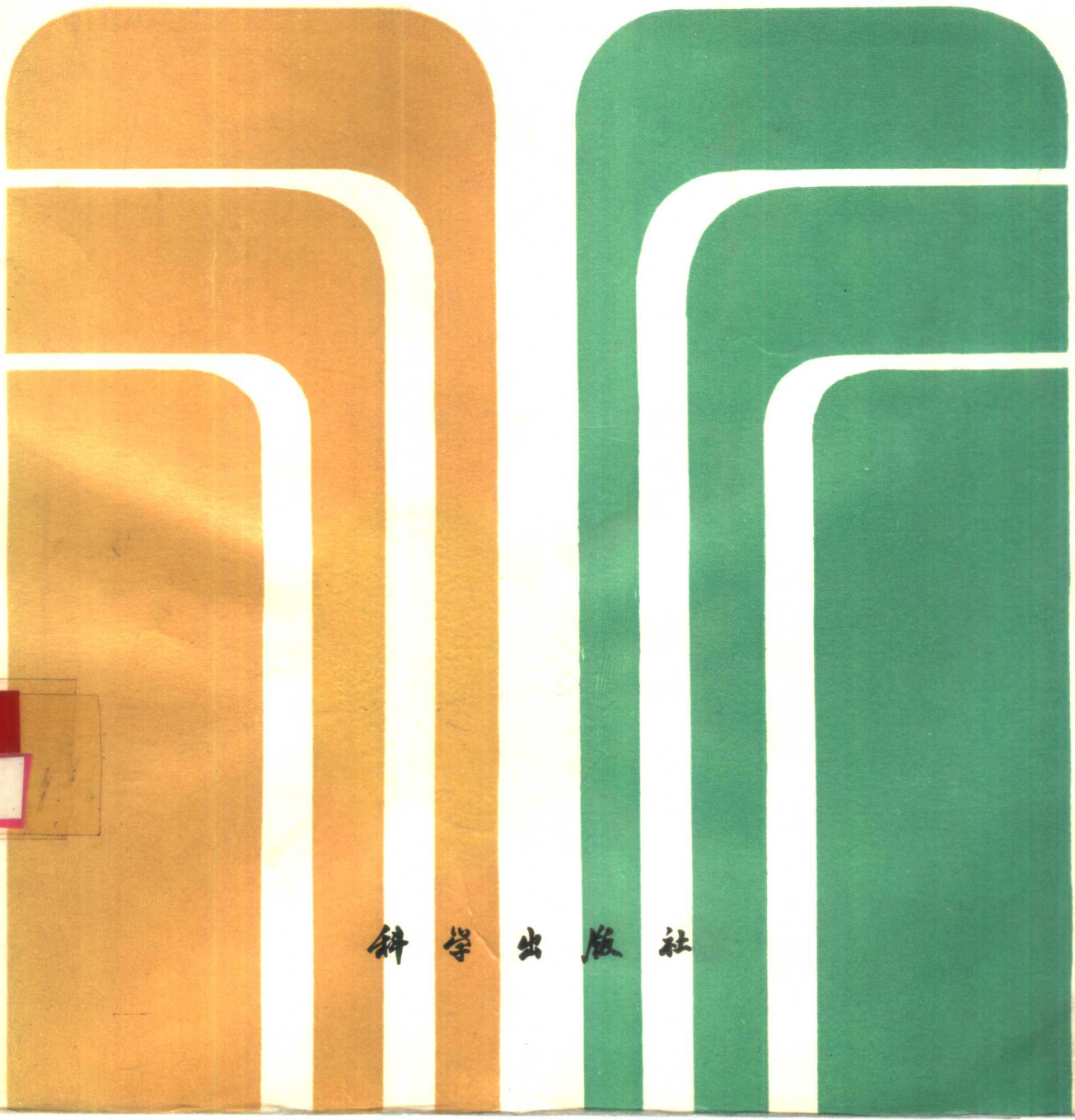
实用类型程序设计

屈延文　邱续欣 著



科学出版社

实用类型程序设计

屈延文　邱续欣 著

科学出版社
1992

（京）新登字 092 号

内 容 简 介

本书详细讨论了面向软件重用与软件自动生成的实用规模软件开发的新技术——类型程序设计方法学．围绕这一主题，本书还讨论了类型程序设计的基本实施原则．

本书共分七章，包括类型程序设计概念，类型函数的算法分析与设计，自下而上构造式的类型程序设计，自上而下的推导式的类型程序设计，并发类型程序设计，大型软件的程序设计，Martin-Löf 类型论及用类型表达式描述的 VAX/VMS 的一个简明的类型语义．

