一种命令式风格的面向对象语言语义框架

华保健[1] 高 鹰[2]
(中国科学技术大学软件学院 合肥 230027)[1] (Intel 中国研究院 北京 100086)[2]

摘 要 面向对象语言在软件工程实践中有着广泛的应用。为面向对象语言定义严格的语义有助于理解面向对象语言的本质特征,对验证软件、提高软件系统可靠性等也具有重要意义。给出了一种新的面向对象语言的语义框架,该框架基于命令式的风格,具有操作语义和类型规则;证明了该语义框架的类型安全定理。

关键词 面向对象语言,类型系统,操作语义,语义框架
中图法分类号 TP311 **文献标识码** A

Imperative-style Semantics Framework for Object-oriented Languages

HUA Bao-jian[1] GAO Ying[2]
(School of Software Engineering, University of Science and Technology of China, Hefei 230027, China)[1]
(Intel Lab China, Beijing 100086, China)[2]

Abstract Object-oriented languages have wide applications in software engineering practice, and designing a rigorous semantics framework for these language has great importance in understanding their key features, and also in software verifications etc. This paper presented a new imperative-style semantics framework for object-oriented languages. This framework includes the operational semantics and a type system. This paper proved the soundness theorem.

Keywords Object-oriented languages, Type systems, Operational semantics, Semantics framework

1 引言

最近十几年以来,面向对象语言已经成为主流的程序设计框架之一,在软件工程的实践中得到了广泛应用。面向对象语言变得愈发流行的主要原因之一是其成功地组合了多种程序设计的重要特征。以 Java 语言[1]为例,其设计借鉴了许多成功语言的特性,如 Smalltalk,C++等。因此,相对于传统的命令式语言,对面向对象语言的核心特性的理解和使用变得更加复杂;给面向对象语言定义严格的形式化的语义框架也成为一个非常重要的研究课题,成为近年来的研究热点。进一步,为面向对象语言设计形式语义框架也有助于增强针对面向对象语言程序的验证系统的表达能力,从而进一步提高由面向对象语言构造的软件系统的正确性和可靠性,对软件安全的相关研究也有重要意义。

已有的面向对象语言形式语义的研究可以分为 3 类方法:对象编码、基本对象和 Hoare 逻辑。文献[2-4]研究了对象编码,这种方法的主要思想是把面向对象语言的典型特征都翻译到一个核心的形式系统中,然后在形式系统中研究这些面向对象特征的行为;根据表达能力的需要,可以采用的核心形式系统包括演算或系统 F 等。文献[5]提出了基本对象的方法,在这个方法中,不再进行类和对象的翻译,而是对这些特性直接建模。微软研究院提出了基于 Hoare 逻辑[6]来定义面向对象语义的新方法,该方法中类和对象的语义以 Hoare 逻辑前后断言的形式呈现,并且给出了一个称为 Spec# 的实际系统[7]。

已有的研究尽管在形式化面向对象语言语义上做了大量工作,但仍存在很多困难和问题。例如,在对象编码方法中,存在的根本困难是它对语义的建模是间接进行,即必须信任中间翻译的步骤是可靠的,这将显著地增大系统的受信计算基(TCB)。而在基于 Hoare 逻辑方法已有研究中,可靠性和完备性都没有得到证明,因此该方法难以用到高安全需求的软件构造中。

针对已有研究中存在的问题,本文围绕面向对象语言的语义定义这一课题,给出了一种命令式风格的语义框架。为了形式表达面向对象语言的语义,本文首先给出了一个小的面向对象语言的模型,称为 Cool(Core Object-oriented Language),该模型中包括了面向对象语言中的典型特征,如类、对象、继承、虚函数等。本质上,Cool 建模了 Java 或者 C# 的一个子集。为了定义 Cool 的动态语义,本文使用了操作语义的方法,该方法基于一个抽象定义的机器模型及施加于其上作为二元关系的一组操作语义规则。静态语义(类型规则)一般是面向对象语言中最复杂的部分,因为其中包括类继承、方法覆盖等复杂情况。为了处理这一部分,本文研究了一个静态语义的系统,并且在其中形式化了虚函数表子类型和类型

