第6卷第5期 软件学报 Vol. 6, No. 5
1995年5月 JOURNAL OF SOFTWARE May 1995

行为规范作为类型*

梅宏 孙永强

(北京大学计算机系,北京 100871) (上海交通大学计算机系,上海 200030)

摘要 在面向对象程序设计中,继承性是导致语义复杂性的因素之一.本文讨论了作为代码共享机制的继承和表示功能特殊化的子类型的共存及相互关系.采用了将行为规范视为类型的概念,即类型为命名操作的集合.并在此基础上给出了类型、类及子类型关系的形式描述.

关键词 面向对象程序设计,数据类型,继承,子类型,行为规范.

面向对象程序设计(OOP)以其支持模块化设计、代码复用及可扩展性等优点,特别有利于大型复杂软件系统的构造.OOP已成为研究的热点和程序设计风格的主流.在OOP的研究中,对其和其他程序设计风格的交互及合成的研究也是一个重点领域.如在OOP和函数程序设计的合成方面就已经有了许多成果[1,2].

笔者在这方面也做了探索性的研究工作,设计并实现了函数式面向对象程序设计语言FOPL[3],该语言结合了OOP语言及函数式语言各自的优点和特点,同时提供对两种风格的支持.我们试图在类型理论下完成二者在语义级上的平滑合成,为此,FOPL的设计是基于OOP≈ADTs+inheritance这一框架[4].

类型及类型理论的研究一直是语言研究的重要方面.语言中类型设施的引入一方面可以使许多程序错误提前被检查出,另一方面也增加了编译时优化的机会,提高语言效率.从工程角度看,类型也可作为程序文档的重要内容.不少传统的OOP语言,如Smalltalk,都是无类型语言,因此附类型OOP语言的研究也是OOP语言研究的一个重要方面.FOPL是附类型的,其类型概念不同于传统语言的类型思想,不是将类型视为一组具有某种共性的值的集合,而是采用文献[5]及Russell语言[6]中的类型思想,将类型定义为描述某类对象的行为规范的一组命名操作的集合.FOPL中的类型给出了一个ADT的规范描述,而FOPL的类则是类型的具体实现,类的例化将创建对象.

继承是OOP语言的重要特征之一.然而,继承的引入带来了语言的语义复杂性,其本质还未为人们很好地理解.为此,形式的、数学的模型是需要的.已有一些关于继承的语义模型被给出[7-9],但遗憾的是这些模型比较简单,它们处理的对象实质上仅是程序执行中没有

* 本文1992-01-25收到,1993-10-04定稿

本研究得到国家自然科学基金及863高科技资助.作者梅宏,1963年生,1994年10月博士后出站,副教授,主要研究领域为新型语言及其支撑环境,软件工程.孙永强,1931年生,教授,博士导师,主要从事新型语言,计算理论方面的研究工作.

本文通讯联系人:梅宏,北京 100871,北京大学计算机系

状态变化的数学项,其内部结构也是对外可见的,不能处理存在于对象内部表示和外部观察间的不同.为克服这些不足及继承所带来的复杂性,FOPL 把传统意义的继承概念划分为继承和子类型关系两个概念,继承仅仅表示了语法级上的代码共享,而子类型关系则表示了对象间的行为依赖关系.下一节我们将详细讨论二者间的关系.

本文的目的是介绍 FOPL 中不同于传统的类型概念,并讨论由此而产生的类型和类的关系、子类型关系、继承等方面的问题.本文的工作构成了 FOPL 语言的基础,也提供了进一步研究 OOP 语言形式语义的基础.限于篇幅,有关工作可见文献[3].

1　继承和子类型关系

在大多数现行 OOP 语言中,继承具有两种不同意义,也正因为如此,继承的存在增加了语义复杂性.对继承的不同理解,使之成为一个概念的混淆源及争论的焦点.

一方面,继承作为一种类构造工具使用.在定义新类时,从一个现存类出发,在其实例变量和方法基础上加入新的内容而构成新类,新类继承旧类的变量和方法.这是实现意义上的继承,其目的是支持类间的代码复用和共享,从而大大节省代码量.另一方面,类间继承也暗示了另一种关系,如类 B 继承类 A,则 B 的每个实例应至少具有 A 的实例的所有变量和方法.因此,在程序中凡是需 A 类对象出现的地方,B 类对象也可出现,此即所谓包含多态性.这是行为意义上的继承,即行为特殊化,B 称为 A 的子类.

