基于轨迹的程序语义之一:轨迹与语义对象*

王岩冰 陆汝占

(上海交通大学计算机系 上海 200030)

摘要 本文提出一种基于轨迹的指称语义框架,该框架结合了操作语义和代数语义的特征,避免使用专门的数学理论,将静态语义和动态语义结合在一起统一处理.本文及其续篇将通过一个中等规模的过程式模型语言来说明上述语义框架更适合描述真正的程序设计语言.本文首先引入轨迹概念和模型语言,然后讨论该语言的各句法成分所对应的语义论域,其中没有使用含有函数空间构造运算的递归论域方程.

关键词 轨迹,类 ALGOL 语言,抽象语法,基调代数,语义论域.

中图法分类号 TP301

在程序设计语言的规约说明中,形式语法已得到广泛应用和普遍接受,但形式语义定义通常被认为是晦涩深奥的,只能被少数专家理解.因此,在程序设计语言的设计、实现和使用中很少得到实际运用.造成这种情况的部分原因是形式语义的书写风格不够清晰,有待改进,但更深刻的原因是高级语言的抽象度较高,难以用操作语义直接刻画,而指称语义学中的概念与人们通常从执行过程角度去理解程序的方式相去甚远.另外,指称语义学建立在艰深的论域理论之上,使得其应用范围受到很大限制.

针对这种情况,P. Mosses 提出了称为 Action Semantics 的语义描述框架.它是一种转换语义方法,有一个称为 Action Notation 的元语言作为中间语言.Action Notation 有一个内核,所有的语言成分都可以转换到内核中的语言成分,对内核的描述采用操作语义.这样,高级语言的语义描述分为两个层次:第 1 层是用 Action Notation 描述对象语言的语义,第 2 层是对 Action Notation 作语义描述.Action Semantics 结合了指称语义、代数语义和操作语义的特征,它不使用论域理论,具有良好的模块性,是一个值得重视的研究方向.[1]由于转换语义本质上是操作语义,因此,Action Semantics 也有概括性差、难以用数学方法处理等操作语义固有的一些缺陷.在不少计算机科学家的心目中,指称语义是一种标准的语义描述方法,形式语义描述的理想情况是同时给出模型语言的操作语义和指称语义,并证明二者的等价性.[2]因此,如何在指称语义的框架下克服传统框架的缺陷,使它适合于处理真正的程序设计语言,就成为一个很有意义的研究课题.

为此,我们提出一种基于轨迹的指称语义框架.本文第 1 节引入轨迹概念,第 2 节给出一个模型语言 L_1,第 3 节讨论该语言的各句法成分所对应的语义论域,在本文的续篇中将详细讨论语义函数的定义.

1 轨迹与轨迹函数

传统的指称语义定义不象操作语义那样涉及语言成分的执行过程,而只考虑各成分执行的最终效果,这种对时序特征的省略,对于简单的模型语言来说,具有简捷和抽象度高的优点,但是,这并不适合处理实际的程序设计语言,究其原因有以下几点:①对于在软件中占大多数的交互式程序和实时程序来说,时序特征是语义刻画的重要内容,这里并不一定存在一个最终的执行状态,而且程序的执行结果依赖于人工指令或自然信号的输入时刻.②即使对批处理程序,时空复杂度特性也是一个不应忽略的语义描述内容.传统的指称语义框架不但造成语义理论与算法理论的脱节,也大大减弱了语义表达能力,从而缩小了语义规约的应用范围.③省略时序特征难以反映程序设计语言的控制流结构,不符合大多数人从执行过程角度理解程序的习惯,使得指称语义描述缺乏直观性,难写难读.为此,我们引入轨迹的概念,并用从输入到状态轨迹的函数代替传统的输入输出函数作为程序的指称,从而在指称语义框架中引入操作语

* 本文研究得到国家自然科学基金和南京大学计算机软件新技术国家重点实验室基金资助.作者王岩冰,1966 年生,博士生,讲师,主要研究领域为程序设计语言,形式规约与验证.陆汝占,1940 年生,教授,博士导师,主要研究领域为自动推理与定理证明,语义模型与软件集成,汉语语义计算.