到稿日期:2012-04-28 返修日期:2012-07-10 本文受国家自然科学基金项目(61170051)资助。
华保健(1979—),男,博士,讲师,主要研究方向为程序语言的设计与实现、形式语义、软件安全,E-mail:bjhua@ustc.edu.cn;高 鹰 男,博士,研究员,主要研究方向为形式化方法、软件安全。

规则。

本文第2节介绍Cool的语法规则;第3节讨论操作语义;第4节讨论类型系统;最后总结全文并指出下一步的工作。

2 语法系统

本节讨论Cool的语法系统,该语法系统在图1中给出。该语法与广泛使用的Java或C#语言的语法规则非常接近,但更加抽象。这种保持相对抽象的设计方案的主要优点是能够使得该系统不拘泥于某种具体的语言的语法规则,而具有更广泛的适用性。

一个程序 p 由零个或多个类 c 组成(这里及以下将使用*Kleen*闭包的符号 * 表达零次或多次的重复)。一个类 c 有两种形式:没有显式继承或有显式继承(类 x 继承自类 y),为了与面向对象语言中通常的规定一致,这里也规定若没有显式继承,则该类默认继承自Object类。每个类包括了成员变量声明和方法 m 的声明。

```
(程序)   p::= c*
(类)     c::= class x{(t x;)* m*}
           | class x extends y {(t x;)* m*}
(方法)   m::= public t x ((t x)*)
           { (t x;)* s* return e;}
(类型)   t::= int | boolean | int[] | x
(语句)   s::= if (e) s* else s*
           | x=e;
           | x=new x ();
           | x=new int[e];
           | x=e.y(e*);
           | x[e]=e;
(表达式) ::= x | intlit | e op e | e[e] | e.length
           | true | false | this | null
(算符)   op::=+ | - | * | / | …
```

图1 Cool的语法规则

方法 m 中包括局部变量的声明和若干条语句 s,且以返回语句结尾。语句包括典型的分支条件、赋值、对象构造、方法调用等,这里为了叙述简单起见,没有加入循环等。一方面,这些语言特性可以通过现有的机制来实现(例如像函数式语言那样用递归来实现循环等);另一方面,加入这些语句特征不会增加额外的困难。

表达式包括常用的整型和布尔型及数组类型上的操作,这些在常见面向对象语言中普遍存在,不再赘述。

尽管与已有研究工作中所定义的语言模型相比,图1中所定义的模型已经更加丰富,但仍然有许多面向对象语言特性没有包括到图1中的模型中,即使如此,在本文所讨论的技术上稍加扩展即可支持这些特性。例如,包的机制只是对代码命名空间的封装,不会影响程序的动态行为;接口对象只是普通对象的一种特殊情况;而对异常处理,在下面小节将要给出的抽象机器模型上增加一个异常栈即可对其支持。因此,在对所有的语言特性进行研究后,本文选择使用一个相对紧凑的子集,它更便于研究面向对象语言的核心特征。

3 操作语义

本节讨论Cool的操作语义。3.1节讨论一个适用于面向对象语言的抽象机器模型规范程序状态;3.2节将讨论程序的归约规则,该规则定义为抽象机器状态的二元转换关系:对给定的程序,都最多存在一个归约规则进行应用;如果没有任何规则可用,则会出现运行错误。

3.1 抽象机器模型

该抽象机器模型的定义见图2。抽象机 M 是三元组(S,H,C),其中 S 是存储,H 是堆,C 是虚函数表(也有部分面向对象语言称为类表)。存储 $S:x\mapsto v$ 是一个部分映射,把声明变量 x 映射到它所对应的存储值 v;S 中至少包含一个映射 $ret\mapsto v$,其中 ret 是一个预定义的变量,存储当前方法的返回值(细节将在下面小节讨论)。