在传统 OOP 语言中,特别是 Smalltalk-80 中,总是用同一网层(Hierachy)来表示基于描述对象内部结构的代码上的共享而得到的实现继承网层和涉及对象的使用及对外可视行为的行为关系网层.然而,越来越多的人认识到,区别对待这两种网层是必要的[10],因为代码共享并不一定导致行为的特殊化,也不是导致这种行为特殊化的唯一方式.一方面,新方法在旧类上的加入可能导致与旧类功能不一致的新类,从而二者行为是不同的;另一方面,对同样的规范可有不同的内部表示和实现,从而使行为继承的两个类间没有代码共享.另外,代码共享也不仅仅局限在在旧类基础上加入新内容,我们也可通过排除旧类中的部分内容而得到共享.我们可用一个例子来清楚地说明这个问题:考虑类 STACK 和 DSTACK,分别表示栈和双端栈,显然,DSTACK 的外视接口包含了 STACK 的外视接口,DSTACK 的规范是 STACK 规范的特殊化,拥有 STACK 中所有操作,在这个意义下 DSTACK 是 STACK 的子类,然而,对于二者的具体实现,我们可考虑先实现 DSTACK,然后让 STACK 继承之并排除无用操作,也可先实现 STACK,然后再加上 DSTACK 独有的部分并继承 STACK 而得到 DSTACK 的实现.显然,在同一继承网层中无法同时且一致地描述这二者间的关系.为此,在 FOPL 中,我们区别对待这两种网层.我们用继承关系描述代码共享,而用子类型关系表示行为特殊化和规范继承.继承仅仅是作为代码共享、即实现的继承的手段,由继承而得到的新类可和原类是完全行为不同的.子类型关系则描述了对象间的行为依赖关系,两个具有子类型关系的类之间可以在实现上是完全不同的.

2　行为规范作为类型

当我们考虑一个对象时,我们并不关心其内部表示,而只是关心该对象可被使用的方

式.对象的类型将给出其可被如何使用的描述.要使用一个对象,只需了解其对外可视行为,因此其类型应给出对外可视行为的描述.

传统语言中的类型概念是类型,是一些具有某种共性的值的集合,显然,这种描述不是很适合于对象的,虽然对象在某一时刻的指称也是值,但更多的,对象应考虑为一个动态结构,其性质由其活动和与其他对象的交互展示出来.在EATCS的ESPRIT3020计划中[5],为考虑函数程序设计和OOP的合成,将类型定义为对象的对外可视行为规范,即类型作为具有对外可视的固有共性的对象集合[11].一个对象属于某类型,则其类的所有实例均属于该类型.一个类型T是另一类型S的子类型是指任何属于T的对象均属于S,也就是说,对任何对象,如它满足T的规范则一定满足S的规范,即规范T强于规范S.

Russell语言将数据类型定义为操作的集合,该操作集提供了一个单一全称值空间上值的解释.在Russell中,值本身是无类型且无意义的,仅仅在一定操作解释下才具有意义.值可被作不同解释,正如机器中的二进制位串,根据不同的操作,它可被解释为逻辑值、整数、浮点数或程序.Russell中值属于同一全称空间,它同构于其上的操作的空间,即满足方程$D \simeq D \to D$.采用如此类型定义有2个优点:(1)避开了是否一个值仅属于一个类型这一问题,值本身并无类型标记,仅通过一定操作对其作用才可解释,同一值可作多种不同解释;(2)类型本身也是值,从而可给出多态性的简单解释.

Russell的思想实质上是源于抽象数据类型的代数方法.一个抽象数据类型是一个初始代数的同构类,但抽象数据类型的意义并不是体现于其代数的载子中,而是体现于代数的在其载子元素上的操作的效果中,因此,一个抽象数据类型的行为规范体现于其操作中.Russell的方法正是对抽象数据类型的灵活而自然的支持.

FOPL的类型思想直接受益于文献[5]和Russell.在下面的讨论中,我们将给出其类型定义并讨论类型和类的关系.