本文通讯联系人,王岩冰,上海 200030,上海交通大学计算机系

本文 1996-12-17 收到原稿,1997-05-26 收到修改稿

© 中国科学院软件研究所 http://www.jos.org.cn

的一些特征,以便体现程序的执行过程.本节讨论轨迹的形式定义,首先对本文所用的术语作一些约定,这些记法可参见文献[3,4].

ω 表示自然数集合,$A\to B$ 表示全体从 A 到 B 且定义域为有限集的部分函数集合.有限函数 $f=\{(x_1,A_1),\ldots,(x_n,A_n)\}$ 写作 $f=\{x_1\mapsto A_1;\ldots;x_n\mapsto A_n\}$,其中 $y_1\mapsto A;y_2\mapsto A;\ldots;y_m\mapsto A$ 可简写为 $y_1,y_2,\ldots,y_m\mapsto A$,$f=\{x\mapsto A_1;\ldots;x_n\mapsto A_n\}$ 的和 Σf 用 $x_1A_1+\ldots+x_nA_n$ 表示,对于 $y\in\Sigma f$,$P_{2,1}(y)=x_i$,也写作 $y\in x_i$,(x_i,a) 也写为 $x_i(a)$.在不引起混淆的情况下,y 的标志可以省略,即用 y 作为 $P_{2,2}(y)$ 的简写.对 $x\in A^*$,$len(x)$ 表示 x 的长度,$sub(x)$ 表示正整数区间 $[1,len(x)]$(当 $x=\varepsilon$ 时为 $\varnothing$),对 $x\in A^+$,设 $x=a_1a_2a_n\ldots(n\geqslant 1)$,$tl(x)\triangleq a_n$,$hd(x)\triangleq a_1a_{n-1}\ldots$(当 $n=1$ 时为 ε).B,C,Z 分别表示布尔值、字符和整数集合,S 表示标识符集合.$A_\perp\triangleq A\cup\{\perp\}$.

定义 1.1. 设 DE 为动态错误标志的非空有限集,并规定 $norm\notin DE$,$loop\notin DE$,A 为非空状态集.$A^n_{DE}\triangleq\{norm\}\times A\cup DE\times A^+\cup\{loop\}\times A^+$,称为 A 关于 DE 的轨迹集.对 $x\in A^n_{DE}$,$norm(x)\Leftrightarrow P_{2,1}(x)=norm$,$derr(x)\Leftrightarrow P_{2,1}(x)DE$,$loop(x)\Leftrightarrow P_{2,1}(x)=loop$.在没有特别指定 DE 的前提下,A^n_{DE} 常简写为 A^n.

轨迹是对程序执行过程的刻画,这一执行过程由一连串状态变化为特征,在没有外界干涉的情况下可能永不停止,终止的执行过程又分为正常终止和异常终止两种情况,文献[2]采用不带异常终止的轨迹概念讨论了各种语义现象.

定义 1.2. 给定 DE 和 A,$A^L_{DE}\triangleq normA+err(DE\cup\{loop\})$,$Last:A^n_{DE}\to A^L_{DE}$,对 $x\in A^n_{DE}$,

$$Last(x)\triangleq \textbf{if } norm(x) \textbf{ then } \langle norm,tl(P_{2,2}(x))\rangle \textbf{ else } \langle err,P_{2,1}(x)\rangle \textbf{ fi}$$

A^L_{DE} 代表 A 关于 DE 的终止状态集.Last 函数用来表示一个轨迹的终止状态.

定义 1.3. 给定 A 和 DE,$Append:A^n_{DE}\times A^n_{DE}\to A^n_{DE}$.对 $x,y\in A^L_{DE}$,

$$Append(x,y)\triangleq \textbf{if } \neg(norm(x) \textbf{ then } x \textbf{ else } \langle P_{2,1}(y),P_{2,2}(x)^*P_{2,2}(y)\rangle \textbf{ fi}$$

Append 函数给出两个轨迹的并置,非终止序列和异常终止序列无法与其它轨迹正常并置,前者达不到第 2 个轨迹,后者要保持出错的确切位置,因此,它们并置上另一轨迹后保持不变,正常终止序列与另一轨迹的并置其终止特征与后者相同.