存储值 v 包括5种可能形式:整型常量 n;布尔常量true和false;堆地址 l 和一个特殊的值▲,其表示存储中未初始化的值。堆地址 l 的形式相对复杂:一方面,在通常的面向对象语言中,堆地址 l 一般称为引用,但本文称 l 为堆地址而不是引用强调了这里讨论的是相对底层的语言语义,而不是源语言的特性;另一方面,堆地址 l 是抽象的实体,而非具体的机器地址;这不但有效隐藏了许多底层实现无关的细节,而且有效地表达了在面向对象语言上通常会施加的很多限制,如不允许指针算术等。

```
(抽象机)   M::= (S,H,C)
(存储)     S::= x↦v,S | ret↦v
(堆)       H::= l↦hv,H |.
(存储值)   v::= n | true | false | l | ▲
(堆值)     hv::= {vptr↦c, x1↦v, …, xn↦v}
             | [vptr↦c,k,n0,…,nk-1]
(虚函数表) C::= c↦T,C |.
(表项)     T::= f↦(a*,x*,s*),T |.
```

图2 抽象机器模型

另一个设计中的关键点是存储值都是原子性的,即都不包含内部结构,这意味着可以把这个模型映射到实际的虚拟机或物理机器。这反映了图2中的模型是对实际虚拟机或物理机器忠实的建模。

堆 $H:l\mapsto hv$ 把堆地址 l 映射到存储在此地址上的堆值 hv。堆值 hv 建模了对象且包括两种形式。第一种形式{vptr $\mapsto c,x_1\mapsto v,\cdots,x_n\mapsto v$}建模了一个共包括 $n+1$ 个数据的对象,其中后面 n 个域 $x_1,\cdots,x_n$ 是通常的数据域(每个都映射到堆值 v),第一个域vptr非常特殊,它就是通常面向对象语言中的虚函数表指针,指向某个具体的类 c 所对应的虚函数表项 T。图2的机器模型中把所有的类所对应的虚函数表项构成了一个统一的虚函数表 $C:c\mapsto T$。每个虚函数表项 $T:f\mapsto(a^*,x^*,s^*)$ 把每个方法 f 映射到它所对应的形参列表 a、局部变量列表 x 和语句列表 s。因此通过虚函数表可以重构方法 f 的源代码相应的信息,这也意味着本模型具有支持反射的能力。与通常的面向对象语言一样,对vptr的赋值是在对象创建期自动完成的。

第二种值的形式是[vptr $\mapsto c,k,n_0,\cdots,n_{k-1}$],它代表了

一个数组对象：第一个域 vptr 仍是虚函数表指针；第二个域 k 是数组的长度域；后面 k 个域分别代表每个数组下标的值。

存储 S 和堆 H 都可以看出映射，本文中使用标准的符号 $dom(S)$ 和 $dom(H)$ 分别表示 S 和 H 的作用域；而 $range(S)$ 和 $range(H)$ 分别表示 S 和 H 的值域。

为形式化 S 的状态改变，本文用记号 $S[x \mapsto v]$ 表示把 S 中的变量 x 的值更新成 v，即等式

$$(S[x \mapsto v])(y)=\begin{cases} v, & \text{若 } x=y \\ S(y), & \text{其它} \end{cases}$$

成立。同样的操作和记号也适用于堆 H。

3.2 操作语义规则

本小节给出形式化的操作语义规则，它们都具有形式：

$$(S,H,C,h) \rightarrow_h (S',H',C',h')$$

其中，$M=(S,H,C)$ 是图 2 中定义的机器状态，h 是特定的语言结构，如语句或表达式等。这条规则表明在当前的抽象机器状态 (S,H,C) 下，通过执行 h 得到 h'，同时抽象机器的状态转换成 $M'=(S',H',C')$。注意到虚函数表 C 在程序的执行期间不会改变，即 $C'=C$ 始终成立，可把 C 从规则中省略，得到更为简洁的规则的形式：

$$(S,H,h) \rightarrow_h (S',H',h')$$

与通常面向对象语言如 Java 中相同，Cool 也采用从左向右计算、按值调用的规则。为了形式化这一点，引入一个归约环境 E：

$$E::=[] \mid E \text{ op } e_2 \mid v_1 \text{ op } E \mid E[e] \mid v[E] \mid E.\text{length} \mid !E \mid \text{if } (E)\ s^* \text{ else } s^* \mid x=E \mid x=\text{new int}[E] \mid x=E.y(e^*) \mid x=v.y(v^*,E,e^*) \mid x[E]=e \mid x[v]=E \mid \text{return } E$$