定义 1. 类型

$$\mathrm{Type} ::= \langle \langle c_1 :: \mathrm{csig}_1, \cdots, c_m :: \mathrm{csig}_m \rangle, \langle s_1 :: \mathrm{ssig}_1, \cdots, s_n :: \mathrm{ssig}_n \rangle, \langle m_1 :: \mathrm{msig}_1, \cdots, m_l :: \mathrm{msig}_l \rangle, \mathrm{Axiom} \rangle$$

其中$c_i(1 \leqslant i \leqslant m)$为构造子,描述了类型元素的结构;$s_j(1 \leqslant j \leqslant n)$为选择子,描述了类型元素的对外可视属性;$m_k(1 \leqslant k \leqslant l)$为一般方法,描述了类型元素的对外可视行为.这三组操作共同对一全称域进行解释而构成一类型,sig是操作的标记(signature),给出操作类型说明,Axiom是对操作的规范,用前置、后置谓词描述.

操作集作为类型的方法和值集作为类型的方法是可以通过一定形式相联系的.我们引入抽象值并建立类型和抽象值集间关系.按传统观点,正是这些抽象值集构成类型,如{true,false}构成布尔类型.而在我们的类型定义中,我们只有一个全称值域,该域中的某一部分表示了一个抽象值集,全称域中元素的意义是由类型解释的,我们可理解为BOOL类型将全称域中某两元素分别解释为true和false,而将其他值映射到表示true和false的子域上,显然,类型应该是全称域上的一个收缩(retract),J. Donahue在文献[12]中用双严格收缩作为类型的语义.这样,我们可使用收缩将操作集合联系到一抽象值域上.如我们用整数1和0分别表示true和false,则我们可有收缩:

$$\lambda x \in D.\ \text{if } x \text{ is INTEGER then } (\text{if } x > 1 \text{ then } \bot \text{ else } x) \text{else} \bot$$

它给出了 D 上元素的解释,但不允许非原始值(如函数)到 BOOL 变量的赋值,如不考虑排除这种赋值可能,则可定义收缩为:

$$\lambda x \in D.\ \text{if } x \text{ is INTEGER then (if } x>255 \text{ then } \perp \text{ else } x) \text{ else } 255$$

这里 BOOL 值是用 8 位二进制表示的.

在我们下面的讨论中,我们采用在处理 ADT 时常用的技术:用某个数学域模拟一个类型的抽象值,如集合{true,false}作为 BOOL 的抽象值,即 BOOL 对象的抽象状态. 为此,我们给出类型的等价定义如下,其中显式地引入抽象状态集 S,前后置谓词中将使用抽象值.

定义 2. 类型

$\text{Type}::=\langle S,\langle c_1::csig_1,\cdots,c_n::csig_m\rangle,\langle s_1::ssig_1,\cdots,s_n::ssig_n\rangle,\langle m_1::msig_1,\cdots,m_1::msig_1\rangle,\text{Axiom}\rangle$.

定义 3. 类

$\text{Class}::=\langle R,\langle c_1::csig_1,\cdots,c_m::csig_m\rangle,\langle s_1::ssig_1,\cdots,s_n::ssig_n\rangle,\langle m_1::msig_1,\cdots,m_1::msig_1\rangle,\text{Equation}\rangle$.

其中 R 为该类的对象可取的抽象值在本类中的具体表示的集合,Equation 是对所有操作的具体实现.

定义 4. 对象

$\text{Object}::=\langle \nu,\langle s_1::ssig_1\cdots,s_n::ssig_n\rangle,\langle m_1::msig_1,\cdots,m_1::msig_1\rangle\rangle$

其中 ν 是变量,其可取值为对象的类中 R 的元素.

定义 5. 对象 o 具有类型 τ 是指:o 为 τ 的某实现类的例化,o 应被类型 τ 解释,即 o 的行为应满足 τ 规范.

然而,在什么条件下一个类是一个类型的实现呢? 这将是我们下面讨论的问题. 如前所述,一个类型潜在对应一个表示该类型对象的可能的抽象状态的抽象值域. 类型中的方法规范形为$\{P\}m(\bar{p})\{Q\}$,前置谓词为 $P=P(s,\bar{p})$,描述方法执行前状态,后置谓词为 $Q=Q(s,t,\bar{p},r)$描述方法执行后返回值. s 为对象的当前抽象值,$\bar{p}$为方法参数,t 为临时变量,r 表示结果. 这个规范的意义是,如果方法执行前前置条件成立,则执行后后置条件成立. 下面我们用例子来作进一步阐述. 本文例子均用 FOPL 语言编写.

