计算机科学 1999Vol. 26№. 7

面向对象的类型-子类型分析及推理规则*)

Type-Subtype Analysis and Subtyping Inference Rules for Object-Orientation

李必信　王云峰　李宣东　郑国梁

(南京大学计算机科学与技术系　计算机软件新技术国家重点实验室　南京 210093)

Abstract In this paper, subtyping relationships between objects in behavioral inheritance, role inheritance and signature-compitable inheritance are raised. Then grammar condition and semantics condition and *sufficent & necessary* conditon and shortcomings of all subtypes are also analyzed. Also then, the connection between subtype and program refinement is discussed. Finally subtyping inference rules of behavioral subtype, role subtype, signature-compitable subtype and their fields subtype, methods subtype for object-orientation are given.

Keywords Subtype analysis, Behavioral inheritance, Role inheritance, Signature-compitable inheritance, Inference rule

1 引　言

面向对象的方法中，对象是作为现实世界中事物的自然映射。为了表达事物的分类关系，在面向对象领域中引入类型的概念，同一类对象具有相同的类型，不同类对象属于不同的类型。另一方面，在面向对象的软件开发的早期阶段，我们可能只知道某个数据抽象的部分操作和行为，随着软件开发过程的进展，对原有的数据抽象又有了新的要求，必须增加新的操作，从而产生新的数据抽象。新的数据抽象可看成是由老的数据抽象精化而来，它们分别由两种类型来描述，这两种类型之间存在子类型关系，而且这种精化过程可以重复多次，从而形成一个层次的子类型结构，它反映了程序各部分的不同要求。以层次方式来设计类型可以控制设计错误的影响范围，把握设计过程以及对实现给予指导。另外，满足子类型关系的继承还可以实现软件的设计重用，往往重用设计比重用代码更有利于软件开发。所以，为了进一步表达事物分类关系的层次性，可以用对象类型之间的子类型关系来表示事物分类的层次关系。本文揭示了行为继承、角色继承及型构兼容的继承中对象间的子类型关系，分析了子类型关系存在的语法条件，语义条件，充要条件等，分析了每种子类型关系的不足，阐述了子类型关系与程序精化的联系，最后给出面向对象的行为子类型，角色子类型，型构兼容子类型及其域和方法等的子类型化推理规则。

2 类型的概念

2.1 类与类型

类的概念：类是对象式程序的唯一的构造单位，是对象式程序设计语言的基本成分，是抽象数据类型的具体实现，它描述了一组相似对象的共同特性。类的语法定义如下：

〈类定义〉::=〈类首〉[〈参数部分〉][〈继承部分〉][〈创建子句〉][〈特征部分〉][〈不变式〉]end[12]

定义 1(类型)　类型 T 刻画了对象集 P(某一对象类型现有实例对象的动态集合)，状态集 Σ(某一类对象可能具有的合法状态的集合)，运算集 S(可作用于所有对象集的运算的集合)，方法集 M(对象为响应消息所能调用的自身操作的集合)和约束 C(规定了该类型对象的合法状态，以及合法状态

*)本文工作得到江苏省应用基础(编号 BJ97036)资助。李必信　博士研究生，研究方向为面向对象技术，程序理解；王云峰　博士研究生，研究方向为面向对象技术，形式化技术；李宣东　博士，副教授，研究方向为面向对象技术，实时系统；郑国梁　教授，博士生导师，研究方向为软件工程，面向对象技术。

之间的时序关系等约束条件)，可理解为是由它们构成的五元组.

类型＝(对象集，状态集，运算集，方法集，约束)，即 $T=(P,\Sigma,S,M,C)$。

类型与类的概念相对立，前者为程序中的概念，后者是程序运行时的概念，两者通过标识值的语言成分(例如，变量、函数，对象等)联系起来。类型的语法定义为：

〈类型〉::＝〈类类型〉|〈类扩展类型〉|〈形式类属名〉|〈依存类型〉[12]

