理论计算机科学

形式语义学引论

中国科学院计算技术研究所·周巢尘著

湖南科学技术出版社

理论计算机科学

形式语义学引论

中国科学院计算技术研究所 周巢尘著

责任编辑：周翰宗

*

湖南科学技术出版社出版
（长沙市展览馆路14号）
湖南省新华书店发行　湖南省新华印刷二厂印刷

*

1985年8月第1版第1次印刷
开本：850×1168毫米　1/32　印张：5.25　插页：4　字数：134,000
印数：1——5,700
统一书号：15204·146　定价：2.00元

前　言

1.什么是形式语义学.

形式语义学 (Formal Semantics) 是研究程序设计语言的语义的学问. 它以数学为工具，运用符号和公式，严格地解释程序设计语言的语义，使语义形式化，故称形式语义学.

程序设计语言是人类用来和计算机系统进行通信、并控制其工作的人工语言. 作为语言，人工语言和自然语言 (如汉语、英语等) 一样，有其语法(Syntax)、语义(Semantics)和语用(Pragmatics) 范畴. 程序设计语言的语法是指程序的组成规则；语义是指程序的含义；语用的所指，说法不一，大致包括程序的使用效果.

为了正确、有效地使用程序设计语言，必须了解语言中各个成份的含义，并且要求计算机系统执行这些成份所产生的效果和其含义完全一致. 但是程序设计语言的语义通常是由设计者用一种自然语言非形式地解释，实施者和使用者依据各自的理解实现和使用这种语言. 由于自然语言本身存在歧义现象，非形式的解释又不严格，故用这种方法解释语义容易造成设计者、用户和实施者对语义的不同理解，影响语言的正确实施和有效使用. 程序设计语言中的过程调用语句就是这方面的一个典型例子. 人们发现对过程调用语句的非形式的解释可能导致不同的理解，产生不同的效果.

人们对语义精确解释的要求产生了形式语义学，形式语义学的研究始于六十年代初，在程序设计语言ALGOL 60的设计中，第一次明确区分了语言的语法和语义，并使用巴科斯-瑙尔范式

(即BNF)成功地实现了语法的形式描述．语法的形式化大大刺激了语义形式化的研究，围绕ALGOL 60的语义出现了形式语义学早期的研究热潮．

美国斯坦福大学麦克阿瑟教授(J. Mc Carthy)在国际信息加工联合会一九六二年大会上作了著名的报告——“通往计算的数学科学”，系统地论述了程序设计语言语义形式化的重要性，和程序正确性、语言的正确实施等的关系，并提出在形式语义学研究中使用抽象语法和状态向量等基本方法．

二十多年来，形式语义学的研究取得了巨大进展．它对程序设计语言的设计、使用和实施产生了深刻的影响，语言的形式语法和形式语义（统称语言的形式定义）已经成为程序语言的必要组成部分．形式语义学也是软件工程学的基础理论之一．从形式语义学的观点看，软件工程学中的软件要求（Requirement）和软件描述（Specification）是在不同详尽程度上对程序语义的刻划，而软件正确性是讨论程序的语义和预期目标的一致；自动程序设计则是研究如何将程序的一种形式语义自动转换成另一种形式语义．

计算机科学界正在讨论未来的程序设计语言．泛函式程序设计语言（Functional Programming Language）和逻辑程序设计语言（Logical Programming Language）的研究，以及根据语言的语义定义自动生成语言编译系统的研究，正蓬勃开展．可以预计在新一代程序设计语言的设计中，语言的形式定义将先于并指导语言的具体设计和实施，形式语义学将发挥更大的作用．

通常的程序设计语言的语法是规定程序组成方法的一些规则，称为具体语法．但在定义程序的语义时，必须首先识别给定的程序，需要分析程序的语法结构．因此，在形式语义学中，使用一种讨论程序分解的语法规则，这种语法称作抽象语法．不同的程序设计语言往往使用不同的记号和表示方式，形式语义学提供的方法则应适用于一切程序设计语言，故抽象语法采用的记号和表示方式也是具体语法的一种抽象．

在定义程序设计语言的语义时，需要一种定义语义的语言，这种语言叫作元语言（Metalanguage）。元语言可以采用已有的数学语言，也可以是专门设计的语言。当然，为了用元语言去定义程序语言的形式语义，必须首先严格定义元语言的语义。