例 1:整数栈类型 ISTACK

```
Type ISTACK
Csig:empty::ISTACK,push::INTEGER→ISTACK→ISTACK
Ssig:top::SELF→INTEGER,pop::SELF→ISTACK
Msig:isempty::SELF→BOOL
Axiom:/* use integer string as abstract value of stack */
    {true}push n s{r=n.s};
    {true}empty{r=space};
    {s≠space}pop s{s=t.r};
    {s≠space}top s{s=r.t};
    {true}isempty s{r=(s==empty)};
Endt
```

设 C 为一类定义,T 为类型定义,我们引入表示函数 f.

定义 6. 表示函数 f::R→S,将对象值的具体表示映射到抽象值集上.

我们用一逻辑公式来描述R,称为表示不变式.

定义 7. 表示不变式I是描述对象值的具体表示的集合的逻辑公式.

表示函数至少应对表示不变式描述的所有具体值表示有定义.

定义 8. C实现T,如果下列条件成立:

(1)在每个对象创立后,不变式即对其成立.

(2)T和C具有相同的构造子集.

(3)对C中每个操作f(构造子,选择子或方法)应满足$\{I\}f(\bar{p})\{I\}$,这里f可在T中出现,也可是C中超出T附加的.

(4)对T中每个操作规范$\{P\}f(\bar{p})\{Q\}$,C中有对应操作f,T$f和C$f具有相同参数数目、类型和结果类型,且在C中满足$\{P\circ f\wedge I\}f(\bar{p})\{Q\circ f\wedge I\}$这里$P\circ f$表示P中的抽象值用f到具体值表示的作用替代,如将P视为函数S→{true,false},则∘表示函数复合.

例2:ISTACK的数组实现

```
Class ASTACK
Typcof:ISTACK
Csig:empty::ISTACK,push::INTEGER→ISTACK→ISTACK
Ssig:top::ISTACK→INTEGER,pop::ISTACK→ISTACK
Msig:isempty::ISTACK→BOOL
Equation:empty=PAIR.new[INTEGER$0,a];a 为定长数组
        push n s=PAIR.new[(s.fst)+[INTEGER$1],(s.snd).update
                [(s.fst).+[INTEGER$1,a]]
        pop empty =empty;
        pop(push n s)=s;
        top empty = error;
        top(push n s)=n;
        isempty empty=BOOL$true;
        isempty(push n s)=BOOL$false
Endc
```

其中PAIR是整数和数组的记录类型,update是修改数组某位置内容的方法.为下面讨论的方便,我们用⟨0,a⟩表示empty,⟨i,a⟩表示非空栈,则我们可定义:

I≡ length(a)≥i≥0,length求数组长度

f⟨0,a⟩ =empty,f⟨i,a⟩ =a[i].f⟨i−1,a⟩

当新创立一个ASTACK对象时,有i=0,故I成立,对构造子push应满足:

$\{I\}$push n s$\{r=n.f\langle i,a\rangle\wedge I\}$

即: $\{I\}$push n s$\{r=a[i+1].f\langle i,a\rangle\wedge a[i+1]=n\wedge I\}$

$\{I\}$push n s$\{r=f\langle i+1,a\rangle\wedge a[i+1]=n\wedge I\}$

按push在ASTACK中定义,我们易看出push满足上规范.对选择子pop应满足:

$\{I\}$pop s$\{f\langle i,a\rangle=t.r\}$

即: $\{I\}$pop s$\{a[i].f\langle i-1,a\rangle=t.r\wedge I\}$

$\{I\}$pop s$\{a[i]=t\wedge r=f\langle i-1,a\rangle\wedge I\}$

也易看出ASTACK中pop满足此规范,同样易验证top和isempty.同时,对所有操

作,我们易验证 I 为不变式,因此,ASTACK 实现 ISTACK.

例 3:ISTACK 的表实现

```
Class LSTACK
Typeof:ISTACK
Csig:empty::ISTACK,push::INTEGER→ISTACK→ISTACK
Ssig:top::ISTACK→INTEGER,pop::ISTACK→ISTACK
Msig:isempty::ISTACK→BOOL
Equation:empty=LIST[INTEGER]$nil;
        push n s=LIST[INTEGER].new [n,s];
        pop empty=empty;
        ……
Endc
```

令 I≡l=nil ∨ l=cons n l′,f nil=empty,f(cons n s)=n.(f s),我们易验证 LSTACK 实现 ISTACK.