类和类型的区别：类是程序的构造单位，是描述一组对象及其操作的唯一语言成分，故其主要作用是具体描述一组对象，提供运行时描述这组对象的"模板"。类描述对象的具体形式和其上可施加的具体操作，且强调所描述的一组对象的共性，因此，与具体的对象联系较密切，而与对象集的大小则联系较少；类型则是标志变量或表达式取值范围的一种语言成分，其主要作用是对这些变量或表达式运行时的取值进行约束。类型强调所描述的一组对象的范围和可施行操作的范围，与对象集的大小联系较密切，而与其中具体对象则联系较少。另外，并不是所有类都可以直接作为类型使用：类是类型的基础，类型靠类来定义，有些类可直接作为类型来使用，如 C++ 中的类就是类型。但是，也有一些类不能直接作为类型来使用，这是因为类型强调所刻画对象是确定的，即对象范围的确定性。对象范围不确定的类不能作为类型，如 Eiffel 中的类属类。

2.2 底类型

定义 2(底类) 任一类型 T 都是由某类 C 直接或间接定义的。类 C 就称为类型 T 的底类。

定义 3(底类型) 若类型 T 的底类为 C，则类型 C(当类 C 非类属类时)或 C 的导出类型(当类 C 为类属类时)称为类型 T 的底类型。

由上述定义可见：类型 T 的底类也是 T 的底类型的底类。底类的概念反映了类型与类之间的联系。类型 T 隐含的适用于其变量的所有特征都是由该类的底类定义的。但类的正文结构中往往不是直接使用类，而是使用基于这些类的类型。引进类型的底类型概念，一方面是为了定义间接定义的类型的底类，另一方面是为了定义类型适应规则。

3 类型的语法和语义解释

用 b 代表任何底类型，则类型表达式可由语法定义为：

$\sigma::=b|\sigma_1\rightarrow\sigma_2$。

例如，如果 nat 是一个底类型，则 $nat\rightarrow nat$ 和 $nat\rightarrow(nat\rightarrow nat)$ 是类型。直观看来 $\sigma\rightarrow\tau$ 是所有从 σ 到 τ 函数的类型(或集合)。然而在不同的环境中，我们可以采用函数的不同形式概念。可以把从 σ 到 τ 的函数 f 看作是一条规则—该规则可以为每个 $x:\sigma$ 安排一个值 $f(x):\tau$，或者这条规则的某种"代码"。直观地讲，我们认为 $\sigma\rightarrow\tau$ 包含了从 σ 到 τ 的并在一些上下文中产生影响的所有函数。

定义 4(连续函数) 一个函数 $f:A^\sigma\rightarrow A^\tau$ 如果它是单调的且具有上确界的定向集合，则它是连续函数。

定义 5(两个函数相等) 如果两个函数在它们定义域中每个点处取值相同，则此两个函数相等，可记为：$f=g:\sigma\rightarrow\tau$ 当且仅当 $\forall x:\sigma.\ f(x)=g(x)$。

由此，我们给出类型的三种解释：

(1)完全的集合论解释：①一个底类型 b 可以任何集合 A^b；②函数类型 $\sigma\rightarrow\tau$ 表明了从 A^σ 到 A^τ 的所有函数集合 $A^{\sigma\rightarrow\tau}$。

(2)递归解释：①底类型可以表示任何适度的集合 (A^b,e_b)；②函数类型 $\sigma\rightarrow\tau$ 表明了从 A^σ 到 A^τ 的所有全局递归函数的适度集合 $(A^{\sigma\rightarrow\tau},e_{\sigma\rightarrow\tau})$。

(3)论域理论解释：①底类型可以表示任何完全偏序 $(A^b,\leqslant_b)$；②函数类型 $\sigma\rightarrow\tau$ 表明了从 A^σ 到 A^τ 的所有连续函数的完全偏序 $(A^{\sigma\rightarrow\tau},\leqslant_{\sigma\rightarrow\tau})$。

注意到：为了搞清象 $(b\rightarrow b)\rightarrow(b\rightarrow b)$ 这种更高阶类型的意思，我们必须确信每个函数空间 $b\rightarrow b$ 是完全偏序的。这可由下列约定实现：无论何时，只要对每个 $x\in A^\sigma$ 有 $f(x)\leqslant_\tau g(x)$ 成立，就认为 $f\leqslant_{\sigma\rightarrow\tau}g$ 成立。