定义 1.4. 给定 A 和 DE,A 到 A^n_{DE} 的函数称为 A 的轨迹函数,$I_A:A\to A^n_{DE}$,对 $x\in A$,$I_A\triangleq\langle norm,x\rangle$;$Trans:(A\to A^n_{DE})\to(A\to A^L_{DE})$,对 $f:A\to A^n_{DE}$,$Trans(f)\triangleq Last^*f$.

轨迹函数可以用来作为程序指令的指称,Trans 函数用来说明从轨迹语义到输入输出语义的转换.

定义 1.5. $Extend:(A\to A^n_{DE})\to(A^n_{DE}\to A^n_{DE})$,对 $x\in A^n_{DE}$,

$$Extend(x)\triangleq \textbf{if } \neg norm(x) \textbf{ then } x \textbf{ else } Append(x,f(tl(P_{2,2}(x)))) \textbf{ fi}$$

定义 1.6. $Con:(A\to A^n_{DE})\times(A\to A^n_{DE})\to(A\to A^n_{DE})$,对 $f_1,f_2:A\to A^n_{DE}$,$Con(f_1,f_2)\triangleq Extend(f_2)^*f_1$.

合成算子用来刻画指令的顺序合成.

2 模型语言

本节给出类 ALGOL 语言的一个模型语言 L_1,作为过程语言的基本模型.L_1 不包含并发、模块、类属、异常处理等大型语言的特征,也不包含指针,但 L_1 包含了语句和表达式的块结构、记录和数值类型、动态数组,可以联立递归的函数和过程定义,子程序可带有常量参数、变量参数和可变长数组参数,表达式的功能尤其强大.总之,L_1 是一个中等规模的模型语言,可以体现过程式语言的主要特征,下面使用的抽象语法格式可参考文献[3],L 的设计特征可参考文献[4,5].

(1) $program\triangleq command^*$

(2) $command\triangleq\Sigma\{assign\mapsto vepair;\ pcall\mapsto procall;\ ifcd\mapsto ifstat;\ whilecd\mapsto whilestat;\ declcd\mapsto declstat\}$

(3) $vepair\triangleq\Pi\{v\mapsto vname;\ e\mapsto expression\}$

(4) $procall\triangleq\Pi\{id\mapsto S;\ aconp\mapsto expression^*;\ avarp\mapsto vname^*\}$

(5) $ifstat\triangleq\Pi\{test\mapsto expression;\ thenb,elseb\mapsto command^*\}$

(6) $whilestat\triangleq\Pi\{test\mapsto expression;\ body\mapsto command^*\}$

(7) $declstat\triangleq\Pi\{decl\mapsto decalaration^*;\ body\mapsto command^*\}$

(8) $vname\triangleq\Sigma\{vn\mapsto S;\ rv\mapsto rvar;\ av\mapsto avar\}$

(9) $rvar\triangleq\Pi\{body\mapsto vname;\ pt\mapsto S\}$

(10) $expression\triangleq\Sigma\{char\mapsto Charliterc1;\ int\mapsto Intliteral;\ cn,lookin,arrayl,arrayh\mapsto S;$
$varval\mapsto vname;\ fcall\mapsto funcall;\ compare,apart\mapsto epaire;\ rpart\mapsto sepair;$
$rexpn\mapsto sepair^*;\ aexpn\mapsto expression^*;\ ifep\mapsto ifexpn;\ declep\mapsto declexpn\}$

