广义继承及其在面向对象程序设计语言中的实现*

李宣东 郑国梁

(南京大学计算机科学系,南京210093)

摘要 本文给出一种包含多种继承行为的、非常灵活的代码复用机制——广义继承,通过给出其操作语义和一个支持类间子类型关系确认的类机制,为其在面向对象程序设计语言中的实现奠定了基础.

关键词 面向对象程序设计语言,继承,语义.

面向对象程序设计作为一种新型、实用的程序设计方法,强调数据抽象、信息隐蔽、模块化设计、可扩充性和代码复用等软件工程原则,特别有利于复杂软件系统的生成,被称为90年代的"结构程序设计方法".该方法的主要特征在于其对数据抽象和继承的支持.支持数据抽象和封装可抽象地定义对象的外延行为而隐蔽其实现细节,从而达到规范和实现的分离,有利于程序的理解、修改和维护,对系统快速原型和有效实现有着很大帮助;支持继承则可在原有代码的基础上构造新的软件模块,从而有利于软件复用.实现上述特征的有效途径之一是采用直接支持面象对象程序设计方法的面象对象程序设计语言开发程序.

在面向对象程序设计语言中,对象由数据(实例变量)和相关操作(方法)抽象封装而成,其行为通过操作展示,不允许外界对其内部状态的直接访问,操作的实现也对用户透明,消息传递是对象间唯一的交互方式,通过传递消息完成对象的创建和对象中操作的调用.类是具有相同内部状态和外部行为的对象族,由实例变量(状态)和操作(方法)集构成.对象通过对类的例化而产生.一个类可以通过继承拥有父类(超类)的状态和操作,而其本身的状态和操作又可以被子类所继承.Wegner 在文献[1]中将继承看成一种增量修改机制,他将可由继承体现的增量修改分为行为兼容、型构兼容、名兼容和取消 4 种形式.目前大多数面向对象程序设计语言仅支持行为兼容的增量修改,即其支持的继承机制只是一种行为兼容的增量修改机制.这种类型的继承具有两方面的含义:一是体现了增量复用性,即子类复用了父类的状态和操作,同时子类可以定义自己的状态和操作;二是体现了类间的子类型关系,即子类是父类行为规范的特殊化和求精,子类的实例可以代替父类的实例.由于子类型关系在

* 本文1994-02-24收到,1994-09-28定稿

作者李宣东,1963年生,博士生,主要研究领域为面向对象程序设计语言,软件工程.郑国梁,1937年生,教授,博士导师,主要研究领域为软件工程,软件开发环境.

本文通讯联系人:李宣东,南京210093,南京大学计算机科学系

面向对象程序设计中占有重要的地位，因而必须有一个方便、有效的方法用以确认类间的这种关系. 将类间的子类型关系通过继承体现出来则便于其确认. 但是，一方面类间的子类型关系受到制约(只有父类和子类间才可能有子类型关系)；另一方面继承作为一种增量修改复用机制的灵活性受到极大限制. 因而将类间的子类型关系和继承分离开来已逐渐成为一种共识[2-4]. Wegner 认为其定义的 4 种形式的增量修改在概念复用和代码复用方面均有各自独特的意义，因而在一种语言中同时支持这 4 种形式的增量修改是十分有意义的. 本文提出一种新的继承机制——广义继承，其包含了 Wegner 在文献[1]中给出的 4 种形式的增量修改行为，同时我们通过给出广义继承的操作语义和一个支持方便、有效地确认类间子类型关系的类机制，为在面向对象程序设计语言中实现广义继承奠定了基础.

1 广义继承

在面向对象程序设计语言中，类由实例变量和方法集构成，方法的实现对用户来说是透明的，他们只能通过规范来使用类. 一个类的用户分为两种：一种是作为类的实例接收消息并执行相应的操作；另一种则是子类，其继承父类的实例变量和方法. 一个类的规范可以分为 4 个层次：

(1) 名规范

一个类的名规范是指该类的实例变量名和方法名所构成的集合.

(2) 型构规范

一个类的型构规范是指该类的实例变量的型构和该类的方法的型构所构成的集合. 实例变量的型构是指其名和类型；方法的型构是指其名和可能有的参数和结果的类型.

(3) 行为规范

一个类的行为规范由定义在该类的实例变量上的不变式 I 和该类所具有的方法的规范的集合构成. 不变式 I 描述了该类的对象在生成期内必须保持的状态；一个方法 m 的规范可由一对前后置条件组成：$(prem, postm)$，前置条件 $prem$ 描述了方法 m 执行前，除去不变式 I 所描述的状态之外，该类的对象还应保持的状态. 后置条件 $postm$ 描述了方法 m 执行后该类对象所保持的状态，若在方法 m 的执行过程中，不变式 I 被破坏，则 $postm \Rightarrow I$. 类的不变式和类中方法的规范可以用一阶逻辑演算来描述[2,5].