本书内容丰富，重点突出，新颖而实用．书中配备了大量的 VAX C 语言的实例程序及程序设计习题．它可以作为高等院校软件专业高年级学生、研究生程序设计课程的教材，也可以作为计算机高级程序员培训及知识更新的教材或参考书．

实用类型程序设计

屈延文　邱续欣 著

责任编辑　刘晓融

科学出版社出版

北京东黄城根北街 16 号

邮政编码：100707

中国科学院印刷厂印刷

新华书店北京发行所发行　各地新华书店经售

*

1992 年 8 月第　一　版　　开本：787×1092　1/16

1992 年 8 月第一次印刷　　印张：36 3/4

印数：1—2 000　　字数：860 000

ISBN 7-03-002854-6/TP·210

定价：32.20 元

序

本书不仅是大学软件专业高年级学生及研究生程序设计方法学课程的更新教材，而且是软件专业人员，特别是软件公司高级软件人员进行软件重用技术与自动生成技术培训的教材.

程序设计方法学的研究出现过几次高潮，并产生出几种流派. 例如"按功能划分模块,自上而下,逐步求精"的结构程序设计方法学;根据程序设计的逻辑语义产生的逻辑程序设计及由最弱前置条件理论产生的逻辑推导式的程序设计; 根据递归函数理论 (λ-演算理论)产生的函数程序设计. 尽管这些程序设计方法学风格不同,但都是基于自上而下的程序设计思想. 由 Simula-67 语言的 class 技术开始，后来在 Ada 语言的 package 技术及 C++ 语言的 class 技术上发展的面向对象的程序设计 (Object-Oreinted Programming, 简称 OOP) 技术在 80 年代得到了广泛的应用,直到 90 年代初,仍然是软件技术研究的大热点.

现在,我们要告诉读者: 程序设计方法学的研究到了一个新的阶段，即 OOP 技术与类型理论相结合的阶段. 我们认为，OOP 技术不与类型理论相结合，在 OOP 技术上提出的问题将不可能得到根本解决，而 OOP 的优势也将受到极大限制. 本书的宗旨即在于详细论述如何将类型理论与 OOP 技术结合起来用于软件重用及软件自动生成.

软件技术的发展从工程的角度来看,应划分成三个主要阶段: 软件重用,软件自动生成及软件智能化. 特别有意思的是，这三个主要发展阶段都与类型理论有着密切联系. OOP 技术与不同的类型理论相结合将会产生出不同的结果. 例如，OOP 技术与经典的类型理论相结合，其技术可以用于软件重用; OOP 技术与构造类型理论相结合，不仅可以用于软件重用，还可以发展软件自动生成技术; 把 OOP 技术与面向逻辑的类型理论相结合,将把软件引向智能化. 所以说,类型理论的研究是当前计算机研究的一个重点.

把 OOP 技术与构造类型理论联系起来,利用类型表达式，面向软件重用、软件自动生成与软件智能化三个方面而发展的新的程序设计方法学就是本书介绍的类型程序设计方法学 (Type Programming), 更严格地说,称为类型函数程序设计.

类型程序设计具有如下的五项基本原则:

- 按类型划分模块.
- 类型是其构造函数唯一确定的.
- 类型中函数(而不是过程)的定义是向所在类型构造函数的化约.
- 系统设计是类型构造(自下而上)与类型分解(自上而下); 类型构造与类型分解是由类型表达式表示的.
- 应用的开发就是类型的开发.

类型程序设计方法学的概念产生于 1986 年. 它是在研究了各种类型理论 (尤其是 Martin-Löf 类型论) 及 ADT 技术的基础上由本书第一作者提出的. 本书第一作者在

给软件专业的研究生讲授形式语义学并介绍各种类型理论的研究情况时就介绍过这种思想. 当时，他就明确指出了类型理论必定要与 Class-based 软件技术相结合的发展趋势，并根据 Martin-Löf 类型理论的数学思想，将类别代数理论及范畴论用构造主义数学思想进行改造，构成了被称之为“构造类别代数理论的数学方法”. 这一方法更容易为软件人员所接受. 也就是在这种思想基础之上，他提出了类型程序设计方法学.