© 中国科学院软件研究所 http://www.jos.org.cn

(11) $funcall \triangleq \Pi\{id \mapsto S;\ ap \mapsto expression^*\}$
(12) $epair \triangleq \Pi\{e_1, e_2 \mapsto expression\}$
(13) $sepair \triangleq \Pi\{id \mapsto S;\ e \mapsto expression\}$
(14) $ifexpn \triangleq \Pi\{test, thenb, elseb \mid \} expression\}$
(15) $declexpn \triangleq \Pi\{decl \mapsto declaration;\ body \mapsto expression\}$
(16) $declaration \triangleq \Sigma\{cdecl \mapsto sepair;\ vdecl \mapsto vardecl;\ dadecl \mapsto darraydecl;\ tdecl \mapsto stpair;$
$fdecl \mapsto funcdecl;\ pdecl \mapsto procdecl;\ rfdecl \mapsto funcdecl^*;\ rpdecl \mapsto procdecl^*\}$
(17) $vardecl \triangleq \Pi\{id \to S;\ t \mapsto tdeno;\ val \mapsto expression_\perp\}$
(18) $darraydecl \triangleq \Pi\{id \mapsto S;\ t \mapsto tdeno;\ l, h \mapsto expression\}$
(19) $stpair \triangleq \Pi\{id \mapsto S;\ t \mapsto tdeno\}$
(20) $funcdecl \triangleq \Pi\{id \mapsto S;\ fp \mapsto stbtriple;\ t \mapsto tdeno;\ decl \mapsto declaration^*;$
$body \mapsto command^*;\ result \mapsto expression\}$
(21) $procdecl \triangleq \Pi\{id \mapsto S;\ fconp, fvarp \mapsto stbtriple^*;\ body \mapsto command^*\}$
(22) $stbtriple \triangleq \Pi\{id \mapsto S;\ t \mapsto tdeno;\ isarray \mapsto Bool\}$
(23) $tdeno \triangleq \sum\{tn \mapsto S;\ rtdo \mapsto stpaire^*;\ atdo \mapsto atdeno\}$
(24) $atdeno \triangleq \Pi\{t \mapsto tdeno;\ l, h \mapsto Intliteral\}$

3 语义论域

指称语义定义建立在对象语言的抽象语法之上,它通常由通过递归论域方程组求解所定义的语义论域和一组从句法对象集到语义论域的语义函数组成.语义函数也是通过相互调用的函数方程组给出的,函数方程组的求解可以在基调代数的框架下得到解释:把抽象语法看作一个基调代数的基调定义(Signature),以这个基调下的全体基调代数作为对象,同态函数作为态射可得到一个范畴,句法对象集加上句法构造运算构成的基调代数 Syn 是这个范畴的初始对象,对抽象语法进行语义解释的实质是构造一个刻画语义构造的基调与 Syn 相同的基调代数 Sem,其论域集族由前面定义的各语义论域组成,语义的合成过程通过代数运算刻画,这样,语义函数就是 Syn 到 Sem 的唯一同态映射.[3]当模型语言规模较小时,采用传统的函数方程形式可能较为合适,然而对于真正的程序设计语言,采用基调代数的形式更好一些,因为代数框架具有良好的模块性质,可以将描述对象局部化,降低修改的复杂度,同时也比较符合直观,本文使用后一种方法,下面将逐步给出一个刻画 L_1 语义的基调代数,这一节讨论论域的定义.为节省篇幅不再给出对应于前一节中抽象句法的基调定义,只作一些必要的说明.从 L_1 的抽象句法可以看出,当模型语言的规模增大以后,句法对象通常以两种形式出现,即原始形式和串形式,为了统一刻画各种对象的并置运算,给定句法对象集后,首先定义一个对应于该句法对象集的基本基调.

定义 3.1. 给定非空有限集 S,S 上的基本基调 BS 定义如下,BS 的类集为 $bcS+listS$,运算符集为 $nullS+concS$.对 $a \in A$,$\langle null, a\rangle : \to \langle list, a\rangle$,$\langle conc, a\rangle : \langle bc, a\rangle, \langle list, a\rangle \to \langle list, a\rangle$.为简便起见,$\langle bc, a\rangle$ 常简写为 a,$\langle list, a\rangle$ 常简写为 a^*.