4 子类型分析

4.1 子类型概念与性质

定义 6(子类型) 如果类型 A 的所有取值也是类型 B 的值，则称类型 A 是类型 B 的子类型，记为 $A\subseteq B$。

由此定义易知子类型关系 $\subseteq$ 具有以下性质：

性质 1(自反性) 任意子类型 T，均有 $T\subseteq T$。

性质 2(传递性) 任意子类型 T,U,V 若 $V\subseteq U$ 且 $U\subseteq T$ 则 $V\subseteq T$。

性质 3 子类型关系(记为"$\subseteq$")是一种偏序关系。

证明：由性质 1 和性质 2 知，$\subseteq$ 是自反的、传递的；设两个类型 A 和 B，如果 $A\subseteq B$ 且 $B\subseteq A$，则 A

$=B$，故$\subseteq$是反对称的，因而$\subseteq$是偏序关系。

4.2 行为子类型

下面给出几个概念：

定义 7（行为子类型） 若对于类型 S 中任一对象 O_1，存在类型 T 的对象 O_2，使得对于所有在类型 T 基础上定义的程序 P，当用 O_1 来替换 O_2 后，都不会改变 P 的行为，那么 S 是 T 的子类型。行为子类型定义又称替换规则，记为$\subseteq_{behavioral}$。

定义 8（行为继承） 满足行为子类型条件的继承关系称为行为继承。

为了保证子类型的正确性，行为子类型关系必须满足以下语法和语义条件：

语法条件：①子类型中必须至少具有父类型中的所有行为；②子类型中可以增加新的行为；③若对父类型中行为进行重定义，则重新定义的行为必须和父类型中的原行为相适应。这里的两个行为相适应是指：①它们具有相同的参数数目；②重定义行为的输入参数的类型必须是原行为相应参数的类型的父类型（或为相同类型）；③重定义行为的输出参数的类型必须是原行为相应参数的类型的子类型（或为相同类型）。

语义条件：语义性质直接关系到软件设计的正确性，故只从语法检查的角度考虑软件设计的正确性是不够的，我们希望子类型对象不仅能合法地成为父类型对象的替换物，更要求子类型对象拥有父类型对象的所有性质，能无差别地替换父类型对象，满足子类型关系。

不难证明其充要条件为：$\forall\ i:\{Pre\}O_2.action,\{Q\}\Rightarrow\exists\ j:(\{Pre\}O_1.action,\{Q\})$

这里$\{P\}S\{Q\}$表示若 S 在条件 P 下执行，则 S 一定终止且终止状态满足条件 Q。

为满足这种具有行为继承的可替换原则的子类型关系，需增加两个（语义）条件：①如果对父类中的行为进行了重定义，则重定义行为被调用后的返回值必须和调用父类中原行为的返回值相同；②如果对父类中的行为进行了重定义，则重定义行为被调用后修改子类中继承父类的属性值必须和在相同初始属性值条件下调用父类中原修改值相同。

引入继承机制的目的是为了在定义新的类时，如果能利用一个已存在的类，直接使用其所有属性和方法，然后再加入新的属性和方法去构造将十分方便。但此继承关系又产生了另一种关系：如果类 A 是从类 B 继承而来，则类 B 的对象就至少具有类 A 对象所具有的一切属性和方法。那么在任何需要类 A 对象的地方使用类 B 的对象应该导致同样的效果。据此替换原则可给出子类型定义：对所有用类型 T 来定义的程序 P，如果用类型 S 的对象 O_1 来替换程序中类型 T 的对象 O_2，P 的行为不变，则称 S 是 T 的子类型。

这种替换原则可表示为：$\{Pre\}P\{Q\}\Rightarrow\{Pre\}P_{O_2}^{O_1}\{Q\}$，其中 $P_{O_2}^{O_1}$表示将 P 中所有将要自由出现的 O_2 同时合法代入 O_1 的结果。这种类型之间在语义上的可替换关系，通常称为行为继承。程序 Q 是程序 P 的精化可定义为：$P\subseteq_{behavioral}Q\ iff\ \forall\ R:P\langle R\rangle\Rightarrow Q\langle R\rangle$，其中 $P\langle R\rangle$表示程序 P 满足后置条件 R 的最弱前置条件。可以进一步推出：

$$P\subseteq_{behavioral}Q\ iff\ \forall\ Pre\forall\ R:\{Pre\}P\{R\}\Rightarrow\{Pre\}Q\{R\}$$