我们可以看到,上面的规范描述是独立于对象的内部结构的,但由于他们所规定的性质确实可以通过消息传递观察到,因此,抽象值集的选取是重要的,该集不应包含有太多信息,应该能够通过消息传递确定给定对象的抽象状态.同时,该集也不应含有不能作为对象抽象状态的元素.

3　子类型关系

T≤S,直觉上,似乎下面定义是足够的.

定义. 如对 S 中每个操作规范{P}f($\bar{p}$){Q},在 T 中都有对应规范{P′}f($\bar{p}$){Q′},使得后者蕴涵前者,即有 P→P′,Q′→Q,则 T≤S.

确实,在此定义下,T 的元素总可用于期望 S 元素的地方,然而,通常我们必须假定 S 和 T 应在不同抽象值集上规范.如整数包类型 IBAG 定义如下:

例 4:Type IBAG

```
Csig:empty::IBAG,push::INTEGER→IBAG→IBAG
Ssig:pop::IBAG→IBAG
Axiom:/* use Array as abstract value */
{true}empty{∀ i.b[i]=0};
{true}push n b{r[n]=b[n]+1∧∀ i(i≠n).r[i]=b[i]};
{∃ i.b[i]>0}pop b{r[n]=b[n]−1∧∀ i(i≠n).r[i]=b[i]}
Endt
```

这里 IBAG 的规范是定义在抽象值 b 上,b 是数组,b[i]表示 i 在包中出现次数,在抽象意义下,我们可允许 b 的元素和整数一样多.

按上面的子类型定义,我们无法导出 ISTACK≤IBAG,为此,我们引入变换函数.

定义 9. 变换函数 g.

对类型 S 和 T,其抽象值集分别为 A_S 和 A_T,如 T≤S,则 $g:A_T\to A_S$ 将 T 的抽象值映射到 S 的抽象值集上.

定义 10. T≤S.

(1)对T中的构造子c,S中也有对应构造子c使得:

·T$c的参数数目多于或等于S$c的参数数目.

·对S$c的每个参数$P_i$,类型为$T_i$,在T$c中对应的类型为S_i,有$T_i \leqslant S_i$,并有变换函数$\varphi_i: A_i^T \to A_i^S$

·对c的结果r,在T$c中类型为$T_r$,在S$c中类型为S_r,有$T_r \leqslant S_r$,并有变换函数$\psi: A_r^T \to A_r^S$

(2)对S中选择子或方法f,总存在T中对应f使得:

·f的参数数目在两个类型中是相同的

·对f的每个参数P_i,设其在T$f中类型为$T_i$,在S$f中类型为S_i,有$S_i \leqslant T_i$,并有$\varphi_i: A_i^S \to A_i^T$

·对f的结果r,设其在T$f中类型为$T_r$,在S$f中类型为S_r,有$T_r \leqslant S_r$,且有$\psi: A_r^T \to A_r^S$

(3)对S中选择子或方法规范:

$$\{P(s,p_1^S,\cdots,p_n^S)\}f(\bar{p}^S)\{Q(s,p_1^S,\cdots,p_n^S,r^S)\}$$

在T中有对应规范为:

$$\{P'(s',p_1^T,\cdots,p_n^T)\}f(\bar{p}^T)\{Q'(s',p_1^T,\cdots,p_n^T,r^T)\}$$

且下列蕴涵成立:

·$P(g(s'),p_1^S\cdots,p_n^S)\to P'(s',\varphi_1(p_1^S),\cdots,\varphi_n(p_n^S))$,即$P\circ g\to P'\circ\varphi_n\circ\cdots\circ\varphi_1$,

·$Q'(s',\varphi_i(p_1^S),\cdots,\varphi_n(p_n^S),r^T)\to Q(g(s'),p_1^S,\cdots,p_n^S,\psi(r^T))$,即$Q'\circ\varphi_n\circ\cdots\circ\varphi_1\to Q\circ g\circ\psi$

这里$g: A_T\to A_S$将T的抽象值映射到S的抽象值集上,。如前定义,由于谓词中参数的相互无关性,复合的顺序是不重要的.