定义 3.2. $synset \triangleq \{program, command, vname, expression, declaration, tdeno, funcdecl,$
$procdecl, sepair, stpair, stbtriple, Bool, S, Charliteral, Intliteral\}$

语义规约通常包括静态语义和动态语义的定义,静态语义用来说明难以用语法描述的上下文约束条件,动态语义用来表示满足静态语义约束的语法对象的含义.传统指称语义定义往往忽略静态语义,或者将两种语义分别叙述.这种作法的弊病是语义规约不能显式地区分编译与执行错误,也影响了动态语义的准确性.另外,程序设计语言走向成熟的重要特征之一是上下文约束条件越来越精致和复杂,从而达到改进语言的可靠性、安全性、易维护性等功能.这使得语义规约中的静态成分占有越来越重要的地位.本文的作法是将静态语义同动态语义结合到一起,每个语法对象同时有静态和动态两种解释.语义对象被定义为由静态指称与动态指称构成的,并满足匹配条件的二元组.

定义 3.3. 给定非空有限集 $Label$,$Label$ 的一个框架函数是一个三元组 $frame = \langle SD, DD, match\rangle$,它们都是 $Label$ 上的函数,满足 $\forall x \in Lable \quad match(x): SD(x) \times DD(X) \to B$.给定一个 $Label$ 上的框架函数 $frame$,由它确定的语义论域函数 D 是 $Lable$ 上的函数.对 $x \in Lable$,$D(x) \triangleq \{(u, v) \mid u \in SD(x) \wedge v \in DD(x) \wedge match(x)(u, v) = tt\}$.

下面先给出几个与存储有关的语义指称集.

(1) $Input, Output \triangleq C^*$
(2) $Bvalue \triangleq boolB + charC + intZ$
(3) $Store \triangleq \omega \overset{fin}{\to} Bvalue$
(4) $Storein \triangleq Store \times Input$
(5) $Storeio \triangleq Store \times Input \times Output$
(6) $Siotrace \triangleq Storeio \to Storeio^*$

另一个重要的概念是环境.

© 中国科学院软件研究所 http://www.jos.org.cn

定义 3.4. $elabel \triangleq \{value, variable, type, lookin, function, procedure\}$，设 *frame* 为 elabel 上的框架函数，D 为 *frame* 所确定的语义论域函数，则

$$Senviron \triangleq S \to (const\ SD(value) + var\ SD(variable) + type\ SD(type) + look\ SD(lookin) + func\ SD(function) + proc\ SD(procedure))$$

$$Denviron \triangleq S \to (const\ DD(value) + var\ DD(variable) + type\ DD(type) + look\ DD(lookin) + func\ DD(function) + proc\ DD(procedure))$$

$$Environ \triangleq S \to (const\ D(value) + var\ D(variable) + type\ D(type) + look\ D(lookin) + func\ D(function) + proc\ D(procedure))$$

下面讨论上述定义中 *frame* 的定义，为此，再给出几个与类型、值和变量有关的语义指称集：

(7) $Btype = stdtype\{B, C, Z\} + rtypeRtype + atypeAtype$；$Rtype = S \to Btype$；$Atype = Btype \times \omega \times \omega$

(8) $Type \triangleq btypeBtype + ltypeBtype$

(9) $Value = bvalueBvalue + rvalRvalue + avalAvalue$；$Rvalue = S \to Value$；$Avalue = Value^* \times \omega$

(10) $Variable = Location(\omega) + rvarRvar + avarAvar$；$Rvar = S \to Variable$；$Avar = Variable^* \times \omega$

因为涉及递归定义，*BtypeValue* 和 *Variable* 在这里是通过方程定义的，但与一般的论域方程不同，它们都有明显的树形结构，也可以采用显式的递归定义. 上述方程在集合范畴中也存在解，所以并不需要论域理论. 下面，首先给出 SD 的定义.