归约环境 E 是包含一个待补全式的表达式或语句，用符号 $E[e]$ 代表将 e 带入 E 中。引入归约环境 E，使得我们用具体语言特性无关的方式定义程序的计算顺序，如下：

$$\frac{(S,H,e) \rightarrow_e (S,H,e')}{(S,H,E[e]) \rightarrow_e (S,H,E[e'])}$$

对表达式的计算规则在下面给出：

$$\frac{v=v_1 \text{ op } v_2}{(S,H,v_1 \text{ op } v_2) \rightarrow_e (S,H,v)} \quad \text{(E-Op)}$$

$$\frac{H(l)=[\text{vptr},k,v_0,\cdots,v_{k-1}] \quad 0 \leqslant i \leqslant k-1}{(S,H,l[i]) \rightarrow_e (S,H,v_i)} \quad \text{(E-Array)}$$

$$\frac{H(l)=[\text{vptr},k,v_0,\cdots,v_{k-1}]}{(S,H,l.\text{length}) \rightarrow_e (S,H,k)} \quad \text{(E-Length)}$$

$$\frac{S(x)=v}{(S,H,x) \rightarrow_e (S,H,v)} \quad \text{(E-Var)}$$

$$\frac{}{(S,H,!\text{true}) \rightarrow_e (S,H,\text{false})} \quad \text{(E-True)}$$

$$\frac{}{(S,H,!\text{false}) \rightarrow_e (S,H,\text{true})} \quad \text{(E-False)}$$

第一条规则 E-Op 给出了对二元运算的规则，该规则没有副作用，即不会修改原有的机器状态。对其它语言特性的归约规则与此类似，不再赘述。特别要提到关于数组的规则 E-Array，这条规则表明堆 H 把堆地址 l 映射到一个数组 $[\text{vptr},k,v_0,\cdots,v_{k-1}]$，因此 $l[i]$ 将归约到 v_i，但这要求下标 i 必须在数组界限内，即条件 $0 \leqslant i \leqslant k-1$ 必须成立，称之为副条件，对副条件的检查可用文献[8]提出的技术。

对语句的归约有如下规则：

$$\frac{}{(S,H,x=v) \rightarrow_s (S[x \mapsto v],H,\cdot)} \quad \text{(S-Var)}$$

$$\frac{l \notin dom(H) \quad H'=H[l \mapsto \{\text{vptr}=\text{Object},x_1 \mapsto \blacktriangle,\cdots,x_n \mapsto \blacktriangle\}]}{(S,H,x=\text{new } y()) \rightarrow_s (S[x \mapsto l],H',\cdot)} \quad \text{(S-New)}$$

$$\frac{l \notin dom(H) \quad H'=H[l \mapsto \{\text{vptr}=\text{Object},k,\blacktriangle_0,\cdots,\blacktriangle_{k-1}\}]}{(S,H,x=\text{new int}[k]) \rightarrow_s (S[x \mapsto l],H',\cdot)} \quad \text{(S-Array)}$$

$$\frac{H(l)=\{\text{vptr}=c,x_1 \mapsto v_1,\cdots,x_n \mapsto v_n\} \quad C(c)(f)=(a_1,\cdots,a_n,y_1,\cdots,y_m,s^*) \quad S'=S[\text{this} \mapsto l,a_1 \mapsto v_1,\cdots,a_n \mapsto v_n] \quad (S',H,s^*) \rightarrow_s (S'',H',\cdot) \quad v=S''(ret)}{(S,H,l.f(v_1,\cdots,v_n)) \rightarrow_s (S''[x \mapsto v],H',\cdot)} \quad \text{(S-Call)}$$