(4) 代码规范

一个类的代码规范是指该类的实现，即该类本身.

以上 4 个规范层次在内容上显然存在如下关系：

$$\text{代码规范} \supset \text{行为规范} \supset \text{型构规范} \supsetneq \text{名规范}.$$

根据类的规范，我们可以给出如下 8 种类型的继承的定义.

定义 1. 代码继承

通过继承，子类拥有父类的实例变量和方法，同时子类可以定义自己的实例变量和方法，但必须保持父类的不变式，这种继承称为代码继承.

定义 2. 行为兼容的继承

通过继承，子类拥有父类的实例变量和方法，同时子类可以定义自己的实例变量和方法，但必须保持父类的不变式. 对父类中的任一方法，子类可以在保持其型构和语义的条件

下重定义,这种继承称为行为兼容的继承或子类型继承.

行为兼容的继承按在子类重定义其继承的父类方法时保持语义的方式又可分为强化后置条件的继承和保持后置条件的继承二类:

定义 3. 保持后置条件的继承

通过继承,子类拥有父类的实例变量和方法,同时保持父类的不变式.对父类中的任一方法 m(其规范为($Prem$, $postm$)),子类可以重新定义.设 m 在子类中重新定义后其规范为($Prem'$, $Postm'$),若有 $Prem \Rightarrow Prem'$, $Postm' \equiv Postm$,
则这种继承称为保持后置条件的继承.

定义 4. 强化后置条件的继承

通过继承,子类拥有父类的实例变量和方法,同时保持父类的不变式.对父类中的任一方法 m(其规范为($Prem$, $postm$)),子类可以重新定义.设 m 在子类中重新定义后其规范为($Prem'$, $Postm'$),若有 $Prem \Rightarrow Prem'$, $Postm' \Rightarrow Postm$,
则这种继承称为强化后置条件的继承.

显然保持后置条件的继承和强化后置条件的继承在子类重定义父类的方法时均遵守保持语义准则,保持后置条件的继承也是强化后置条件的继承,反之不然.在强化后置条件的继承下被子类重定义的父类方法不仅包含了其在父类中具有的功能,还可能增加了新的功能(例如,可能修改了子类中新增加的某些实例变量)[5],也即强化后置条件的继承允许子类在重定义其继承的父类方法时以扩充的形式改变其规范.

定义 5. 增量继承

通过继承,子类拥有父类的实例变量和方法,同时子类可以定义自己的实例变量和方法.对父类中的任一方法,子类可以重定义其实现,但不能改变其型构和规范,这种继承称为增量继承.增量继承和行为兼容的继承(子类型继承)的区别在于前者允许子类破坏父类的不变式,但不允许子类改变父类方法的规范;而后者不允许子类破坏父类的不变式,但允许子类以扩充的形式改变父类方法的规范.

定义 6. 型构兼容的继承

通过继承,子类拥有父类的实例变量和方法,同时子类可以定义自己的实例变量和方法.对于父类中的任一方法,子类在保持其型构的条件下可以重定义,这种继承称为型构兼容的继承.

定义 7. 名兼容的继承

通过继承,子类拥有父类的实例变量和方法,同时子类可以定义自己的实例变量和方法.对于父类中的任一方法,子类在保持其名的条件下可以重定义,这种继承称为名兼容的继承.

定义 8. 限制性继承

通过继承,子类拥有父类的实例变量和方法,同时子类可以定义自己的实例变量和方法.对父类中的任一方法,子类可以取消,也可以在保持其语义和型构的条件下重定义,这种继承称为限制性继承.

以上定义的行为兼容的继承,型构兼容的继承、名兼容的继承和限制性继承分别体现了 Wegner 在文献[1]中提出的行为兼容、型构兼容、名兼容和取消 4 种形式的增量修改行为.

一种在行为上概括了以上定义的 8 种继承机制的代码复用机制——广义继承，可定义如下：

定义 9. 广义继承

通过继承，子类可以全部或部分拥有父类的实例变量和方法，子类同时可以定义自己的实例变量和方法. 对继承的任一父类方法，子类可以在保持其名的条件下重定义，这种继承称为广义继承.

显然，广义继承包含前 8 种继承的行为.