(1) $SD(value) \triangleq Type$

(2) $SD(variable) \triangleq Type$

(3) $SD(type) \triangleq Btype$

(4) $SD(lookin) \triangleq \{eoln, eof\}$

(5) $SD(function) \triangleq Type^* \times Btype$

(6) $SD(procedure) \triangleq Type^* \times Type^*$

DD 的定义如下：

(1) $DD(value) \triangleq Value$

(2) $DD(variable) \triangleq Variable$

(3) $DD(type) \triangleq Store \to Store \times Variable$

(4) $DD(lookin) \triangleq Input \to B$

(5) $DD(function) \triangleq Value^* \to (Store^\# \to Value_{\perp_d})_{\perp_s}$

(6) $DD(procedure) \triangleq Value^* \times Variable^* \to Siotrace_{\perp_s}$

类型的静态含义是复杂值及变量构成方式，其动态含义则是在内存中分配单元的方式. 由于函数定义采用了闭式作用域机制，函数的执行过程与内存状态无关，而只依赖于常量参数. 函数的执行也不改变内存状态，出现在函数的 DD 值中的 $Store^\#$ 分量只是临时内存，在函数调用过程结束时被释放. 当函数和过程的参数出现数量或类型的不匹配时，结果为静态错误信息 $\perp_s$，为节省篇幅不再给出 *match* 的定义.

定义 3.5. 设 L 为 synset 的基本基调的论域函数，$bset \triangleq \{Bool, S, Charliteral, Intliteral, program\}$，$L(Bool) \triangleq B$，$L(S) \triangleq C^*$，$L(Charliteral) \triangleq C$，$L(Intliteral) \triangleq Z$，$L(program) \triangleq (InputStoreio^\#)_\perp$，对 $x \in bset$，$L(x^*) \triangleq (L(x))^*$. 设 $cset \triangleq synset - bset$，$ls$ 为 *cset* 的基本基调的类集，*frame* 为 ls 上的框架函数，D 为 *frame* 所确定的语义论域函数，对 $x \in cset$，$L(x) \triangleq Environ\ D(x)$.

下面讨论上述定义中 *frame* 的定义，为此，定义一个辅助框架函数 *frame'*，首先定义 SD'.

(1) $SD'(command) \triangleq \varnothing$

(2) $SD'(vname) \triangleq Type$

(3) $SD'(expression) \triangleq Type$

(4) $SD'(declaration) \triangleq Senviron$

(5) $SD'(tdeno) \triangleq Btype$

(6) $SD'(funcdecl) \triangleq S \times SD(function)$

(7) $SD'(procdecl) \triangleq S \times SD(procedure)$

(8) $SD'(sepair) \triangleq S \times Type$

(9) $SD'(stpair) \triangleq S \times Btype$

(10) $SD'(stbtriple) \triangleq S \times Type$

当 $x = command, declaration$ 时，$SD'(x^*) \triangleq SD'(x)$，否则，$SD'(x^*) \triangleq (SD'(x))^*$.

DD' 的定义如下：

(1) $DD'(command) \triangleq Siotrace$

(2) $DD'(vname) \triangleq Storein \to store^\# \times Variable_{\perp_d}$

(3) $DD'(expression) \triangleq Storein \to Store^\# \times Valued_{\perp_d}$

(4) $DD'(declaration) \triangleq Storein \to Store^\# \times Denviron_{\perp_d}$

(5) $DD'(tdeno) \triangleq DD(type)$

(6) $DD'(funcdecl) \triangleq S \times DD(function)$

(7) $DD'(procdecl) \triangleq S \times DD(procedure)$

(8) $DD'(sepair) \triangleq S \times (Storein \to Store^\# \times Value_{\perp_d})$

(9) $DD'(stpair) \triangleq S \times DD(type)$

(10) $DD'(stbtriple) \triangleq S \times DD(type)$

当 $x = command, declaration$ 时，$DD'(x^*) \triangleq DD'(x)$；

当 $x = tdeno, funcdecl, procdecl, sepair, stbtriple$ 时，$DD'(x^*) \triangleq (DD'(x))^*$.