$$\frac{}{(S,H,\text{return } v) \rightarrow_s (S[ret \mapsto v],H,\cdot)} \quad \text{(S-Ret)}$$

$$\frac{H(l)=[\text{vptr},k,v_0,\cdots,v_i,\cdots,v_{k-1}] \quad H'=H[l \mapsto [\text{vptr},k,v_0,\cdots,v,\cdots,v_{k-1}]]}{(S,H,l[i]=v) \rightarrow_s (S,H',\cdot)} \quad \text{(S-Assign)}$$

$$\frac{(S,H,s_1) \rightarrow_s (S',H',\cdot)}{(S,H,\text{if(true) } s_1 \text{ else } s_2) \rightarrow_s (S',H',\cdot)} \quad \text{(S-IfT)}$$

$$\frac{(S,H,s_2) \rightarrow_s (S',H',\cdot)}{(S,H,\text{if(false) } s_1 \text{ else } s_2) \rightarrow_s (S',H',\cdot)} \quad \text{(S-IfF)}$$

第一条规则 S-Var 对变量 x 赋值 v。第二条规则 S-New 进行对象分配，规则中的条件表明所分配的堆地址 l 是全新的。对数组的分配规则 S-Array 的含义与此规则类似。

最复杂的规则是方法的调用和返回语句。规则 S-Call 在抽象机器状态 (S,H) 上归约 $x=l.f(v_1,\cdots,v_n)$，为此，它首先找到对象 $H(l)=\{\text{vptr}=c,x_1 \mapsto v_1,\cdots,x_n \mapsto v_n\}$，然后依靠虚函数表指针 vptr 域的值 c 找到名字为 f 的方法，即：

$$C(c)(f)=(a_1,\cdots,a_n,y_1,\cdots,y_m,s^*)$$

上式用到了图 2 中 C 的结构。剩余的步骤是参数的按值调用传递。规则 E-Ret 和规则 E-Call 相对应，它设置了一个显式的变量 ret 来存储返回值。

4 静态语义

本节给出的静态语义借鉴了类型化演算[9]的设计思想，但是本文必须解决面向对象语言的独特特征，如虚函数表和继承等所带来的困难。

4.1 类型和类型环境

类型和类型环境在图 4 中给出。类型 τ 包括基本类型 int，boolean 和 int[]。变量 x 代表类类型，void 用来给语句定型，ns 用来给常量 null 定型。$\tau * \tau$ 和 $\tau \rightarrow \tau$ 分别是积类型与函数类型。

(类型)	$\tau::=\text{int} \mid \text{boolean} \mid \text{int}[] \mid \text{x} \mid \text{void} \mid \text{ns} \mid \tau * \tau \mid \tau \rightarrow \tau$
(方法环境)	$\Gamma::=\text{x}:\tau,\Gamma \mid .$
(类环境)	$\Sigma::=\text{x}:\tau,\Sigma \mid .$
(程序环境)	$\Delta::=\text{x}:(\Sigma,\text{y}),\Delta \mid .$

图 4 类型和类型环境

方法环境 Γ 存放每个方法的参数与局部变量及其所对应的类型；类环境 Σ 存放每个类的域的类型；程序环境 Δ 存

放整个程序的类型：其中 x 代表类的名字，y 是 x 的父类。

4.2 子类型规则

子类型规则给出类型间子类型二元关系，记作断言 $\Delta \vdash \tau_1 <: \tau_2$；即 τ_1 是 τ_2 的子类。该断言包括以下规则：

$$\frac{c_2 \text{ extends } c_1}{\Delta \vdash c_2 <: c_1} \quad \text{(S-As)} \qquad \frac{}{\Delta \vdash \tau <: \tau} \quad \text{(S-Refl)}$$

$$\frac{\Delta \vdash \tau_1 <: \tau_2 \quad \Delta \vdash \tau_2 <: \tau_3}{\Delta \vdash \tau_1 <: \tau_3} \quad \text{(S-Trans)}$$

$$\frac{\Delta \vdash \tau_i <: \tau_{i'} \quad (1 \leqslant i \leqslant n)}{\Delta \vdash \tau_1 * \cdots * \tau_n <: \tau_{1'} * \cdots * \tau_{n'}} \quad \text{(S-Prod)}$$