用程序设计语言编写的程序，规定了对计算机系统中数据的一个加工过程。形式语义学的基本方法是用一种元语言将程序加工数据的过程及其结果形式化，从而定义程序的语义。由于形式化中侧重面和使用的数学工具不同，形式语义学可分为四大类：

操作语义学（operational Semantics），着重模拟数据加工过程中计算机系统的操作；

指称语义学（Denotational Semantics），主要刻划数据加工的结果，而不是加工过程的细节；

代数语义学（Algebraic Semantics），可看作是指称语义学的一个分支，以使用代数学为特征；

公理语义学（Axiomatic Semantics），用公理化的方法描述程序对数据的加工。

2.本书的选材和组织

鉴于形式语义学在程序设计语言和软件工程学中的重要地位，故编著此书。希望此书成为迫切渴望知识更新的我国软件工作者学习形式语义学的入门书，也可作为大专院校计算机科系的教材和教学参考书。

形式语义学的内容极为广泛。国外的一些大学将每类语义学作为单独课程；甚至在讲授指称语义学时，还增设论域理论(Domain Theory)。但针对我国现况，编著一本入门书则为当务之急。当然，这不应该是一本通俗的科普书，只增加读者对一不熟悉领域的一般知识；或者是一本字典，罗列各种新名词，而不给予读者深入的了解。这本书应该介绍形式语义学的基本概念、基本理论和基本方法，使得掌握本书内容的读者易于深入到形式语义学的各个领域。

每类形式语义学都是建立在所用元语言的数学理论基础上

的。借助于元语言的理论，才可能严格定义程序设计语言的语义，论证语义的合理性，讨论不同语义定义的优劣，以及语义的其它特性。通过各种语义学途径定义各式各样程序语言成分的语义时，也需要探讨表述方法，以求表述的语义简洁易懂。由于语言成分花样繁多，表述方法也就十分繁杂。集各类语义学的理论基础与表述方法于一书，篇幅过于浩大。作为入门书，只介绍表述方法，不介绍基础理论，又会使读者误认为形式语义学只是一种符号化了的"非形式语义学"。对于我国的软件工作者来说，形式语义学的理论部分比之表述方法恐怕更为陌生。故本书的前三章以一个简单的程序设计语言为例，介绍操作语义学、指称语义学和公理语义学的基本理论和方法，并侧重于基本理论。对其中的概念和事实都以必要的严格形式陈述和剖析，望初学者认真阅读。第四、五、六章讨论过程调用语句，非确定型和并发型程序设计语言的语义，介绍这些语言现象引起的语义学理论和方法的发展。这些方面的语义学研究远远没有完结，非确定性和并发性的研究是当前形式语义学研究的前沿领域。第七章介绍时态语义。时态逻辑（Temporal Logic)用于语义学的研究开始于七十年代后期，近年来蓬勃发展，已成为公理语义学的重要分支。希望这后四章能为有志于深入学习形式语义学或研究形式语义学的读者打下基础。

作为一本入门书，本书中不讨论语义学中待解决的问题，也不涉及语义学的应用。本书没有提供习题，但在各章内容中（尤其在后四章中）提出了若干问题或者留下若干待证的结论。有兴趣的读者可自己动手补足这些细节，这对于理解形式语义学的理论和方法是很有帮助的。

由于著者对代数语义学并不熟悉，本书中只好略去这一内容。其它缺陷和谬误也在所难免，望各界不吝指教。

目　录

第1章

操作语义学

§1.1 引　言

程序设计语言的实施者是按照语言的语义在具体的计算机系统中编制语言的解释程序或者编译程序的．但反过来看，语言在任何计算机系统中的任何一种实施一旦完成，那末对这个计算机系统而言，语言的含义也就完全确定了；也就是说，语言的一种实施可看作是语言语义的一种定义．用语言的实施作为语言的语义定义，也就是将语言成分所对应的计算机系统的操作作为语言成分的语义，故称之为操作语义．当然，语言的语义应该是标准的，不应依附于一个特定的计算机系统，一种具体的实施．因此操作语义学中使用抽象的机器和抽象的解释程序来定义语言的语义．

操作语义学的基本思想来源于程序设计语言的实施．但第一次系统地、严格地陈述这一途径的是英国学者兰丁(P. J. Landin)他于一九六四年一月发表了“表达式的机械化求值”一文．文中使用“栈-环境-控制-外贮”抽象机器（简称SECD机器），定义了表达式的操作语义．