软件产业最终会发展成为数学的工业. 将理论与实践结合起来是软件产业发展的关键问题. 我们应当提倡理论工作者面向实际，实际工作者多学些理论. 除此之外，我们应当组织一支强大的面向实际的理论工作者队伍. 面向实际的理论工作者与纯理论工作者的任务不同. 纯理论工作者的主要工作任务在于完善与发展理论体系本身. 而面向实际的理论工作者的任务则在于从实践中提出理论问题，并研究如何将理论应用于实际. 他们善于将实际问题转化成理论问题(也许理论问题的提出，在初级阶段还不能做到十分严格与完整，这种完善工作应由纯理论工作者去完成)，也善于发现什么样的理论研究工作最方便于指导实践，继而倡导与宣传这种理论. 正是他们将理论与实践携起手来，为高高在上的理论搭上扶梯供工程师们攀登. 这支力量是计算机软件事业最富有活力的一部分，没有这样一支队伍，计算机软件事业就不能得到长足的进步. 本书作者的工作就是面向实际的理论工作，虽然我们非常想把问题说明得非常严格，但我们仍然不准备拘泥于理论系统的严密，而着重研究如何用理论来解决实际问题.

在这种理论与方法指导下，我们还开发了一套软件系统，命名为 NCI/VAX C_ADT 软件重用与自动生成系统. 该系统是在 VAX/VMS 支持下开发的，具有三个版本序列. 这项工作是由本书第一作者领导的一个研究小组于 1985 年开始的. 其中最应当提到的是蔡林同志(现在美国伊里诺工学院读博士学位)与龚明同志(现在澳大利亚昆士兰大学读博士学位)，他们的最初工作是这项研究的很好的开端. 本书采用的全部实例程序都是在 VAX/VMS 操作系统支持下运行通过的，并全部是本研究小组人员编写的. 参加这项工作的有：王长青、邱续欣、张捷、刘环、文海明、朱美正，马鸿彬、朱小平、韦红及从秀芳等同志. 在开发系统时，还得到华北计算技术研究所小型机软件室的张如辰、南海两位同志的支持与帮助.

本书所反映的内容是机电部“七五”攻关项目课题的成果，这项研究也得到了国家“863”高技术计划的资助.

本书的第一、二、三、五、六、七章及附录由屈延文执笔，第四章大部分内容由邱续欣执笔，王长青为第 5.8 节执笔，马鸿彬为 5.7 节的部分内容执笔. 全书由屈延文统一审核定稿.

目　　录

引　言

——世界是类型的

《实用类型程序设计》首先向读者回答的一个问题是：程序设计为什么必须是类型的？不知有多少人提出过这个问题，这的确是必须回答的问题.

有人回答说(甚至相当权威的计算机科学家也是这样回答的)：

> “一个程序设计语言有了类型，便于编译程序进行静态语义检查(包括类型检查)，程序设计可以少犯错误；没有类型也能进行程序设计，只是容易犯错误，程序设计困难”.

按照这样的说法，有无类型仅仅是个好坏、难易问题，并不是问题的本质，充其量是方法论方面的问题.

我们却是这样回答这个问题的：

> “世界是类型的. 所有的科学都是类型的，或者说是范畴的. 物理学要分类型，化学要分类型，生物学要分类型，地质学也要分类型等等. 科学是分类的科学，研究是分类的研究，抽象是类型的抽象. 划分类型是科学研究的基本方法和途径，从方法论与世界观的统一原则来看，认识来源于客观世界，把事物分成类型首先是由于世界是类型的.”

读者可以设想，没有类型概念如何去做到精确的思考！与世界是类型的相对立的观点，认为世界是浑沌的. 我国古代哲学认为世界之初是浑沌的，但也进化成有类型概念的万物.

就计算机而言，现代计算机的指令系统就是类型的，例如布尔类型指令、定点指令、浮点指令、字符指令、I/O 指令等等，实际上都是有类型的，所有由软件实现的类型都是建立在这些基本类型之上的. 但我们早期的程序设计语言却没有显式地给出类型成分. 尽管程序设计语言没有给出显式类型说明的成分，但是在进行程序设计时，要保证正确设计，程序员头脑中也要分类型设计，甚至可以说，程序员头脑中如没有类型概念，他/她就不会思考.