规则 S-As 称为指定规则，即类 c_2 是 c_1 的子类，当且仅当 c_2 被声明继承自 c_1。规则 S-Refl 是自反规则，即一个类总是自身的子类；规则 S-Trans 是传递规则；最后一条 S-Prod 是积类型上的子定型关系。

4.3 类型规则

对表达式的类型规则的断言是 $\Delta, \Gamma, c \vdash e : \tau$，即表达式 e 在环境 Δ、Γ 和类 c 的前提下具有类型 τ。具体规则如下：

$$\frac{\Delta;\Gamma;c \vdash e_1 : \text{int} \quad \Delta;\Gamma;c \vdash e_2 : \text{int}}{\Delta;\Gamma;c \vdash e_1 \text{ op } e_2 : \text{int}} \quad \text{(T-op)}$$

$$\frac{\Delta;\Gamma;c \vdash e_1 : \text{int}[] \quad \Delta;\Gamma;c \vdash e_2 : \text{int}}{\Delta;\Gamma;c \vdash e_1[e_2] : \text{int}} \quad \text{(T-Array)}$$

$$\frac{\Delta;\Gamma;c \vdash e : \text{int}[]}{\Delta;\Gamma;c \vdash e.\text{length} : \text{int}} \quad \text{(T-Length)}$$

$$\frac{\Gamma(x) = \tau}{\Delta;\Gamma;c \vdash x : \tau} \quad \text{(T-Var)} \qquad \frac{}{\Delta;\Gamma;e \vdash \text{this} : c} \quad \text{(T-This)}$$

$$\frac{}{\Delta;\Gamma;c \vdash \text{null} : \text{ns}} \quad \text{(T-Null)} \qquad \frac{}{\Delta;\Gamma;c \vdash n : \text{int}} \quad \text{(T-Int)}$$

$$\frac{}{\Delta;\Gamma;c \vdash \text{true} : \text{boolean}} \quad \text{(T-True)}$$

$$\frac{}{\Delta;\Gamma;c \vdash \text{false} : \text{boolean}} \quad \text{(T-False)}$$

$$\frac{\Delta;\Gamma;c \vdash e : \text{boolean}}{\Delta;\Gamma;c \vdash !e\text{boolean}} \quad \text{(T-Not)}$$

T-Var 规则表明具备变量的类型由环境 Γ 指定。T-Null 规则指定常量 null 的类型是类型符号 ns。表达式 this 的类型由其所包含的类指定，这反映在规则 T-This 中。其它规则的含义与此类似。

语句的类型规则断言形式是 $\Delta;\Gamma;c \vdash s : \text{void}$，即语句 s 在给定的环境中有类型 void；此处，void 表示语句是可定型的。下面给出语句的类型规则：

$$\frac{\Delta;\Gamma;c \vdash e : \text{boolean} \quad \Delta;\Gamma;c \vdash s_1 : \text{void} \quad \Delta;\Gamma;;c \vdash s_2 : \text{void}}{\Delta;\Gamma;c \vdash \text{if}(e) \text{ then } s_1 \text{ else } s_2 : \text{void}} \quad \text{(T-If)}$$

$$\frac{\Delta,\Gamma,c \vdash x : \tau_1 \quad \Delta,\Gamma,c \vdash e : \tau_2 \quad \Delta \vdash \tau_2 <: \tau_1}{\Delta,\Gamma,c \vdash (x = e) : \text{void}} \quad \text{(T-Assign)}$$