IBM公司维也纳实验室六十年代在研究程序设计语言PL/I的形式定义时，提出了描述操作语义的一种元语言——维也纳定义语言(简称VDL)．一九七四年欧洲计算机制造商联合会(ECMA)和美国国家标准局（ASI）正式建议使用VDL定义的PL/I 的语义作为PL/I的一种标准．

一九八〇年前后，英国爱丁堡大学的计算机科学家们提出了结构式操作语义学．它在更一般的数学结构（不必是抽象机器）上用数学的归约关系（Reduction Relation）建立语义的解释系统．这种语义学具有结构式特征，也就是语言中复合成分的语义是由其子成分的语义复合而成．结构式的语义定义可为软件工程学中的结构式程序设计方法提供重要原理．

本章中将以一个简单程序设计语言FLOW为例，介绍兰丁式的抽象机语义定义和结构式的操作语义定义．FLOW语言也将用于以后几章中．

§1.2 FLOW语言

为了侧重讨论程序语言的一些基本控制结构的语义，FLOW语言中不给出原子语句（如赋值语句）和布尔表达式的语法定义，视作待定参数．

FLOW的语法：

(1) 原子语句集Asts，用A表示其中元素；

(2) 布尔表达式集Bexp，用B表示其中元素；

(3) 语句集Sts，用S表示其中元素，定义如下：

$$S ::= \textbf{skip} \mid A \mid S;\ S \mid (\textbf{if B then S else S}) \mid (\textbf{while B do S}).$$

Asts和Bexp是FLOW的两个参数，这里未作详细规定．但是作为原子语句和布尔表达式，自然要求从语法上就能和FLOW中的其他部分相区别，这一要求在以后的讨论中还会用到．一旦给定这两个参数，就可得到一个特定的语言．

例如，令

X——变元集，以x表示其中元素；

N——非负整数集，以n表示其中元素；

Aexp——算术表达式集，以a表示其中元素，定义如下：

$$a ::= n \mid x \mid +(a,\ a) \mid -(a,a) \mid *(a,a),$$

这里我们采用前缀式运算符，如 $-(a_1, a_2)$ 表示a_1减去a_2。

这样，Asts可定义为：

$$A::=a=>x$$

即将变元x赋以表达式a。Bexp定义为：

$$B::=tt|ff|=(a,a)|<(a,a)|\wedge(B,B)|\neg B,$$

这里我们也使用前缀式运算符，如$<(a_1, a_2)$表示表达式a_1的值小于表达式a_2的值。

这一语言记作为AFLOW。

§1.3 栈-状态-控制机器

本节中我们定义一个抽象机器来解释执行FLOW的程序。为了解释执行FLOW程序，需要三个工作区：

(1) 存贮程序的控制区，以c表示；

(2) 记载程序变元当前值的机制，称为状态，记作s，全体状态的集合记为State；

(3) 记载中间结果的栈区，记作st。

三元组 (st, s, c) 的一种取值构成解释执行FLOW程序的抽象机器的一个状态，称作为大状态(Grand State)，和s以资区别，记作gs。抽象机的动作是由大状态间的转移规则(Transition Rule)规定的，转移规则形为

$$(st, s, c)\Rightarrow(st', s', c')$$

规定抽象机可从大状态 (st, s, c) 转移至大状态(st', s', c')。转移规则也可以是一个模式，它的每个实例规定了抽象机大状态的一个可能的转移。

下面我们就用转移规则模式来定义解释执行FLOW程序的抽象机的行为：

A，分析

(1) (st, s, (if B then S_1 else S_2)c)

$$\Rightarrow(S_2:S_1:st,\ s,\ B\ \text{if}\ c),$$

这条规则是说，当抽象机执行到条件语句时，先把分语句存入栈区，其中“:”表示分割符；然后准备求出布尔表达式B的值，即将B保存在控制区的首部；在求出B的值后，再执行该条件语句，即把条件语句标志保存在控制区的B 的后面．而栈区、状态区和控制区的其它部分不变。

(2) (st, s, (while B do S)c)

$$\Rightarrow(\text{skip}:S;\ (\text{while B do S}):st,s,B\ \text{whill c}),$$

这条转移规则和分析条件语句的规则很相似，只是在栈区中已形成了当B取真值时转往执行的语句 S; (while B do S)，即先执行S，再执行此循环语句；以及当B取假值时转往执行的语句skip，即跳过此语句，准备顺序执行下条语句。

