计算机科学1999Vol 26№. 3

协变多态计算中类型安全问题的研究*)

On Type Safety in Covariant Polymorphic Computing Model

王筱瑾 陶先平 柳 杨 吕 建

(南京大学计算机软件新技术国家重点实验室 计算机软件研究所 南京210093)

Abstract In strongly-typed object-oriented programming language, it's an important issue how to ensure type safety in covariant polymorphic computing model. Many resolutions have been proposed to both ensure type safety and provide natural express ability to programmers. In this paper, we analyse these proposals, then give one proposal implemented in our ND-Polya language.

Keywords Type safety, Contravariance, Covariance, Subtype, Multi-methods, Method-select

1. 引言

在O-O程序设计中,保证类型安全的反变计算不符合人们的思维习惯,而表达自然的协变计算又会引发类型的不安全,因而,由协变多态计算引出的类型安全问题一直是人们所关注的问题,人们提出了若干种办法来既保证类型安全,又提供自然灵活的表达能力,如Kim B. Bruce等在LOOM语言中提出的matching关系[3],较为常用的mulit-methods方法[1,4],Ingalls的precise typings方法[4],David L. Shang的参数化类的方法[5]等。

本文的主要工作就是在介绍协变与反变等概念的基础上,讨论了各种有代表性的解决方案,对其优缺点加以分析,并在此基础上提出了自行设计的ND-Polya语言的解决方案。

2. 类型安全、反变、协变

在O-O程序设计思想中,对象的一个重要属性是可替换性,即若一个对象提供了一组操作,则该对象可出现在任何只要求这些操作行为的上下文中。这种可替换性引进了多态,也带来了实现上的困难[1]。大多数O-O语言都使用继承来支持这种可替换性,使得继承类具有它的父类的所有行为,这样,子类的实例就应能被用于所有父类实例可出现的地方,即满足了可替换性。但是,这种类型系统存在一个基本缺陷:它是建立在继承这种实现技术上的,而不是直接以对象的操作集为基础,这就带来了一系列问题,如:一个消息可能会传给一个不能响应它的对象,即可替换性未被实现,这是一种类型错误。为避免这种错误,保证类型安全,大多数O-O语言引进了子类型概念。

O-O语言中,子类(Subclass)和子类型(Subtype)的概念并不相同,类的继承仅仅产生子类,而不一定产生子类型。子类型概念是为实现可替换性引进的,因而子类型关系定义为:父类型实例上成立的假定应该对子类型的实例也成立。也就是说,凡是在可以出现父类型实例的上下文中,子类型实例都可以出现,即子类型实例与父类型实例间满足可替换原则。在传统语言中,只要保证子类型的值集包含在父类型的值集中即可;而在O-O语言中,一个子类要成为父类的子类型,除了考虑类中的属性(子类属性的类型值集应包含父类属性的类型值集)外,还必须考虑类中的方法,只有子类中的方法能在任何上下文中代替父类中的相应方法,子类才能成为父类的子类型,满足了子类型关系,就满足了可替换原则。

设A是父类,B是A的子类,m_A是A中的一个

*)本项目受江苏省青年科学基金(BQ96005)、国家自然科学基金(69473038)和国家杰出青年科学基金(61525204)资助。
王筱瑾 硕士生,研究方向:面向对象语言及环境,软件Agent理论及技术。**陶先平** 讲师,博士生,研究方向:面向对象语言及环境,软件自动化及形式化。**柳杨** 硕士生。**吕建** 教授,博导,南京大学计算机软件新技术国家重点实验室主任,研究领域:软件自动化及形式化,面向对象语言及环境,软件Agent理论及技术。

方法，若 B 直接从 A 中继承了方法 m_B，则 m_B 能在任何上下文中替代 m_A，没有类型不安全问题；若 m_B 被重定义了，则只有 m_B 可在任何上下文中代替 m_A，即 B 必须是 A 的子类型，才能保证类型的安全，如下例。

```
class point{
    real x,y;
    boolean equal(point p){
      return((x==p.x)&&(y==p.y));
    }
    boolean break (point p){
      return(p.equal(self));
    }
}
```

```
class colorpoint extends point{
    String c  // 点的颜色
    boolean equal(colorpoint p).
      return((x==p.x)&&(y==p.y)
&&          (c==p.c));
    }
}
```