在后来的程序设计语言中恢复了事物本身的类型概念，不管设计者的主观愿望如何，这样做决不是想当然的，因为这样做才把事物本身是类型的，程序员按类型思考与程序设计语言的表达也是类型的这三者统一起来. 难道我们能够认为“世界是类型的，思维是类型的，而表达语言是无类型的或无类型表达能力的”是正确的吗？

在后面的讨论中，我们会提到德国数学家 Martin-Löf 的类型论. 从哲学的观点来看，他认为(或者说我们可以归纳为)：

1. 世界是按类型构造的.

2. 类型是分层次的.

3. 低层次类型是较高层次类型的一个成员（member，或表示成 $U_{i-1} \in U_i$）.

所以说，类型论首先是一种宇宙观，并在这个宇宙观上发展成了直觉主义数学的一个新的学派. 从类型论观点来看，既然世界是类型的是一种宇宙观，就不是可有可无、是好是坏、是易是难的问题.

如果说在设计一个小程序、小系统时，还容易忽略类型的意义，那么在设计一个大系统时，若没有类型概念，则将无法进行. 首先我们来看一下操作系统的设计. VMS 操作系统尽管是用仅有几个计算机字长的存储类型的汇编程序设计语言设计的，但设计者们在组织操作系统时完全是按类型进行的. 很难设想没有类型概念能设计好 VMS 这样的操作系统!

对于类型概念的认识也是不断提高的. 最初的高级程序设计的类型概念是初步的. 后来发展的不同风格的高级程序设计语言，以及同一语言的改进版本，在类型概念上都更清楚、更丰富了. 在开发大型软件时，需要引入从 Simula-67 开始的 class 技术与概念，这个概念后来在 Ada 语言的 package 技术中得到完全的发展. class 或 package 是大型软件的组织技术，可以说程序员掌握 class 或 package 技术的应用应是最基本的技能之一，它是程序设计中模块化与层次化的实现技术(OOP 技术). 程序员没有掌握 class 或 package 技术，程序设计的模块化、层次化便是空洞的及原始的，因为所谓"按单功能划分模块"与"一般一个模块不多于 50 句"的设计原则是不可操作或很难操作的，程序设计的实践也证明了这一点.

那么，package 在 Ada 语言中是如何定义的? 一个 package 被定义为一个语言成分，一个把某些数据结构与这些数据结构相关的操作(函数与过程)组合在一个包中的语言成分. 有意思的是，语言的类型概念到了 Ada 语言时，也把数据结构及其操作统一在一个类型之中了(这也是一个进步). 于是，我们又发现，在一个语言中定义了两个把数据结构与操作统一在一个概念之中的语言成分：Type（类型）与 package（程序包）；它们虽有差别，但本质却一样. 从这个意义上看，package 概念完全可以由严格数学定义的类型概念所代替(有没有数学规定是 Type 与 package 的本质区别).

随后，抽象类型概念又被提出(又是一次进步)，而首先要实现的是为其扩充参数机制和数据类型的公理说明. 许多程序设计人员都知道函数有参数，过程有参数，却不知类型也有参数. 这实际上是代数的同构、同态概念，不只类型可以有参数，实际上许多概念都可以有参数(例如，小到生成式，大到操作系统).

软件重用是当前软件产业的主要技术，软件自动生成是在重用基础上，软件开发的更高层的技术追求. 为了实现软件自动生成，必须研究软件的构造技术. 软件的基本概念类型也必须是构造的. 类型的构造包括如下的几个概念：

1. 集合必须是构造的. 这包括集合对象是如何构造的及如何用已知集合构造集合.

2. 函数是如何构造定义的.

3. 证明是构造的与可执行的.

4. 类型的范畴是如何构造的.

5. 范畴的范畴是如何构造的(范畴层次性).

Martin-Löf 采用逻辑方法描述了类型的构造理论，而这是为许多程序人员所不熟悉

的. 能否把软件人员所熟悉的类型概念(代数概念)也变成构造的理论呢? 我们提出了构造类别代数理论,并在此基础上提出类型程序设计方法论.

有许多软件人员在为程序设计语言富足类型. 例如, 为 λ-演算(或 LISP 语言)增加类型概念;为项重写系统增加类型概念,研究类型归约计算机. 再例如, 为逻辑语言和面向对象的语言增加类型等等.

总之,世界是类型的,思维是类型的,表达也应是类型的. 在这种统一性的追求中,给我们带来的是高效益与事业的发展.