B，求值

由于FLOW语言中布尔表达式是一种参数，没有详尽的定义，因此在建立布尔表达式求值的转移规则时，必须预先假定布尔表达式求值的语义函数$\mathscr{B}$．$\mathscr{B}$规定了任一布尔表达式的语义，即$\mathscr{B}$是布尔表达式至其语义的一个函数．而布尔表达式的语义又可看作是程序变元当前值 (即状态s 的值) 至布尔值的一个函数．令T = {tt, ff}，其中tt表示真值，ff表示假值．则

$$\mathscr{B}\in(\text{Bexp}\to(\text{State}\to T)),$$

亦写作

$$\mathscr{B}:\ \text{Bexp}\to(\text{State}\to T),$$

这里 (L→R) 表示由集合L至集合R的全体函数组成的函数空间。在语义学中，通常采用花体字表示语言成分的语义函数，而将语言成分用双重括号括起，故给定一个布尔表达式B，有

$$\mathscr{B}[\![B]\!]:\ \text{State}\to T$$

对给定的状态s，又有

$$\mathscr{B}[\![B]\!](s):T$$

即对给定的布尔表达式B和状态s，由$\mathscr{B}$可得到B在状态s下所取的布尔值。

以上这些记号法在本书中将不断使用，不再一一说明．使用

语义函数$\mathscr{B}$及上述记号布尔表达式求值的转移规则可定义如下：

$$(st,\ s,\ Bc)\Rightarrow(\mathscr{B}[\![B]\!](s):st,\ s,\ c),$$

即抽象机按照$\mathscr{B}$对给定B及s求出布尔值$\mathscr{B}[\![B]\!](s)$，并将其存入栈区。

由于FLOW语言中没有算术表达式，故算术表达式的求值规则不必在此讨论。

C，执行

下列各规则中c = ；c′或者c = c′ = ϕ（ϕ表示空序列，在不致引起混淆时，本书中亦用来表示空集合）。

$$(1)\ (st,\ s,\ \textbf{skip}\ c)\Rightarrow(st,s,\ c'),$$

即执行**skip**不导致st和s的改变，但导致抽象机转向执行其后继部分c′。当c中内容非空时，c应以顺序算子“；”为起始跟以后继部分c′，故c = ；c′；当c中内容为空时，其后继部分亦为空，故c = c′ = ϕ。

$$(2)\ (st,s,Ac)\Rightarrow(st,\mathscr{A}[\![A]\!](s),c'),$$

这里$\mathscr{A}$代表原子语句的语义函数，如$\mathscr{B}$一样，此处作为预先假定的。原子语句设想为赋值语句，它按照程序变元的当前值求出表达式值，再改变被赋程序变元的值，故假定其语义为状态空间至状态空间的一个函数。即

$$\mathscr{A}:\ Asts\rightarrow(State\rightarrow State)$$

$$(3)\ (tt:S_2:S_1:st,\ s,\ \textbf{if}\ c)\Rightarrow(st,s,S_1c).$$

$$(4)\ (ff:S_2:S_1:st,s,\textbf{if}\ c)\Rightarrow(st,s,S_2c).$$

这两条规则是说，在求出布尔表达式值后条件语句的执行导致抽象机选择相应的分语句，并转向执行该分语句。对于循环语句亦有两条类似的规则。

$$(5)\ (tt:S_2:S_1:st,\ s,\ \textbf{while}\ c)\Rightarrow(st,\ s,\ S_1c)\ .$$

$$(6)\ (ff:S_2:S_1:st,\ s,\ \textbf{while}\ c)\Rightarrow(st,\ s,\ S_2\ c)\ .$$

至此，我们定义了解释执行FLOW程序的一个抽象机，也就给出了FLOW语言的一种操作语义．下面以AFLOW中的一个程序为例，在假定$\mathscr{A}$，$\mathscr{B}$和通常理解一致的前提下，看其操作语义是否与通常理解亦一致，

例．计算$x!$的程序FAC．

$1\Rightarrow y;\ x\Rightarrow z;\ (\text{while}\ \neg=(z,0)\ \text{do}\ *(y,\ z)\Rightarrow y;\ -(z,1)\Rightarrow y)$．