$$\frac{\Delta,\Gamma,c \vdash x : \tau \quad \Delta \vdash C <: \tau}{\Delta,\Gamma,c \vdash x = \text{new } C() : \text{void}} \quad \text{(T-New)}$$

$$\frac{\Delta,\Gamma,c \vdash x : \text{int}[] \quad \Delta,\Gamma,c \vdash e : \text{int}}{\Delta,\Gamma,c \vdash x = \text{new int}[e] : \text{void}} \quad \text{(T-Array)}$$

$$\frac{\begin{array}{l}\Delta,\Gamma,c \vdash x : \tau \quad \Delta,\Gamma,c \vdash : e : c' \\ \Delta \vdash f : \tau_1 * \cdots * \tau_n \to \tau' \\ \Delta,\Gamma,c \vdash e_i : \tau_i' \quad \Delta \vdash \tau_i' <: \tau_i\end{array}}{\Delta,\Gamma,c \vdash x = e.f(c_1, \cdots, c_n) : \text{void}} \quad \text{(T-Call)}$$

最复杂的规则 T-Call 中包括多个前提：首先确定对象 e 的类型是 c'，然后从 c' 所对应的虚函数表中确定方法 f 的形参类型是 $\tau_1 * \cdots * \tau_n$，则实参的类型必须逐个与形参类型满足子类型关系，即 $\Delta, \Gamma, c \vdash e_i : \tau_i$ 且 $\Delta \vdash \tau_1 <: \tau_i'$。

使用文献[10,11]提出的结构化归纳技术不难证明 Cool 的类型可靠定理。

定理 1(类型可靠定理) 若语句 s 是良类型的，则 s 的归约不会出现错误，即总有归约规则可以应用。

证明：基于对 s 的结构化归纳，其中，表达式部分基于对 e 的结构化归纳。

结束语 面向对象语言在软件工程中有着广泛的应用，对其语义进行严格的研究既是重要的课题，也是非常困难的研究领域。本文给面向对象语言定义了一种命令式风格的语义框架，与传统方法相比，该语义框架更通用，更接近软件工程实践中通常使用的语义模型。本文工作为面向对象语言程序的验证等奠定了新的语义基础，这也是今后重要的研究方向。

参 考 文 献

[1] Gosling J, Joy B, Steele G, et al. The Java Programming Language Specification(3rd edition)[M]. New Jersey: Prentice Hall, 2005

[2] Cardelli L. A Semantics of Multiple Inheritance [C]// Lecture Notes in Computer Science. Volume 173, Heidelberg: Springer, 1984: 51-69

[3] Kamin S, Reddy U. Two Semantic Models of Object-oriented Languages[M]. Theoretical Aspects of Obejct-oritend Programming: Types, Semantics, and Language Design, 1994: 464-495

[4] Cook W, Palsberg J. A Denotational Semantics of Inheritance and its Correctness[C]// ACM Symposium on Object Oriented Programming: Systems, Languages and Applications (OOPLAS). 1989: 433-444

[5] Igarashi A, Pierce B, Wadler P. Featherweight Java- A Minimal Core Calculus for Java and GJ[J]. ACM Transactions on Programming Languages and Systems, 2011, 23(3): 396-450

[6] Hoare C A R. An Axiomatic basis for computer programming [J]. Communication of the ACM, 1969, 12(10): 576-580

[7] Barnett M, Leino K, Schulte W. The Spec # programming system: An overview [C]// Proceedings of CASSIS 2004. LNCS vol. 3362, Springer, 2004: 158-170

[8] 华保健，陈意云，李兆鹏，等. 安全语言 PointerC 的设计及形式证明[J]. 计算机学报，2008，31(4)：556-564

[9] Pierce B. Types and Programming Languages [M]. Cambridge: MIT Press, 2002

[10] Drossopoulou S, Eisenbach S, Khurshid S. Is the Java Type System Sound[J]. Theory and Practice of Object Systems, 1999, 7(1): 3-24

[11] Wright A, Felleisen M. A Syntactic Approach to Type Soundness[J]. Information and Computation, 1994, 115(1): 38-94