第一章 概 论

在这一章，我们将向读者介绍软件开发中概念性的知识与总体知识，目的是使读者从总体上把握住软件开发的要点，掌握解决问题的方法。

1.1 程序的正确性——首先追求的目标

本节，我们将介绍程序设计的逻辑基础，即程序正确性的基本概念。当然，我们假定读者已经具备了数理逻辑（特别是一阶谓词演算）的基本知识。

程序正确性，从哲学的角度看，我们认为其证明是如图 1.1.1 所示的三个世界的协调。

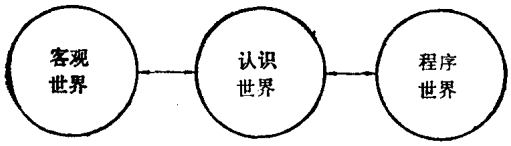


图 1.1.1

从认识论的角度来看，要想采用形式方法证明“认识世界”等价“客观世界”，也就是说，要证明我们对于客观世界的认识是完全正确的，当然是一个社会活动（社会过程）。但是，在假定认识是正确的之后，根据这个认识我们编出程序（如果是可编的话），那么这个程序是否完成了程序员想做的事情，却又是另一回事。所谓程序正确性就是指如下的意思：

“程序做的→程序员想做的”

按照软件工程的方法，从“认识世界”到“程序世界”的转换，首先要写说明（specification），然后根据这个说明去实现程序。一个程序说明主要任务在于说明“做什么”（What to do），而实现却主要解决“怎么做”（How to do）。描述“做什么”采用一阶谓词演算方法曾被认为是很理想的，一阶谓词演算是一个形式逻辑系统，具有一整套的证明技术。将一阶谓词演算与程序联系起来，实际上要解决如下的两个问题：

1. 将谓词演算转换成程序。

2. 将程序转换成谓词演算。

第一个问题，是所谓的软件自动生成问题，即如何从谓词演算推导出程序，这就是我们将在第四章详细向读者介绍的自上而下的推导式程序设计方法。而第二个问题则是程序验证问题。

验证一个程序的正确性时，会有如下三种情况：

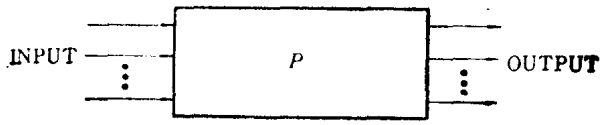
1. 程序系统↔程序员头脑中的外部系统。

2. 程序系统→程序员头脑中的外部系统.

3. 程序系统↛程序员头脑中的外部系统.

第一种情况，即程序系统完全与程序员头脑中的外部系统等价，则认为程序是正确的. 但是，程序完全反映了程序员头脑中的外部系统，并不认为程序员头脑中的系统与客观系统是等价的. 第二种情况，即程序系统蕴涵程序员头脑中的外部系统. 也就是说，程序做的事情比程序员要它做的事情多，但程序员要求做的，程序并无遗漏，此时，我们也认为程序是正确的. 第三种情况，即认为程序是错误的.

下面，我们比较形式地讨论一下程序正确性的概念.



(a)

	程序员的意图	程序执行
INPUT	Input Specification $\Phi(\bar{x})$	Input Conditions $\Phi(\bar{x})$
OUTPUT	Output Specification $\Psi_H(\bar{x},\bar{z})$	Output Results $P_H(\bar{x},\bar{z})$

(b)

图 1.1.2

图 1.1.2 中的 $\Phi(\bar{x})$ 为输入说明；$\Psi_H(\bar{x},\bar{z})$ 是程序员的主观目标，希望结果为输出说明. $\Phi(\bar{x})$ 及 $\Psi_H(\bar{x},\bar{z})$ 都是用谓词书写的. $P_H(\bar{x},\bar{z})$ 表示程序终止后，程序实际输出的结果，有时称 $P_H(\bar{x},\bar{z})$ 为执行谓词——程序做了什么.

设有一个程序 P，它的输入变量值的集合是 S_1，输出变量值的集合是 S_2，通过程序 P，如果程序是终止的，那么可以建立如图 1.1.3 的关系.