根据可替换原则的表示可以发现实际上程序 $P_{O_2}^{O_1}$正是程序 P 的精化，由此可以采用行为继承来进行软件开发，就是逐步精化的过程。因此行为继承的一个好处是可以实现软件的增量开发。

行为子类型从对象行为的共性出发考虑子类型关系，认为现实世界中的各种事物都有其各自的行为规范，行为规范决定了事物的分类关系：具有相同行为的事物属于同一类，而对底层类事物行为共性的提取便产生了父类型，因而子类型必定遵守其父类型的行为规范，并可以增加新的行为。行为子类型的缺陷在于规范只能反映事物的一个侧面，而不是全部，而且行为规范中往往含有很多非本质的特征。

4.3 角色子类型

尽管行为子类型在很大程度上反映了事物之间的分类关系，但由于行为规范不能反映事物的全部特征，因而依然存在着子类型关系和事物的分类关系不相适应的情况，替换原则并非能适合于所有分类关系。同时行为子类型中不允许存在行为冲突，因而无法反映和解决现实世界中客观存在的行为冲突问题。另一方面行为继承中子类必须遵守父类的行为规范，因而继承必须以行为兼容的方式实现，存在很大的局限性。为此文[10]提出基于角色的子类型关系，把角色子类型规范作为事物的分类标准，这是因为：①角色子类型以事物所担任的角色作为事物的分类标准，角色包含了与事物分类有关的本质特征，能完全对应事物的实际分类关系；②角色允许行为冲突，通过一定的冲突解决机制来进行裁决，因而与现实世界之间建立了更为直接的映射关系，使用范围更加广泛；③角色继承在满足角色继承前提条件下，吸取了实现继承中软件复用方式灵活多样的

优点：子类不必遵守父类的行为规范，对于父类中的方法，子类可以继承，可以取消，也可以在保持其名的条件下重定义。这样做的目的是事物存在行为冲突的客观要求：事物的多重角色可以具有不同的行为规范。

定义 9（角色子类型） 设对象类型 A，B 分别表示角色 A'，B'。若角色 A' 是角色 B' 的先决条件，即 B' 的角色判断条件是 A' 的角色判断条件的增强，所有担任角色 B' 的事物也必然担任角色 A'，则称 B 是 A 的子类型，A 是 B 的父类型。这种基于角色分类的子类型关系称为角色子类型。角色子类型体现了事物的层次分类关系，记为 $\subseteq_{ak}$。

角色子类型的语义：子类型关系表示了对象集之间的包含关系，即父类型的对象集包含了子类型的对象集，子类型的对象自动成为父类型的对象，依据对象判定式来确定类型的对象集，对象类型间的子类型体现了对象判定式的增强。

定义 10（角色继承） 满足角色子类型条件的继承关系称为角色继承。

4.4 型构兼容子类型

子类型是定义在类间的一种关系，若类 A 是类 B 的子类型，则类 A 的实例对象的行为规范是类 B 的实例对象的行为规范的特殊化和精化，类 A 的实例可以代表类 B 的实例。由于不允许子类破坏父类的不变式（用以描述其实例变量的状态在生存期间必须满足的条件），不允许子类重定义其继承的父类方法以改变这些方法的规约，现有大多数面向对象的程序设计中支持的继承机制采用了增量继承的形式和子类型继承的实质。显然，子类型是定义在类间的一种关系，这种关系应该不依赖于继承而存在，将子类型关系和继承相联系便于类间子类型关系的确认，但是，一方面继承作为代码复用机制的灵活性受到限制，另一方面类间的子类型关系不能得到充分体现（子类型关系只能建立在父类和子类之间，无继承关系类之间不存在子类型关系）。因此，在面向对象的程序设计语言中将子类型关系和继承分离是目前的一种趋势。在面向对象程序设计语言中，若子类型关系和继承分离，则可以允许子类破坏父类的不变式和重定义其继承的父类方法以改变这些方法的规约，继承成为一种更加灵活的代码复用机制。为此，我们引入型构兼容子类型概念。