(4)对S中构造子规范

$$\{P(s,p_1^S,\cdots,p_n^S)\}c((\bar{p}^S)\{Q(s',p_1^S,\cdots,p_n^S,r^S)\}$$

和T中的对应构造子规范

$$\{P'(s',p_1^T,\cdots,p_n^T,\cdots,p_{n+m}^T)\}c(\bar{p}^T)\{Q'(s',p_1^T,\cdots,p_n^T,\cdots,p_{n+m}^T,r^T)\}$$

有下面蕴涵式成立:

·$P(g(s'),\varphi_1(p_1^T)\cdots,\varphi_n(p_n^T))\to P'(s',p_1^T,\cdots,p_n^T)$,即$P\circ g\circ\varphi_n\circ\cdots\circ\varphi_1\to P'$

·$Q'(s',p_1^T,\cdots,p_n^T,r^T)\to Q(g(s'),\varphi_1(p_1^T),\cdots,\varphi_n(p_n^T),\psi(r^T))$,即$Q'\to Q\circ g\circ\varphi_n\circ\cdots\circ\varphi_1\circ\psi$.

需要指出的是,构造子的参数即是其类型说明中已列出的全体,而选择子和一般方法的参数则不能将对象本身算进去.如上面例子中,push有两个参数,类型分别为ISTACK和INTEGER,而pop,top,isempty均无参数.

为证明ISTACK是IBAG的子类型,我们令$g: A_T\to A_S$为:

g(s)[i]=k,k是i在s出现次数.

φ_i和ψ或为恒等函数、或为g.

对构造子push我们需证:

• true→true

• r=n. s→g(r)[n]=g(s)[n]+1∧∀ i(i≠n). g(s)[i]=g(r)[i]

对选择子 pop 需证:

• ∃ i. (g(s)[i]>0)→s≠space

• s=t. r→g(r)[t]=g(s)[t]−1∧∀ i(i≠t). g(r)[i]=g(s)[i]

以上验证是直接的. 从而知 ISTACK≤IBAG

基于上面的子类型定义,定义 8 中应做如下修改和推广:

(2)C 的构造子集是 T 的构造子集的子集.

(4)A. 对 T 中每个非构造子操作规范{P}f($\bar{p}$){Q},C 中也有一对应操作 f,它和 T $ f 具有相同参数数目,其参数 p_i 在 C 中类型为 T'_i,在 T 中类型为 T_i,$T_i \leqslant T'_i$;其结果 r 在 C 中类型为 T'_r,在 T 中类型为 T_r,$T'_r \leqslant T_r$;$\varphi_i: A_i^{T'} \to A_i^T$,$\psi: A_r^{T'} \to A_r^T$ 为变换函数,且满足规范:

$$\{P \circ \varphi_n^{-1} \circ \cdots \circ \varphi_1^{-1} \circ f \wedge I\} f(\bar{p}) \{Q \circ \psi \circ \varphi_n^{-1} \circ \cdots \circ \varphi_1^{-1} \circ f \wedge I\}$$

这是 φ_i^{-1} 为 φ_i 的逆函数,$\varphi_i^{-1} \circ \varphi_i = id$(恒等函数). ψ^{-1}亦然.

B. 对 T 中构造子规范{P}c($\bar{p}$){Q}和 C 中对应构造子 c,C $ c 的参数数目多于或等于 T $ c 的参数数目,对 T $ c 中每个参 p_i,在 T 中类型为 T_i,在 C 中类型为 T'_i,$T'_i \leqslant T_i$;其结果 r 在 T 中类型为 T_r,在 C 中类型为 T'_r,$T'_r \leqslant T_r$;$\varphi_i: A_i^{T'} \to A_i^T$,$\psi: A_r^{T'} \to A_r^T$ 为变换函数,且满足规范:

$$\{P \circ \varphi_n \circ \cdots \circ \varphi_1 \circ f \wedge I\} c(\bar{p}) \{Q \circ \psi \circ \varphi_n \circ \cdots \circ \varphi_1 \circ f \wedge I\}$$

定义 8 中(1)和(3)不需改变.

最后,我们可导出如下定理:

定理 1. 如果类 C 实现类型 T,T≤S,则 C 实现 S.

证明:C 实现 T,则有表示函数 $f: R \to A_T$ 和表示不变式 I. 要证明 C 实现 S,需验证定义 8 中(1)(2)(3)(4),其中(1)、(2)、(3)的证明是直接的,主要是(4)的证明.