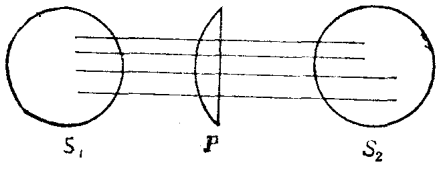


图 1.1.3

请注意，一个程序的执行停止时，才能建立这种映射. 如果一个程序的值域定义为：$D\bar{x} \times D\bar{y} = D_1$，这种映射在考虑了无定义⊥意义之后，应当是 $D_1^{\perp} \to D_1^{\perp}$. 为了以后讨论问题方便，将这个关系表示为

$$S_1\langle P\rangle S_2 \tag{1.1.1}$$

$$S_2 = S_1\langle P\rangle \tag{1.1.2}$$

其中式(1.1.1)的形式为“说明形式”，而式(1.1.2)的形式为“运算形式”，运算形式中的等式只在程序 P 执行停止时才有意义，或称 S_2 有定义，而一个集合可以表示成

$$S = \{\langle \bar{x},\bar{y}\rangle \mid p(\bar{x},\bar{y})\}$$

其中 $p(\bar{x},\bar{y})$ 为约束条件. 如果 S 不空,那么一定存在 $\bar{x},\bar{y}$ 使 $p(\bar{x},\bar{y})$ 为真.

下面,我们定义程序的正确性.

定义 1.1.1 如果对于每一个使 $\Phi(\bar{a})$ 为真的 $\bar{a}$, 程序 P 的计算是终止的,那么称程序 P 对于 $\Phi(\bar{x})$ 是终止的.

定义 1.1.2 如果对于每一个使 $\Phi(\bar{a})$ 为真并且使程序 P 计算终止的 $\bar{a}$, $\Psi(\bar{a}, p(\bar{a}))$ 为真,那么程序 P 对于 $\Phi(\bar{x})$ 及 $\Psi(\bar{x},\bar{z})$ 是部分正确的.

定义 1.1.3 如果对于每一个使 $\Phi(\bar{a})$ 为真的 $\bar{a}$, 程序 P 是终止的并且使 $\Psi(\bar{a}, p(\bar{a}))$ 为真,那么称程序 P 是完全正确的.

这里的 $\Phi(\bar{x})$ 表示输入谓词, $\Psi(\bar{x},\bar{z})$ 表示输出谓词. 所谓对于每一个使 $\Phi(\bar{a})$ 为真的 $\bar{a}$, 是说所有满足输入条件 $\Phi(\bar{x})$ 的每一个输入 $\bar{a}$.要仔细区别定义 1.1.2 与定义1.1.3之间的差别. 定义 1.1.2 谈到对于每一个使 $\Phi(\bar{a})$ 为真并且使程序 P 终止的 $\bar{a}$, 使 $\Psi(\bar{a}, p(\bar{a}))$ 为真(满足输出条件). 如仅在使 $\Phi(\bar{a})$ 为真的 $\bar{a}$ 上, $\Psi(\bar{a}, p(\bar{a}))$ 可能无结果(或说没有产生计算结果), 因为计算没有终止. 换句话说, 使 $\Phi(\bar{a})$ 为真的每一个 $\bar{a}$, 要么 $\Phi(\bar{a},p(\bar{a}))$ 为真, 要么程序就不终止(但凡终止的都是正确的), 所以叫部分正确. 搞清了部分正确概念,完全正确概念就容易了.

当程序执行到 i 点,此时程序状态的关系谓词,或称条件,可以用图 1.1.4 表示.

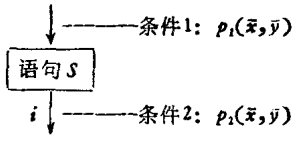


图 1.1.4

对于语句 S 而言, 条件 1: $p_1(\bar{x},\bar{y})$ 称为前置条件, 条件 2: $p_2(\bar{x},\bar{y})$ 称为后继条件. 显然,程序执行到 i 点时,假定程序变量状态为 $(\bar{x},\bar{y}) = (\bar{a},\bar{b})$, 那么将 $(\bar{a},\bar{b})$ 代入之后使之为真的谓词是可以有很多的. 这些在 i 点为真的谓词(或称条件)的强弱程度是不一样的. 下面,我们给出强化及弱化条件的两个过程:

1. 强化条件的过程

如果在程序 i 点的条件是如下的析取形式:

$$P = P_1 \vee P_2 \vee P_3 \vee \cdots \vee P_k \quad (k \geq 2)$$

企图寻找该析取形式中的任何子析取式并也使之在程序 i 点为真的过程称为强化条件的过程. 读者应当明白,并不是 P 的所有子析取式都在 i 点为真,在 i 点为真的 P 的子析取式也不是唯一的,也许有多个.