在面向对象的语言中，型构一般具有这样的形式：型构＝（类型序列，类型序列）。例如，特征 f 的型构的形式为：f＝（TL1，TL2），这里 TL1 和 TL2 都是类型序列。对属性特征，TL1 为空；对过程特征，TL2 为空；对函数特征，TL1 为变元类型序列（也可能为空），TL2 为结果类型序列（单元素序列）。

定义 11（型构兼容子类型） 设有两个型构 s 和 t，其中：$s=((B_1,\cdots,B_n),(S))$，$t=((A_1,\cdots,A_m),(R))$，则 t 是 s 的子类型（即 $t\subseteq_{signature}s$ 成立）当且仅当：①$n=m$，即 t 与 s 所含类型个数相同；②$\forall\ i(1\leqslant i\leqslant n)A_i\subseteq B_i$，且 $R\subseteq S$。

型构兼容子类型的语义：型构的组成成分是类型序列，两个型构相应的类型序列之间满足某种一致的子类型关系，并不要求两个型构之间（也就是包含此型构的两个类之间）一定具有某种继承关系，由此实现了子类型关系和继承关系的分离。

定义 12（型构兼容的继承） 满足型构兼容子类型关系的继承称为型构兼容的继承。

型构兼容继承的语义：子类可以拥有父类的变量和方法。对父类的任一方法，子类可以重新定义以改变该方法的规约和实现。对于未被重定义的父类方法，其规约在子类中不能改变，同时，子类还可以定义新的实例变量和方法以完成自身的构造。

显然，型构兼容的继承允许子类重定义其继承的父类方法以改变这些方法的规约，因此具备增量继承的灵活性。同时，型构兼容的继承确保父类方法的规约的改变必须通过子类的显式重定义，因此充分支持类的封装。

4.5 子类型的充分和必要条件及形式描述

定理 1（一般描述） 类型 A 是类型 B 的子类型（记为 $A\subseteq B$）的充分必要条件是 A 的实例具有 B 的实例所共有的概念和外部行为。

对象的外部行为由一组方法的型构和规约组成，方法的规约可以建立在由某个数学域 Σ 表示的抽象状态上的前后置条件的形式描述：$\{P\}m(p)\{Q\}$。由数学域 Σ 表示的抽象状态由称为对象的抽象状态，前置条件 $P=P(s,p)$ 描述了方法 m 执行前对象的抽象状态和方法 m 的参数必须满足的条件；后置条件 $Q=Q(s,s_0,p,r)$ 描述了方法 m 执行后对象的抽象状态和方法 m 的结果所满足的条件，这里 s 表示对象的当前抽象状态，s_0 表示方法 m 执行前的抽象状态；p 表示方法的参数；r 表示方法的结果，这样一个对象类型可以表示为五元组：对象类型＝(P,Σ,S,M,C)。这里 P 表示某一对象类型现有实例对象的动态集合，Σ 表示某一类对象可能具有的合法状态的集合，S 表示可作用于所有对象集的运算的集合，M 表示对象为响应消息所能调用的自身

操作的集合，C 规定了该类型对象的合法状态，以及合法状态之间的时序关系等约束条件。

定理 2（形式描述） 类型 $A=(P_A,\Sigma_A,S_A,M_A,C_A)$ 是类型 $B=(P_B,\Sigma_B,S_B,M_B,C_B)$ 的子类型（$A\subseteq_{sub,\phi}B$）的充分必要条件是：(1) $P_A\supseteq P_B$；(2) 存在转换函数 $\phi:\Sigma_A\to\Sigma_B$，对 B 的实例对象所共有的每个方法 $m\in S_B$（规约为 $\{P\}m(p)\{Q\}$），在 A 的实例对象所共有的方法中都能找到具有方法 m 的参数和结果类型的方法 $m'\in S_A$（规约为 $\{P'\}m'(p)\{Q'\}$），并且有 $((p\circ\phi)\to P')\wedge(Q'\to Q\circ\phi))$，这里 $\tau=\phi(\sigma)$，$P\circ\phi=P[\phi(\sigma)/\tau]$。

