leftarrow P.\ Q \Leftarrow C'$

$P.\ C \wedge Q \Leftarrow P \wedge C'.\ Q$

$P.\ Q \Leftarrow P \wedge C'.\ C \Rightarrow Q$

C是P被S精化的充分前置条件当且仅当$C \Rightarrow P$被S精化。

C' 是P被S精化的充分前置条件当且仅当$C' \Rightarrow P$被S精化。

----------条件结束

11.4.13 精化

逐步精化法（逐步地精化）（单调性,传递性）

若 $A \Leftarrow$ **if** b **then** C **else** D **fi** 和$C \Leftarrow E$ 和$D \Leftarrow F$ 为定理，

则 $A \Leftarrow$ **if** b **then** E **else** F **fi** 为定理。

若 $A \Leftarrow B.\ C$ 和 $B \Leftarrow D$ 和 $C \Leftarrow E$ 为定理，则 $A \Leftarrow D.\ E$ 为定理。

若 $A \Leftarrow B \| C$ 和 $B \Leftarrow D$ 和 $C \Leftarrow E$ 为定理，则 $A \Leftarrow D \| E$ 为定理。

若 $A \Leftarrow B$ 和 $B \Leftarrow C$ 为定理，则 $A \Leftarrow C$ 为定理。

部分精化法（单调性，归并）

若 $A \Leftarrow$ **if** b **then** C **else** D **fi** 和 $E \Leftarrow$ **if** b **then** F **else** G **fi** 为定理，

则 $A \wedge E \Leftarrow$ **if** b **then** $C \wedge F$ **else** $D \wedge G$ **fi** 为定理。

若 $A \Leftarrow B.\ C$ 和 $D \Leftarrow E.\ F$ 为定理，则 $A \wedge D \Leftarrow B \wedge E.\ C \wedge F$ 为定理。

若 $A \Leftarrow B \| C$ 和 $D \Leftarrow E \| F$ 为定理，则 $A \wedge D \Leftarrow B \wedge E \parallel C \wedge F$ 为定理。

若 $A \Leftarrow B$ 和 $C \Leftarrow D$ 为定理，则 $A \wedge C \Leftarrow B \wedge D$ 为定理。

情况精化法

$P \Leftarrow$ **if** b **then** Q **else** R **fi** 为定理当且仅当

$P \Leftarrow b \wedge Q$ 和 $P \Leftarrow \neg b \wedge R$ 为定理。

----------精化结束

----------公理和定律结束

11.5 名字

abs: *xreal*→§*r*: *xreal*· *r*≥0	*abs r* = **if** *r*≥0 **then** *r* **else** –*r*
bin (二元值)	*bin* = ⊤ , ⊥
ceil: *real*→*int*	*r* ≤ *ceil r* < *r*+1
char (字符)	*char* = ..., “a”, “A”, ...
div: *real*→(§*r*: *real*· *r*>0)→*int*	*div x y* = *floor* (*x*/*y*)
divides: (*nat*+1)→*int*→*bin*	*divides n i* = *i*/*n*: *int*
entro: *prob*→§*r*: *xreal*· *r*≥0	*entro p* = *p* × *info p* + (1–*p*) × *info* (1–*p*)
even: *int*→*bin*	*even i* = *i*/2: *int*
	even = *divides* 2
floor: *real*→*int*	*floor r* ≤ *r* < *floor r* + 1
info: *prob*→§*r*: *xreal*· *r*≥0	*info p* = – *log p*
int (整数)	*int* = *nat*, –*nat*
LIM (极限量词)	见公理和定律
log: (§*r*: *xreal*· *r*≥0)→*xreal*	*log* (2^x) = *x*
	log (*x*×*y*) = *log x* + *log y*
max: *xrat*→*xrat*→*xrat*	*max x y* = **if** *x*≥*y* **then** *x* **else** *y* **fi**
	– *max a b* = *min* (–*a*) (–*b*)
MAX (最大量词)	见公理与定律
min: *xrat*→*xrat*→*xrat*	*min x y* = **if** *x*≤*y* **then** *x* **else** *y* **fi**
	– *min a b* = *max* (–*a*) (–*b*)
MIN (最小量词)	见公理与定律
mod: *real*→(§*r*: *real*· *r*>0)→*real*	0 ≤ *mod a d* < *d*
	a = *div a d* × *d* + *mod a d*
nat (自然数)	0, *nat*+1: *nat*
	0, *B*+1: *B* ⇒ *nat*: *B*
nil (空串)	↔*nil* = 0
	nil; *S* = *S* = *S*; *nil*
	nil ≤ *S*
null (空束)	¢*null* = 0
	null, *A* = *A* = *A*, *null*
	null: *A*
odd: *int*→*bin*	*odd i* = ¬ *i*/2: *int*
	odd = ¬*even*
ok (空程序)	*ok* = σ′=σ