2. 弱化条件的过程

如果在程序 i 点的条件是如下的合取式:

$$P = P_1 \wedge P_2 \wedge P_3 \wedge \cdots \wedge P_k \quad (k \geq 2)$$

企图寻找该合取形式中的任何子合取式并也使之在程序 i 点为真的过程称为弱化条件的过程. 当然,如果 P 为真,那么 P 的所有子合取式也全为真. 如果 P 不为真, 但 P 的某些子合取式却可能为真.

显然在程序 i 点处存在着一个条件是最弱的. 如果前置条件是最弱的, 则称为最弱前置条件. 最弱前置条件是充分必要条件, 前置条件如不是最弱的, 那么它只是充分条

件.

在程序验证中还有一个谓词概念是必须谈到的,即不变式概念.

定义 1.1.4 如果对于每一个输入 $\bar{a}$ 使 $\Phi(\bar{a})$ 为真,并且程序无论什么时候执行到 i 点都有 $\bar{y}=\bar{b}$ 使得 $p_i(\bar{a},\bar{b})$ 为真,那么就说谓词 $p_1(\bar{x},\bar{y})$ 在断点 i 相对于 $\Phi(\bar{x})$ 是一不变式断言(简称不变式).

我们可以将程序正确性的讨论建立在不变式基础之上.

定理 1.1.1A 程序 P 是对 $\Phi(\bar{x})$ 终止的,当且仅当下式为真:

$$\forall P\forall\bar{x}\exists h\exists\bar{y}[P_h(\bar{x},\bar{y})]$$

定理 1.1.2A 程序 P 是对 $\Phi(\bar{x})$ 及 $\Psi_h(\bar{x},\bar{y})$ 部分正确的,当且仅当下式为真:

$$\exists P\forall\bar{x}\forall h\forall\bar{y}[P_h(\bar{x},\bar{y})\rightarrow\Psi_h(\bar{x},\bar{y})]$$

定理 1.1.3A 程序 P 是对 $\Phi(\bar{x})$ 及 $\Psi_h(\bar{x},\bar{y})$ 完全正确的,当且仅当下式为真:

$$\forall P\forall\bar{x}\exists h\exists\bar{y}[P_h(\bar{x},\bar{y})\times\Psi_h(\bar{x},\bar{y})]$$

以上定理是从程序正确性的正面描述的,相反地,如叙述一个程序不正确,也相应有如下三个定理.

定理 1.1.1B 程序 P 对 $\Phi(\bar{x})$ 是不终止的,当且仅当下式为真:

$$\exists P\exists\bar{x}\forall h\forall\bar{y}[\neg P_h(\bar{x},\bar{y})]$$

定理 1.1.2B 程序 P 对 $\Phi(\bar{x})$ 及 $\Psi_h(\bar{x},\bar{y})$ 不是部分正确的,当且仅当下式为真:

$$\forall P\exists\bar{x}\exists h\exists\bar{y}[P_h(\bar{x},\bar{y})\wedge\neg\Psi_n(\bar{x},\bar{y})]$$

定理 1.1.3B 程序 P 对 $\Phi(\bar{x})$ 及 $\Psi_h(\bar{x},\bar{y})$ 是不正确的,当且仅当下式为真:

$$\exists P\exists\bar{x}\forall h\forall\bar{y}[P_h(\bar{x},\bar{y})\rightarrow\neg\Psi_h(\bar{x},\bar{y})]$$

注意: 以上 $\forall\bar{x}$ 及 $\exists\bar{x}$ 应当理解为"对于每一个满足 $\Phi(\bar{x})$ 为真的 $\bar{x}$" 及"存在着一个满足 $\Phi(\bar{x})$ 为真的 $\bar{x}$". $P_h(\bar{x},\bar{y})$ 表示在 h 停机点上的程序执行产生的不变式. 这几个定理实际上与程序正确性概念的定义是一致的.

应当说明的是,我们在程序设计活动中,首先要注意的是停机问题(或说终止条件). 虽然,一般来说,程序的停机问题是不可以判定的,但是具体说来,每一个程序是否停机,只要程序员采用正确的方法是可以判定的(虽然给不出判定停机问题的一般算法).