显然，利用上述形式描述方法的规约和确定对象类型之间的子类型关系，需要语言至少具有一阶谓词逻辑演算的描述和推理能力。

5 子类型推理规则

下面各子类型化推理规则中，判断“$\Gamma\vdash_s a:A$”表示：在环境 Γ 中型构（或行为规范或角色）为 s 的对象 a 具有类型 A。

5.1 域子类型（field subtyping）

域的类型是无序的，是类型的 n 元不相交并；成分类型由不同的标记索引，域类型的元素就是一对标记，类型的值是由标记确定的。

规则 1

$$\frac{\Gamma\vdash_s A_1\subseteq B_1\cdots\Gamma\vdash_s A_n\subseteq B_n\cdots\Gamma\vdash_s B_m:type}{\Gamma\vdash_s\{t_1:A_1\cdots t_n:A_n\}\subseteq\{t_1:B_1\cdots t_n:B_n\cdots t_m:B_m\}}$$

这说明 n 个成分的域也是 n+k 个成分的域，如果相应的成分具有子类型关系。

5.2 方法子类型（method subtyping）

规则 2 $\dfrac{\Gamma\vdash_s\tau_1\subseteq\sigma_1\quad\Gamma\vdash_s\sigma_2\subseteq\tau_2}{\Gamma\vdash_s\sigma_1\to\sigma_2\subseteq\tau_1\to\tau_2}$（逆变式规则）

规则 3 $\dfrac{\Gamma\vdash_s\sigma_1\subseteq\tau_1\quad\Gamma\vdash_s\sigma_2\subseteq\tau_2}{\Gamma\vdash_s\sigma_1\to\sigma_2\subseteq\tau_1\to\tau_2}$（协变式规则）

规则 2 是表示方法子类型关系的逆变式规则，规则 3 是表示方法子类型关系的协变式规则。规则 3 比规则 2 更直观，但可能带来意想不到的结果。例如使用协变规则构造的程序，静态类型正确，但运行时就可能出现类型错误。因此，所有静态类型的面向对象语言，或者使用逆变规则来定义方法类型，或者根本不允许方法重载中重新定义有争议的类型，因为逆变风格的重定义是相当有用的。

5.3 行为子类型（behavioral sutyping）

给定一个对象类型 A，A 的行为子类型 B 可通过向 A 中加入域（*fields*）或方法（methods）得到。

规则 4

$$\frac{\Gamma\vdash_{behavioral}\sigma_1\subseteq\tau_1,\cdots,\Gamma\vdash_{behavioral}\sigma_n\subseteq\tau_n}{\Gamma\vdash_{behavioral}\{x_1:\sigma_1,\cdots,x_n:\sigma_n,\cdots,x_k:\sigma_k\}\subseteq\{x_1:\tau_1,\cdots,x_n:\tau_n\}}$$

规则 4 说明对象 T 是对象类型 S 的子类型必须满足：①T 至少含有 S 的所有“行为规范”；②T 的这些“行为规范”均是从 S 中相应“行为规范”继承而来的。把规则 4 中符号“$\vdash_{behavioral}$”分别用“$\vdash_{role}$”或“$\vdash_{signature}$”代换，把“行为规范”分别用“角色”或“型构”代替后，就得到角色子类型、型构兼容子类型的推理规则。

结束语 P. America 在文[1]中阐述了并行面向对象语言子类型关系，在文[2]中阐述了具有行为子类型关系的面向对象语言的具体设计和实现；文[6,7]只给出一种特殊形式的对象类型的语义及解释；文[8,9]主要研究类型的推理问题，没有给出子类型的推理系统，特别是对行为子类型、角色子类型及型构兼容子类型的推理规则根本没有涉及；文[10]仅提出角色子类型的概念及性质，以及它与行为子类型的比较；文[11]提出型构兼容的继承的概念及其语义；本文与它们的不同之处是：严格地给出几种子类型的定义及语法及语义解释，给出子类型存在的语法及语义条件，以及充分必要条件，提出型构子类型的概念，给出一系列适合行为子类型、角色子类型及型构兼容子类型的子类型化推理规则，从而丰富了子类型系统理论。本文揭示了行为继承、角色继承及型构兼容的继承中对象间的子类型关系，分析了子类型关系存在的语法条件，语义条件，充要条件等，分析了每种子类型关系的不足，阐述了子类型关系与程序精化的联系，最后给出面向对象的子类型关系中记录、变量、函数等的子类型化推理规则，同时兼论了相关子类型、幂子类型、递归类型及类型一致等的含义及子类型化推理规则。