FAC中只有程序变元x，y和z，设State为非负整数三元组的集合．在x，y和z的初始值分别为2，0，0时解释执行FAC程序形成的大状态转移序列如下，其中转移号“⇒”上标以此转移所依据的规则号，而用到的$\mathscr{A}$、$\mathscr{B}$函数假设为：

$$\mathscr{B}[\![\neg=(z,0)]\!](x,y,z)=\begin{cases}\text{tt}, & z\neq 0,\\ \text{ff}, & z=0,\end{cases}$$

$$\mathscr{A}[\![1\Rightarrow y]\!](x,\ y,\ z)=(x,\ 1,\ z),$$
$$\mathscr{A}[\![x\Rightarrow z]\!](x,\ y,\ z)=(x,\ y,\ x),$$
$$\mathscr{A}[\![*(y,z)\Rightarrow y]\!](x,y,z)=(x,y*z,z),$$
$$\mathscr{A}[\![-(z,1)\Rightarrow z]\!](x,y,z)=(x,y,z\dot{-}1),$$

这里“$\dot{-}$”为非负减，即

$$n\dot{-}m=\begin{cases}n-m, & n\geqslant m,\\ 0, & n<m.\end{cases}$$

$(\varnothing,\ s_0,\ \text{FAC})$　　$s_0=(2,\ 0,\ 0)$

$\xRightarrow{C,(2)}(\varnothing,\ \mathscr{A}[\![1\Rightarrow y]\!](s_0),\ S_1)$　　设$\text{FAC}=1\Rightarrow y;S_1$

$\xRightarrow{C,(2)}(\varnothing,\ \mathscr{A}[\![x\Rightarrow z]\!](s_1),\ S_2)$　　$s_1=(2,1,0)$，

$\mathscr{A}[\![1\Rightarrow y]\!](s_0)=s_1$．

设$S_1=x\Rightarrow z;\ S_2$

$\xRightarrow{A,(2)}(\text{skip}:S_3;S_2,s_2,\neg=(z,0)\text{while})$　　$s_2=(2,1,2)$，

$\mathscr{A}[\![x\Rightarrow z]\!](s_1)=s_2$．

设$S_2=(\text{while}\neg=(z,0)\ \text{do}\ S_3)$

$$\overset{B,(1)}{\Longrightarrow}(\mathscr{B}[\![\neg=(z,0)]\!](s_2):\mathbf{skip}:S_3;S_2,\ s_2,\ \mathbf{while})$$

$$\overset{C,(5)}{\Longrightarrow}(\varnothing,\ s_2,\ S_3;\ S_2) \qquad \mathscr{B}[\![\neg=(z,0)]\!](s_2)=\mathrm{tt}$$

$$\overset{C,(2)}{\Longrightarrow}(\varnothing,\mathscr{A}[\![*(y,z)\Rightarrow z]\!](s_2),S_4;S_2)$$

设$S_3=*(y,z)\Rightarrow z;\ S_4$

$$\overset{C,(2)}{\Longrightarrow}(\varnothing,\mathscr{A}[\![-(z,1)\Rightarrow z]\!](s_3),\ S_2) \qquad s_3=(2,\ 2,\ 2),$$

$$\mathscr{A}[\![*(y,z)\Rightarrow z]\!](s_2)=s_3$$

$$\overset{A,(2)}{\Longrightarrow}(\mathbf{skip}:S_3;S_2,s_4,\neg=(z,0)\mathbf{while}) \qquad s_4=(2,\ 2,\ 1),$$

$$\mathscr{A}[\![-(z,1)\Rightarrow z]\!](s_3)=s_4$$

……　　省略若干次转移

$$\overset{A,(2)}{\Longrightarrow}(\mathbf{skip}:S_3;S_2,s_5,\neg=(z,0)\ \mathbf{while}) \qquad s_5=(2,\ 2,\ 0)$$

$$\overset{B,(1)}{\Longrightarrow}(\mathscr{B}[\![\neg=(z,0)]\!](s_5):\mathbf{skip}:S_3;\ S_2,\ s_5,\ \mathbf{while})$$

$$\overset{C,(6)}{\Longrightarrow}(\varnothing,\ s_5,\ \mathbf{skip}) \qquad \mathscr{B}[\![\neg=(z,0)]\!](s_5)=\mathrm{ff}$$

$$\overset{C,(1)}{\Longrightarrow}(\varnothing,\ s_5,\ \varnothing)$$

上述转移序列表明，只要$\mathscr{A}$，$\mathscr{B}$符合通常的理解，在x的初值为2时执行FAC，程序可以终止，终止时变元y处存有所求的阶乘值2。