	ok. *P* ═ *P* ═ *P*. *ok*
prob (概率)	*prob* = §*r*: *real*· 0≤*r*≤1
rand (随机数)	*rand n*: 0,..*n*
rat (有理数)	*rat* = *int*/(*nat*+1)
real (实数)	*r*: *real* ═ *r*: *xreal* ∧ –∞<*r*<∞
suc: *nat*→(*nat*+1)	*suc n* = *n*+1
xint (广义整数)	*xint* = –∞, *int*, ∞
xnat (广义自然数)	*xnat* = *nat*, ∞
xrat (广义有理数)	*xrat* = –∞, *rat*, ∞
xreal (广义实数)	*x*: *xreal* ═ ∃*f*: *nat*→*rat*· *x* = *LIM f*

----------名字结束

11.6 符号

T	真值	( )	括号
⊥	假值	{ }	集合括号
¬	非	[ ]	表括号
∧	与	⟨ ⟩	函数（范围）括号
∨	或	⚡	幂集
⇒	蕴含，等于或强于	¢	束的大小，基数
⟹	蕴含，等于或强于	$	集合大小，基数
⇐	被蕴含，由…导出	↔	串大小，长度
⟸	被蕴含，由…导出	#	表大小，长度
=	等于，当且仅当	|	选择合并，否则
═	等于，当且仅当	‖	独立（并行）组合
≠	不同于，不等于	~	集合或表的内容
<	小于		
>	大于	*	串的重复
≤	小于或等于	□	函数的定义域
≥	大于或等于	→	函数箭头
+	加	∈	集合的元素
+	表连接	⊆	子集
–	减	∪	并集
×	相乘，乘法	∩	交集
/	除	@	指针索引
,	束的并	∀	对所有，全称量词
,..	从（包括）到（不包括）的束并	∃	存在，存在量词
‘	束的交	Σ	求和，累加量词
;	串连接	Π	乘积，累乘量词
;..	从（包括）到（不包括）的连接	§	那些，解量词
:	在里面，束包含	′	*x*′是状态变量*x*的最终值
::	包含	“	“hi”是一个文本或字符串
:=	赋值	a^b	指数
.	相关（顺序）组合	a_b	串索引
·	量词缩写	*a b*	索引，应用，组合
!	输出	◅ ▻	字符串更改
?	输入	∞	无限

√　　输入检查

assert　　**if then else fi**
chan　　**ivar**
do od　　**or**
ensure　　**result**
exit when　　**var**
for do od　　**wait until**
frame　　**while do od**
go to

----------符号结束

11.7 优先级

0　　⊤ ⊥ ( ) { } [ ] ⟨ ⟩ **if fi**　**do od** 数 文本 名字 上标 下标
1　　@　并置
2　　前缀– ¢ $ ↔ # * ~ ⚡　→ √
3　　×　/　⋂
4　　+　中缀–　+　⋃
5　　;　;..　‘
6　　,　,..　|　◅ ▻
7　　=　≠ <　>　≤　≥　:　::　∈　⊆
8　　¬
9　　∧
10　　∨
11　　⇒　⇐
12　　:=　!　?
13　　**exit when**　**go to**　**wait until**　**assert**　**ensure**　**or**
14　　.　‖　**result**
15　　∀· ∃·　Σ·　Π·　§·　*LIM*·　*MAX*·　*MIN*· **var**· **ivar**· **chan**· **frame**·
16　　＝　⇒　⇐

上标和下标中的运算相当于加了括号内的运算。

并置从左向右结合，因此*a b c*等同于(*a b*) *c*。中缀运算符 @　/　– 从左向右结合。中缀运算符 *　→ 从右向左结合。中缀运算符 ×　⋂　+　+　⋃　;　‘　,　|　∧　∨　.　‖ 是可结合的（可从两个方向结合）。

在第 7, 11, 和16级，运算符是连续的。例如，*a=b=c* 既不向左也不向右结合，但等同于*a* = *b* ∧ *b* = *c*。在任意级别的连续运算符可以混合使用。例如 *a*≤*b*<*c* 等同于 *a*≤*b* ∧ *b*<*c*。

运算符 ＝　⇒　⇐ 除优先级别外与 =　⇒　⇐ 等同。

----------优先级结束

11.8 分配性

下列表达式中的运算符对于任意运算对象可分布在束的并上：

[*A*]　*A@B*　*A B*　*-A*　*$A*　*↔A*　*#A*　*~A*
A^B　A_B　*A*×*B*　*A*/*B*　*A*⋂*B*　*A*+*B*　*A*-*B*　*A*+*B*　*A*⋃*B*　*A*;*B*　*A*‘*B*
¬*A*　*A*∧*B*　*A*∨*B*

*A***B*中的运算符仅对于左侧的运算对象可分布在束的并上。

----------分配性结束
----------参考结束
----------一种实用的程序设计理论结束