在描述程序设计中的正确的逻辑现象方面,有两个系统是必须谈到的,第一个系统是由 C. A. R. Hoare 于 1969 年提出的程序逻辑的公理系统,这个公理化系统不研究程序终止性的部分正确的公理规则,因为在这个公理系统中,研究的是给定一个输入条件其正确的输出条件是什么,而不保证程序的终止性. 另一个系统是由 E. W. Dijkstra 提出的最弱前置条件的公理语义系统,这个公理系统以程序必须终止为前提,是程序完全正确的公理系统. 因为在这个公理系统中,研究的是给定一个假定为真的后继条件,其正确的前置条件是什么,推出的前置条件是最弱的.

下面,我们简单介绍一下 Hoare 的公理系统.

一个程序设计语言语句的语法定义如果是

$$\begin{aligned}
&s::=\text{skip}\mid x:=e\mid s_1;s_2\mid \underline{\text{if}}\,e\,\underline{\text{then}}\ s_1\ \underline{\text{else}}\ s_2\ \underline{\text{fi}}\mid\\
&\qquad\underline{\text{while}}\ e\ \underline{\text{do}}\ s\ \underline{\text{od}}\\
&e::=n\mid\text{true}\mid\text{false}\mid\underline{\text{not}}\ e\mid -e\mid(e)\mid e_1\ \text{op}\ e_2\\
&\text{op}::=+\mid-\mid*\mid/
\end{aligned}$$

其中 s 表示语句，e 表示表达式，x 是变量．那么这个程序的逻辑基础可以用如下的公理规则表示：

1.（空语句规则）

$$\vdash P\{\text{skip}\}P$$

2.（赋值语句规则）

$$\vdash P[e/x]\{x:=e\}P$$

3.（复合语句规则）

$$\frac{\vdash P\{s_1\}Q \quad \vdash Q\{s_2\}R}{\vdash P\{s_1;s_2\}R}$$

4.（条件语句规则）

$$\frac{\vdash P\wedge B\{s_1\}Q \quad \vdash P\wedge \neg B\{s_2\}Q}{\vdash P\{\underline{\text{if}}\ B\ \underline{\text{then}}\ s_1\ \underline{\text{else}}\ s_2\ \underline{\text{fi}}\}Q}$$

5.（迭代语句规则）

$$\frac{\vdash I\wedge B\{s\}I}{\vdash I\{\underline{\text{while}}\ B\ \underline{\text{do}}\ s\ \underline{\text{od}}\}(I\wedge\neg B)},$$

6.（后继规则）

$$\frac{\vdash P\to P_1 \quad \vdash P_1\{s\}P_2 \quad \vdash P_2\to Q}{\vdash P\{s\}Q}$$

以上符号中，P，P_1，P_2，Q，R 等是谓词，I 是不变式，对于形式

$$P\{s\}Q$$

P 称为前置条件谓词，Q 称为后继条件谓词．一种更适合于程序人员进行程序逻辑验证的方法由[17]提供，它是对程序的数据流进行逻辑分析．这种方法认为，程序中的基本现象是赋值、分支、聚合及循环迭代．关于如何学会验证程序的技术与方法，我们将在第三章介绍自上而下推导式程序设计时结合讨论．

程序设计中的错误，除了语法方面的外，语义方面的主要表现在“论域”与“辖域”两点上，逻辑方面的主要表现在“否定词”(NOT)的使用上；程序员在使用否定词时要特别小心．程序员的逻辑基础差，也常常表现在设计程序的分支结构中，各种判定条件组合混乱，甚至有的程序通路是不可到达的．作者希望通过本书的训练，程序人员能写一手清楚、简明的好程序．

1.2 算法效率决定程序效率

老的程序设计概念是突出一个“省”字，即省空间，省时间．在现代程序设计概念中，也讲程序占用空间和程序效率．程序效率主要由什么决定呢？我们说程序效率主要由算法效率决定．设计程序时，算法复杂性一旦确定，无论采用如何省的程序设计技巧，已经不能改变程序的基本效率了．衡量一个程序效率主要由算法复杂性决定．那么什么是算法复杂性呢？算法复杂性分空间复杂性与时间复杂性．在讨论算法的时间复杂性时，常常谈到“多项式时间复杂性”，“指数复杂性”．例如，一个算法的时间复杂性原来是 $O(n^3)$，如果能够找到一种办法，使该算法的时间复杂性变成 $O(n^2)$，这对一个算法工作者来说是