2 广义继承的操作语义

上节给出的 9 种类型的继承除代码继承外均允许子类重定义其继承的父类方法. 然而，在子类中未被重定义的父类方法应该保持其在父类中的规范，不受其在父类中的实现的改变和在子类中重定义了的父类方法的影响，也即被继承的父类方法的规范在子类中被改变必须通过重定义显式说明，这一原则我们称之为继承的规范原则，其体现了方法的规范和实现分离的思想，有益于提高软件系统的可靠性、可重用性和易维护性. 这一原则一旦被违背，我们就无法根据型构规范和行为规范来使用一个类，通过继承的复用也就毫无意义了，从而面向对象程序设计语言的许多优点也将被掩盖.

目前面向对象程序设计语言中提供的继承机制及其语义仅支持代码继承、增量继承和保持后置条件的继承，其语义用来解释广义继承和其它类型的继承将违背继承的规范原则，从而导致错误. 这种继承机制及其操作语义的典型代表是 Smalltalk－80 中的继承机制和方法查询算法. Smalltalk 中的方法查询算法如下[6]：

消息接收者接到消息后，在自己所属类中查询与消息匹配的方法. 若找不到，则继续在超类中查找；若再找不到，则继续在超类的超类中查找，如此一直进行下去，直至找到为止.

若消息的接收者为 self，则查询从发送者所属的类开始进行.

若消息被发送至 super，则查询从包含 Super 的方法所在类的直接超类开始查找.

现设有类 A 和 B，B 是 A 的子类，B 通过广义继承拥有 A 的所有实例变量和方法. m 是 A 中的一个方法，其规范是($prem$，$postm$). 类 B 对方法 m 进行了重定义，改变了 m 的规范，使其成为($prem'$，$postm'$). n 是 A 中的另一个方法，其在 B 中没有被重定义，因而方法 n 在 A 中和 B 中的规范相同. 若方法 n 的实现定义中出现语句 self. m，按上述继承的操作语义，由于 B 中重定义的方法 m 过载了 A 中定义的方法 m，则类 B 中的方法 n 所实现的功能不一定是其规范所说明的功能，这取决于其在类 A 中的实现和方法 m 在类 B 中重定义后的规范，这样违背了继承规范原则，从而导致错误，这将对提高软件系统的可靠性、可重用性和易维护性均带来困难. 因而目前面向对象程序设计语言中所采用的继承的语义不能正确解释广义继承.

Wegner 在文献[1]中认为继承是一种增量修改机制，该思想源于 Cook[3]. 其将继承描述为父类实体 P(父类特征(实例变量和方法)的集合)加上增量修改实体 M(子类中新增的特征和重定义的父类特征的集合)生成子类实体 R(子类特征的集合)：

$$R=P+M,$$

继承的本质在于将 P 和 M 中的伪变量 self 归属于 R(即 self 代表子类实体)，如图 1 所示.

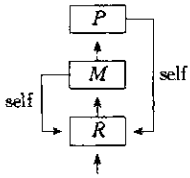
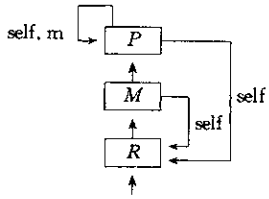


图1



m在M中重定义以改变其规范、型构或被取消

图2

目前给出的面向对象程序设计语言中继承的语义(操作语义[5]和指称语义[7])都是建立在此基础之上的.

广义继承的本质却有所不同,其本质在于将 M 中的伪变量 self 和 P 中的一部分 self 归属于 R,而将 P 中的另一部分 self 归属于 P 本身. 归属于 P 的 self 是指那些为调用在 M 中重定义以改变其规范、型构或被取消(未继承)的方法而使用的语句中出现的 self(如图 2).

在此基础上,可以建立广义继承的操作语义. 广义继承的操作语义仍可用 Smalltalk 中的方法查询算法来定义:

消息接收者接到消息后,在自己所属类中查询与消息匹配的方法. 若找不到,则继续在超类中查找;若再找不到,则继续在超类的超类中查找,如此一直进行下去,直至找到为止.

若消息的接收者为 self,若包含 self 的方法所属的类不是消息发送者所属的类,设其为 A,则从类 A 开始到消息发送者所属的类为止,寻找最后一个相对于 A 来说没有重定义与消息匹配的方法以改变其规范或型构的类,或者第一个其子类没有继承与消息匹配的方法的类,设其为 B,若 B 存在,则从其开始查找与消息匹配的方法,否则从类 A 开始查找与消息匹配的方法. 如果包含 self 的方法所属的类是消息发送者所属的类,则查询从消息发送者所属的类开始进行.

若消息被发送至 super,则查询从包含 super 的方法所属的类的直接超类开始进行.