图1　类型不安全的例子

point 为二维点，colorpoint 为有色二维点，是 point 的子类。两个类中都有 equal 方法，用来比较该类的实例对象和参数 p 是否是同一点。因为类 colorpoint 中多一个属性 c，所以 $equal_{colorpoint}$ 方法重定义了 $equal_{point}$ 方法，由上述，$equal_{colorpoint}$ 方法必须在任何上下文中都可代替 $equal_{point}$ 方法，才能保证类型安全。上例中，$equal_{colorpoint}$ 方法只能处理 colorpoint 类型的参数，不能处理所有 point 类型的参数，因而是类型不安全的。譬如，通过类 point 中方法 break 对 eqnal 的使用：

```
point pl=new point();          colorpoint cpl=new colorpoint();
pl.break(cpl);
```

最后一句会调用 colorpoint 中的 equal 方法，即 cpl.equal(pl)，pl 的类型是 point，比 $equal_{colorpoint}$ 方法的参数类型 colorpoint 大，从而引发一个类型错误。这正是因为 $equal_{colorpoint}$ 方法不能在任何上下文中代替 $equal_{point}$ 方法所引起的。总之，为保证类型安全，m_B 的参数类型必须是 m_A 的参数类型的父类型，而 m_B 的返回类型是 m_A 返回类型的子类型。

前面曾谈到，子类型关系定义为：父类型实例上成立的假定应该对子类型的实例也成立。设函数 f：S→T，函数 g：S′→T′，若要 f 是 g 的子类型（记为 f<=g，即 S→T<=S′→T′），即要求 f 可在任何可以出现 g 的地方出现，这就意味着 f 的结果 T 必须能代替 T′，即 T<=T′；并且 f 可以处理 g 所能处理的所有参数，即 S′<=S，这就是反变规则。也就是说，反变规则能保证子类型关系。

在目前 O-O 语言的继承关系中，子类方法重定义父类方法时，可以有反变、协变和不变三种规则，分别表现如下：

设子类 B 中的方法 m：S→T，父类 A 中的方法 m：S′→T′，若 S′<=S，T<=T′，则称 B 中的 m 反变重定义 A 中的 m；若 S<=S′，T<=T′，则称 B 中的 m 协变重定义 A 中的 m；若 S=S′，T<=T′，则称 B 中的 m 不变重定义 A 中的 m。

反变或不变规则可保证类型安全，但它们在语义表达上极不自然。协变规则（如 Eiffel 语言、O_2 语言）则自然，表达力强，且符合程序设计的精化思想，但会引起类型不安全。本文将介绍解决这个矛盾的几种典型方法，并结合自己的工作提出了我们的方法。

binary 方法是类中的一种特殊的方法，即该方法含有一个本类所对应类型的参数[4]，如类 point 中的 equal(point p)方法。由于 binary 方法的参数类型与本类的类型是一样，所以含有 binary 方法的父类和子类之间不易保持子类型关系，从而影响类型的安全。

3. 几种解决方案

3.1　matching 关系

Kim B. Bruce 等在静态类型 O-O 语言 LOOM 中去掉了子类型关系，提出了一种新的 matching 关系[3]，matching 关系用"<#"表示，比子类型关系更一般、更弱。在 LOOM 中，类定义中的 MyType 即指本类所对应的类型。在此基础上，matching 关系定义如下：

设 A 和 B 是对象类型，B<#A 当且仅当 A 中的所有方法都要以相同的类型出现在 B 中，即 B 是通过加新的方法到 A 中获得的。

上述定义中的"相同"是指字面上的相同，包括 MyType 形式的。例如 point 中的 equal(MyType p) 和 colorpoint 中的 equal(MyType cp)，前后 MyType 代表不同的类型但是"相同"的，这时 colorpoint<#point。