参考文献

1 America P. A parallel object-oriented language with inheritance and subtyping. In: Proc. ACM Conf. on Object-Oriented System, Language, and Application, 1990. 161～168

2 America P. Designing an object-oriented programming language with behavior subtyping. LNCS, 489 Springer Verlag, 1991

3 Shang D L. Covariant deep subtyping reconsidered. ACM Sigplan Notices, 1995(5)

4 Armstrong J M, Mitchell R J. Use and abuse of inheritance. Software Engineering Journal, 1994, 9(1): 19～26

（下转第 32 页）

制，可以处理这个问题。下面是一个解决方案。

我们定义两个系统消息，名为 CaptureResource 和 ReleaseResource，分别表示捕获该对象需要的资源和释放该对象占用的资源。对于一般对象，不需要处理这两个消息，也就是交给父类处理即可。对于直接和资源相对应的对象（文件对象、信号量对象等），需要作一定的处理。例如，对于文件对象，在释放资源时，应当存储打开文件的文件名，打开的模式，当前文件指针等。在捕获资源时，应当根据存储的文件名、打开模式，打开文件，将文件指针移到先前的位置。这些与资源直接相关的对象，其实现应当在系统提供的类库内，可以通过修改类库来实现。

我们可以注意到，当一个对象 A 收到上述消息时，不需要向包含的对象 B 发出相应消息，因为 B 会自己独立地进入或离开睡眠状态，而与 A 无关。

对于同步消息处理的系统来说，这样做就足够了，但是对于异步消息处理的系统，这样做还是有问题的，因为这样有可能会死锁，还需要一定的死锁避免算法，这里就不详述了。

另一个值得注意的问题是，捕获资源并不一定能够成功，即此类资源已经耗尽，只需要让其它某些对象睡眠就可以释放资源。这里的不成功是指外部条件发生了变化，例如上面例子中的文件对象，要打开文件时，发现文件已经被删除了。这种情况应当是由用户（指使用面向对象系统编程的人）来避免的，因为假设我们有很大的内存和系统资源，根本不必进行状态切换，那该用户的程序实际上做了一件事，就是当一个对象打开一个文件时，该文件被某个对象删除了。这是用户设计时应当避免的情况。

结论 综上所述，我们可以通过给对象定义状态（清醒、朦胧和睡眠）来解决面向对象的服务器不能正确得到解除引用信号时碰到的问题。使用这种实现方式产生的效率问题还需要进一步评价。上述问题的解决方法也不是唯一的，例如消息类型分析中实际上仅使用了对一个类的局部分析，应当还可以作对整个系统的全局分析。也可以使用静态分析的方法，进行状态的预转换（还没有到转换的时间，但是先进行转换，以达到提高效率的方法）等。

参 考 文 献

1 Jones S L P. The Implementation of Functional Programming Languages. Prentice Hall，1987

2 Java Development Kit 1.1.5 Specification. Sun Microsystem. Available at：1997 URL：http://www.javasoft.com/products/jdk/1.1/index.html

（上接第 27 页）

5 Cardelli L. Basic polymorphic typechecking. Science Computer Programming 8/2（April 1987）Revised 6/21/88

6 Martin Abadi and Luca Cardelli. A semantics of object types. In：Proceeding of the 19th Annual Symposium on Logic in Computer Science. 1994. 332～341

7 Abadi M，Cardelli L. An interpretation of objects and object types. In：the Proc. of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programing Languages，Jan. 1996

8 Plevyak J，Chien A. Precise concrete type inference for object-oriented language. In：Proc. ACM of Conf. on Object-oriented Programming Systems，Language，and Application，1994. 324～340

9 Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In：Proc. of Conf. on Object-oriented Programming Systems，Language，and Application. Pheonix，Arizona：SIGPLAN，ACM Press. 1991. 146～162

10 袁晓东，陈家骏，郑国梁. 基于角色分类的子类型关系. 计算机研究与发展，1997，34（11）：583～588

11 李宣东，郑国梁. 型构兼容的继承及其语义. 计算机学报，1996，19（1）：23～29

12 徐家福，王志坚，瞿成祥. 对象式程序设计语言. 南京大学出版社，1992，12