A. 对 T 中非构造子操作规范{P}f($\bar{p}$){Q},在 C 中有操作 f,且有变换函数 $\varphi_i: A_i^T \to A_i^{T'}$,$\psi: A_r^{T'} \to A_r^T$,使得

$$\{P \circ \varphi_n^{-1} \circ \cdots \circ \varphi_1^{-1} \circ f \wedge I\} f(\bar{p}) \{Q \circ \psi \circ \varphi_n^{-1} \circ \cdots \circ \varphi_1^{-1} \circ f \wedge I\}$$

如 T≤S,则有变换 $g: A_T \to A_S$,$h_i: A_i^S \to A_i^T$,$h: A_r^T \to A_r^S$,使得对 S 中规范{P}f($\bar{p}$){Q}有 T 中对应规范{P′}f($\bar{p}$){Q′},且满足:

$$P \circ g \to P' \circ h_n \circ \cdots \circ h_1, Q' \circ h_n \circ \cdots \circ h_1 \to Q \circ g \circ h$$

$$\Rightarrow P \circ g \circ h_1^{-1} \circ \cdots \circ h_n^{-1} \to P', Q' \to Q \circ g \circ h \circ h_1^{-1} \circ \cdots \circ h_n^{-1}$$

$$\Rightarrow P \circ g \circ h_1^{-1} \circ \cdots \circ h_n^{-1} \circ \varphi_n^{-1} \circ \cdots \circ \varphi_1^{-1} \circ f \wedge I \to P \circ \varphi_n^{-1} \circ \cdots \circ \varphi_1^{-1} \circ f \wedge I$$

$$Q' \circ \psi \circ \varphi_n^{-1} \circ \cdots \circ \varphi_1^{-1} \circ f \wedge I \to Q \circ g \circ h \circ h_1^{-1} \circ \cdots \circ h_n^{-1} \circ \psi \circ \varphi_n^{-1} \circ \varphi_1^{-1} \circ f \wedge I$$

$$\Rightarrow P \circ g \circ (\varphi_n \circ h_n)^{-1} \circ \cdots \circ (\varphi_1 \circ h_1)^{-1} \circ f \wedge I \to P' \circ \varphi_n^{-1} \circ \cdots \circ \varphi_1^{-1} \circ f \wedge I$$

$$Q' \circ \psi \circ \varphi_n^{-1} \circ \cdots \circ \varphi_1^{-1} \circ f \wedge I \to Q \circ g \circ h \circ \psi \circ (\varphi_n \circ h_n)^{-1} \circ \cdots \circ (\varphi_1 \circ h_1)^{-1} \circ f \wedge I$$

因 C 实现 T,故有$\{P' \circ \varphi_n^{-1} \circ \cdots \circ \varphi_1^{-1} \circ f \wedge I\} f(\bar{p}) \{Q' \circ \psi \circ \varphi_n^{-1} \circ \cdots \circ \varphi_1^{-1} \circ f \wedge I\}$,

从而有:$\{P \circ g \circ (\varphi_n \circ h_n)^{-1} \circ \cdots \circ (\varphi_1 \circ h_1)^{-1} \circ f \wedge I\} f(\bar{p}) \{Q \circ g \circ h \circ \psi \circ (\varphi_n \circ h_n)^{-1} \circ \cdots \circ (\varphi_1 \circ h_1)^{-1} \circ f \wedge I\}$

其中 $\varphi_i \circ h_i : A_i^S \to A_i^{T'}$，$h \circ \psi : A_r^{T'} \to A_r^S$，令 $f' = g \circ f : R \to A_S$，I 不变，则我们可知 C $f 实现 S $f.

B. 对 T 中构造子规范 $\{P\}c(\bar{p})\{Q\}$ 和 C 中对应 c，有变换函数 $\varphi_i : A_i^{T'} \to A_i^T$，$\psi : A_r^{T'} \to A_r^T$，使得：

$$\{P \circ \varphi_n \circ \cdots \circ \varphi_1 \circ f \wedge I\}c(\bar{p})\{Q \circ \psi \circ \varphi_n \circ \cdots \circ \varphi_1 \circ f \wedge I\}$$