§1·4，程序的计算

FAC程序是用来计算$x!$的，更一般地讲，上例中的状态s_5可

看作是程序FAC作用于初始状态s_0上的计算结果。程序所完成的计算是程序用户至为关心的，是鉴别程序语义定义是否合理的主要因素。

FLOW语言是确定型语言，FLOW程序所完成的计算也应该是确定的。

设 P(l, r)是一个二元关系，用P表示满足此关系的元素对的集合，即 $P=_{df}\{(l,r)\mid P(l,r)\}$。在不致引起混淆时，关系P(l,r)和集合P将不加区别。

定义。二元关系P称为关于第一个变元是确定的，当且仅当

$$\forall l, r_1, r_2.\ (P(l,r_1)\wedge P(l,r_2))\supset(r_1=r_2).$$

定理。转移关系“⇒”关于左边的大状态是确定的。

证明。需证明对任意大状态gs，gs_i (i = 1, 2) 若满足

$$gs \Rightarrow gs_i \quad (i=1,\ 2),$$

则有

$$gs_1 = gs_2.$$

考察全部转移规则后，使用原子语句以及布尔表达式和其它语言成分在语法上是可相区别的假设，可以证明任一个大状态至多是一个转移规则的一个左方大状态模式的一个实例，故其可能转移到的大状态也至多有一个。

定义。可达到性 (Reachability)。

大状态gs′称为可由大状态gs经转移关系“⇒”达到的，记作

$$gs\Rightarrow^{*}gs'$$

当且仅当 (1) gs = gs′ 或 (2) 存在gs″，使得

$$gs\Rightarrow^{*}gs''$$

且

$$gs''\Rightarrow gs'.$$

“$\Rightarrow^{*}$”不是确定型关系，但由一个大状态出发所能达到的“终极”大状态应该是唯一的。这一事实可由著名的丘奇(Church)-罗叟 (Rosser) 性质所推断。

设P是一个二元自反、递传关系。

定义。P称作是具有丘奇-罗叟性质的，当且仅当

$$\forall x,y_1,y_2\exists z.\ (P(x,y_1)\wedge P(x,y_2))$$
$$\supset(P(y_1,z)\wedge P(y_2,z)).$$

推论. 可达到关系“$\Rightarrow^*$”是具有丘奇-罗叟性质的.

证明. 可达到性是一种自反、传递关系,可达到性的丘奇-罗叟性质弱于转移关系的确定性. 故由上述定理可以推出此结论,有兴趣的读者不妨一试.

对任意语句S,定义状态空间State至其自身的偏函数com(S):

com(S)(s) = s′ 当且仅当 $(\varnothing,s,S)\Rightarrow^*(\varnothing,s',\varnothing)$. 由于$(\varnothing,s',\varnothing)$是一类“终极”大状态,也就是不可能再转移的大状态. 这样,可达到性的丘奇-罗叟性质保证了这一定义的合理性. 形如$(\varnothing,s,S)$的大状态称为初始大状态,而$(\varnothing,s,\varnothing)$形的大状态叫作终止大状态.

从一个初始大状态出发不一定总能达到终止大状态的;如令S: **(while B do skip)**, s满足$\mathscr{B}[\![B]\!](s)=tt$,则由$(\varnothing,s,S)$出发将永不终止(这种现象称之为发散). 故

$$com:\ Sts\longrightarrow(State\longrightarrow' State)$$

这里“$\longrightarrow'$”表示偏函数(即部分映射).

还可以证明由初始大状态出发不会达到非正常终极,即所能达到的“终极”大状态都是终止大状态;也不会产生既能发散又能终止的非确定现象(注意:这类非确定现象是不能由丘奇-罗叟性质排斥的).

由于缺乏“标准”语义,一种语义的“合理性”是不能严格证明的. 但可以通过推断所定义的语义的特性来鉴别它的合理程度(当然证明和其它语义的等价;就象证明各种可计算性定义的等价性一样)是论证语义“合理性”的更为重要的手段. 我们可以证明

(1) $com(S_1;\ S_2)=com(S_2)\circ com(S_1)$,

(2) $com((\textbf{if B then } S_1 \textbf{ else } S_2))=cond(\mathscr{B}[\![B]\!],com(S_1),com(S_2))$,

(3) $com((\textbf{while B do } S))=cond(\mathscr{B}[\![B]\!],com(S;\ (\textbf{while B do } S)),id)$