© 中国科学院软件研究所 http://www.jos.org.cn

$DD'(vname^*)\triangle Storein\to store^{\#}\to Variable^{\dagger}_{\perp_d}$

$DD'(expression^*)\triangle Storein\to Store^{\#}\to Value^{\dagger}_{\perp_d}$

为节省篇幅，只给出 $match'(declaration)$ 的定义：对 $u\in SD'(declaration)$，$v\in DD'(declaration)$，

$$match'(declaration)(u,v)\triangle \forall\, si\in Storein,\ \textbf{if}\ \neg norm(P_{2,1}(v(si)))\ \textbf{then}\ P_{2,2}(v(si))=\perp_d$$
$$\textbf{else}\ P_{2,2}(v(si))\neq\perp_d\wedge dom(u)=dom(P_{2,2}(v(si)))\wedge$$
$$\forall\, x\in dom(u)\ match(P_{2,1}(u(x)))(u(x),P_{2,2}(v(si)))(x))\ \textbf{fi}$$

最后，定义 $SD(x)\triangle SD'(x)_{\perp}$，$DD(x)\triangle DD'(x)_{\perp}$，对 $u\in SD(x)$，$v\in DD(x)$，

$$match(x)(u,v)\triangle \textbf{if}\ u=\perp\ \textbf{then}\ v=\perp\ \textbf{else}\ v\neq\perp\wedge match'(x)(u,v)\ \textbf{fi}$$

从上述定义可以看出，程序设计语言的表达式与普通的表达式含义不完全相同，本质上它不是表示一个值而是一个求值过程．因为表达式包含函数调用，函数体可以包含语句，另外还有递归定义的函数，这使得求值过程可能因异常终止或死循环而得不到结果，而且表达式的求值可能会引起内存状态的变化．这称为函数的负作用．L_1 为了提高可靠性和安全性禁止表达式有负作用，因此，在以上定义中出现的 $Store^{\#}$ 分量也是临时内存，在计算过程中止时被撤消，引入 $Store^{\#}$ 分量是因为结果值依赖于临时内存状态，也是为了反映计算过程的动态特征，但说明项的执行过程在更新环境的同时也会改变内存状态，因为在内存中分配单元以及显示或隐式的初始化过程都会修改内存：变量、表达式和说明项的计算过程都与 Output 无关，但与 Input 有关．这是因为系统提供了自变量为输入状态的 eol，eof 预定义函数．

上面给出的语义论域是面向批处理程序的，对应于交互式程序的语义论域将在本文续篇中讨论．本文续篇还将针对批处理和交互式两种情形，讨论原子句法对象的语义解释以及语义论域上对应于句法构造运算的语义构造运算，在新的语义框架下完成对 L_1 的语义描述．

参考文献

1 Mosses P D. Action semantics. Cambridge: Cambridge University Press, 1991

2 De Bakker J, De Vink E. Control flow semantics. Cambridge: MIT Press, 1996

3 陈意云. 形式语义学基础. 合肥，中国科学技术大学出版社，1994
(Chen Yi-yun. Fundation of formal semantics. Hefei: China University of Science and Technology Press, 1994)

4 Watt D A. Programming language syntax and semantics. Hemel Hempstead: Prentice Hall International, 1990

5 Tennent R D. Elementary data structures in Alaol-like language. Science of Computer Programming, 1989, 13(1): 73～110

6 Mosses P D. Denotational semantics. In: Van Leeuwen ed. Handbook of Theoretical Computer Science, Vol B. Amsterdan: Elsevier, Amsterdam and the MIT Press, 1990. 575～631

Semantics of Program Based on Trace Part 1: Trace and Semantic Objects

WANG Yan-bing LU Ru-zhan

(*Department of Computer Science Shanghai Jiaotong University Shanghai* 200030)

Abstract This paper and its sequel present a denotational semantic descriptive method based on trace. The method combines characteristics of operational semantics and algebraic semantics, avoids domain theory as the theoretical basis, and unites static semantic and dynamic semantic description. It is contended that the new method is more suitable to be used to deal with real programming language by using it to define semantics of a middle-scale Algol-like model language. In this paper, the concept of trace is introduced first, then a model language is given and the corresponding semantic domains are defined without using recursive domain equations.

Key words Trace, Algol-like language, abstract syntax, signature-algebra, semantic domain.

© 中国科学院软件研究所 http://www.jos.org.cn