3　支持确认类间子类型关系的类机制

由于在支持广义继承的语言中,子类型关系已和继承分离,因而需要有另外的机制支持类间子类型关系的确认. 我们采用的办法是将类间的子类型关系由类定义本身体现出来. 为能体现类间子类型关系,类定义必须包含两个接口:实例用户接口(CIF)和继承用户接口(IIF). 实例用户接口是该类的实例所能接收的消息所匹配的方法的集合,而继承用户接口则是该类可以被子类继承的实例变量和方法的集合. 由于两个类之间是否存子类型关系取决这两个类的不变式和类中方法的规范,因而在类的定义中必须给出不变式的定义和方法的规范的定义. 然而由于目前程序设计语言尚无法支持基于一阶逻辑的推理和演算,我们只能将类的不变式和方法的规范抽象成符号(类名)的集合. 通过以下形式在类中定义不变式:

$$INVARIANT\{A_1, A_2, \cdots, A_n, A\};$$

以上形式出现在类 A 的定义中表示类 A 保持类 $A1$,类 $A2$,…,类 An 中的不变式,同时保持自己(类 A)的不变式. 同样,类中方法的规范也用类名集合表示. 若类 A 中的方法 m 规范为:

$$\{A_1, A_2, \cdots, A_{n-1}, A_n\},$$

则表示类 A 中的方法 m 的规范包含了类 $Ai(i=1,2,\cdots,n)$ 中方法 m 的规范所表示的功能. 若类 A 继承了父类中的一个规范为 S_m(S_m 为类名集合)的方法 m,则方法 m 在类 A 中的规范 T_m(T_m 为类名集合)按如下规则定义:

(1) 若 m 在 A 中没有被重定义,则 $T_m=S_m$;

(2) 若 m 在 A 中被重定义以改变其实现,则 $T_m=S_m$;

(3) 若 m 在 A 中被重定义以扩充其规范(保持原有规范,增加新的功能),则 $T_m=S_m \cup \{A\}$;

(4) 若 m 在 A 中被重定义以改变其规范或型构,则 $(T_m \neq S_m) \wedge (T_m \supseteq S_m)$.

现设有类 A 和类 B,若以下条件成立:

(1) 类 B 的不变式 IB 包含类 A 的不变式 IA:$IB \supseteq IA$;

(2) 对类 A 的实例用户接口中的任一方法 m(规范为 SA_m),类 B 的实例用户接口中也存在相同型构的方法 m(规范为 SB_m),且 $SB_m \supseteq SA_m$;

则类 B 是类 A 的子类型.

以上定义的类机制使得类间的子类型关系可以有类定义本身确定,从而子类型关系和继承分离,并且使广义继承的操作语义简化如下:

消息接收者接到消息后,在自己所属类中查询与消息匹配的方法. 若找不到,则继续在超类中查找;若再找不到,则继续在超类的超类中查找,如此一直进行下去,直至找到为止.

若消息的接收者为 self,设在包含 self 的方法所属的类中,与消息匹配的方法的规范为 S(S 是一类名集合),则从消息发送者所属的类中开始查询规范为 S 且与消息匹配的方法.

若消息被发送至 super,则查询从包含 super 的方法所属的类的直接超类开始进行.

4 小 结

广义继承是非常灵活的代码复用机制,其包含了多种继承的行为. 在面向对象程序设计语言中支持广义继承的前提是子类型关系脱离继承并且继承遵守规范原则. 本文给出的广义继承的操作语义和支持确认类间子类型关系的类机制为在面向对象程序设计语言中实现广义继承奠定了基础.

参考文献

1 Wegner P, Zdonik S. Inheritance as an incremental modification mechanism or what like and isn't like. ECOOP'88, 1988.

2 America P. A behavioral to subtyping in object—oriented programming languages. ESPRIT Project 415 Doc No. 443, Jun. 1989.

3 America P. A parallel object—oriented language with inheritance and subtyping. ECOOP/OOPSLA, 1990.

4 Snyder A. Encapsulation and inheritance in object—oriented languages. OOPSLA, 1986.

5 Meyer B. Object—oriented software construction. Prentice—Hall, New York, 1988.

6 Goldberg A, Robson D. Smalltalk—80: the language and its implementation. Addison—Wesley, 1983.

7 Cook W, Palsberg J. A denotational semantics of inheritance and its correctness. Proc. of the OOPSLA ACM Conference, 1989.

GENERAL INHERITANCE AND ITS IMPLEMENTATION IN OBJECT—ORIENTED PROGRAMMING LANGUAGES

Li Xuandong　Zheng Guoliang

(*Department of Computer Science, Nanjing University, Nanjing* 210093)

Abstract　A flexible mechanism of rescuing code already written, which contains behaviors in many varieties of inheritance and is called general inheritance, is introduced in this paper. For its implementation in object—oriented language, its operational semantics and a class mechanism sopporting subtyping—check are given.

Key words　Object—oriented programming language, inheritance, semantics.