如 $T \leqslant S$，则有变换函数 $g : A_T \to A_S$，$h_i : A_i^T \to T_i^S$，$h : A_r^T \to A_r^S$，使得对 S 中规范 $\{P\}c(\bar{p})\{Q\}$ 和 T 中对应规范 $\{P'\}c(\bar{p})\{Q'\}$，满足：

$$P \circ g \circ h_n \circ \cdots \circ h_1 \to P', Q' \to Q \circ g \circ h_n \circ \cdots \circ h_1 \circ h$$

$$\Rightarrow P \circ g \circ h_n \circ \cdots \circ h_1 \circ \varphi_n \circ \cdots \circ \varphi_1 \circ f \wedge I \to P' \circ \varphi_n \circ \cdots \circ \varphi_1 \circ f \wedge I,$$

$$Q' \circ \psi \circ \varphi_n \circ \cdots \circ \varphi_1 \circ f \wedge I \to Q \circ g \circ h_n \circ \cdots \circ h_1 \circ h \circ \psi \circ \varphi_n \circ \cdots \circ \varphi_1 \circ f \wedge I$$

$$\Rightarrow P \circ g \circ (h_n \circ \varphi_n \circ \cdots \circ (h_1 \circ \varphi_1 \circ f \wedge I \to P' \circ \varphi_n \circ \cdots \circ \varphi_1 \circ f \wedge I$$

$$Q' \circ \psi \circ \varphi_n \circ \cdots \circ \varphi_1 \circ f \wedge I \to Q \circ g \circ (h_n \circ \varphi_n) \circ \cdots \circ (h_1 \circ \varphi_1) \circ h \circ \psi \circ f \wedge I$$

其中 $h_i \circ \varphi_i : A_i^{T'} \to A_i^S$，$h \circ \psi : A_r^{T'} \to A_r^S$，令 $f' = g \circ h : R \to A_S$，I 不变，我们可知 C $c 实现 S $c.

综上得证：C 实现 S.

注：不变式 I 为真是 C 实现 S 的条件之一，因此，I 在蕴涵式上的加入不会影响隐含式的真值.

4 结束语

本文讨论了 FOPL 语言的类型思想及其中继承和子类型关系的共存和相互关系. 其类型定义基于对象的行为规范，能更为灵活和自然地支持 ADT，并将类型也作为一阶对象处理. 继承和子类型关系分别构成了对象间的两个不同关系网层，对它们的分别处理对更深入地理解 OOP 语言及其语义是有益的.

FOPL 是一实验性语言，其原型系统已在 SUN386 工作站上用 C 语言实现.

参考文献

1 Goguen J A *et al*. Unifying functional, object-oriented and relational programming with logical semantics. Sri International, 1987.

2 Bobrow D *et al*. Commonloops: merging lisp and object-oriented programming. ACM SIGPLAN Notice, 1986, **21** (11).

3 梅宏. 函数式面向对象程序设计语言 FOPL—设计和实现[博士论文]. 上海交通大学, 1992.

4 Danforth S, Tomlinson C. Type theories and object-oriented programming. ACM Computing Surveys, 1988, **20** (**1**).

5 EATCS BULLETIN, No. 40, 1990.

6 Donahue J, Demers A. Data types are values. ACM Trans. on Programming Languages and Systems, 1985, **7**(**3**).

7 Cardelli L. A semantics of multiple inheritance. LNCS 173, 1984.

8 Mitchell J C. Polymorphic type inference and containment. LNCS 173, 1984.

9 Breazu—Tannen V *et al*. Inheritance as implicit coersion. Tech. Report MS-CIS-89-01, University of Pennsylvania, 1989.

10 Snyder A. Encapsulation and inheritance in object-oriented programming languages. ACM Conference on OOPSLA, 1986.

11 America P. Inheritance and subtyping in a parallel object-oriented language. ECOOP'87, 1987.

12 Donahue J. On the semantics of data types. SIAM Journal on Computing, 1979.

BEHAVIORAL SPECIFICATIONS AS TYPES

Mei Hong

(*Department of Computer Science, Beijing University, Beijing* 100871)

Sun Yongqiang

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

Abstract In object-oriented programming, the inheritance is one of the factors which induce semantic complexity. In this paper, the coexistence and relationship of inheritance, as a mechanism for sharing codes, and subtyping, which expresses specialization in functionality, are discussed. The concept of type, which states that types are behavioral specifications of objects, that is, the sets of named operations, is presented. Based on this concept, types, classes and subtyping relationship are described formally.

Key words Object-oriented programming, data type, inheritance, subtyping, behavioral specification.