由于去掉了子类型关系，LOOM 中规定：被继承方法重载时不允许改变其参数类型。由此，match-

ing 关系的一个重要特征是：如果两个类是父子类，那么它们相应的对象类型满足 matching 关系。

在 OO 语言中，子类型允许一种类型的值"冒充"其父类型的值，这有两种情况：一是可以把函数应用于其参数所声明类型的子类型上，二是可用于变量的赋值。在 LOOM 语言中，因为没有了子类型，上述两种情况的处理较为困难，因而引进了一个新类型 #A(A 是一个对象类型)，叫 Hash types，它能解决"冒充"问题，Hash types 定义如下：如果一个变量或形参被声明为是 #A 类型的，那么它能接收任何与 A 相 matching 的类型的值。

LOOM 的静态类型系统保证了程序在运行期间不会有类型错误，是类型安全的。但是，LOOM 语言中的 matching 关系和 Hash types 有如下三个缺陷：一是一个变量或形参要赋不只一种类型的值时，它必须声明为 Hash types，否则只能赋它本身类型的值；二是若一个消息的接受者是 Hash types，不能传给它一个 binary 消息。三是它对不是 binary 方法的协变情况依然难以处理。

3.2 Multi-methods 方法

G. Castagna 等人提出了一种使协变规则类型安全的办法，即 multi-methods 方法[1]：一个 multi-methods 方法是一组方法体的集合，这组方法体对应于同一个方法名(即同一个消息名)。该集合中的每一个方法体都叫该 multi-methods 方法的一个分支。

在传统的 OO 语言中，运行时刻时消息使用哪一个方法，仅由接收者的类决定；而在 multi-methods 方法中，方法体的选择还依赖于参数的类，multi-methods 主要有两种实现办法：一是 CLOS 语言中的实现办法，二是封装 multi-methods 方法，这里主要介绍封装 multi-methods 方法。

以 colorpoint 类为例，colorpoint 中的 equal 方法不能处理 point 类的参数，我们可在该类中增加一个方法：

```
class colorpoint extends point {
    String c;
    boolean equal(colorpoing p) {
        return((x==p.x)&&(y==p.y)&&(c==p.
        c));
    }
    boolean equal(point p) {
        return((x==p.x)&&(y==p.y));
    }
}
```

图2 multi-methods 方法的例子

该类中的两个同名 equal 方法体就组成了一个 multi-methods 方法，它们成为该 multi-methods 方法的分支。

这种 multi-methods 方法与 C^{++} 的方法重载是本质不同的，C^{++} 中对重载方法体的选择是在编译时刻完成的，而 multi-methods 中方法体的选择是在运行时刻完成的，上例中，根据运行时刻 p 的类型决定调用哪个分支。

那么，如何确定 multi-methods 方法的类型及它们之间的子类型关系呢？Multi-methods 的类型定义为它所有分支的类型的集合，在 colorpoint 类中，equal 的 multi-methods 类型为{colorpoint->boolean，point->boolean}，类型集合之间的子类型关系定义为：设 G、H 是两个类型集合，如果对 H 中的每个类型 A，在 G 中都存在一个类型 B<=A，则 G<=H，即 G 是 H 的子类型[1]，根据这个定义可得：

{colorpoint->boolean，point->boolean}<=
{point->boolean}

所以，colorpoint 是 point 的子类型，因而类型是安全的，也就是说，multi-methods 方法可以很好地处理 binary 方法。

通常，在单继承的情况下，multi-methods 中的分支数为2。一个分支是从父类协变地重定义得到的方法，另一个分支是以该方法在继承层次中第一次出现时的参数为参数的方法，它足以处理所有可能出现的参数情况。因此，这种方法并未增加多少程序量。

多继承情况下的 multi-methods 方法比较复杂。详细讨论见文[1]。

在 CLOS 语言中，multi-methods 方法的实现稍有不同。它们不是定义在某个类中，而是作为全局定义的函数。它的缺陷是缺乏封装性，对象的方法不再封装在对象的类定义中，这不符合 O-O 程序设计原则。

封装 multi-methods 方法的实现可以有两种途径：一是修改编译器，由编译器自动增加 multi-methods 中的分支方法，如对 O_2 语言编译器的修改[5]；二是在源程序级提供一种实现 multi-methods 的机制，如 John Boyland 等提出的寄生方法(Parasitic Methods)[5]。

3.3 Precise typings 方法

Ingalls 提出的一种所谓在单分配语言中模拟 multi-methods 的办法，实际上是 precise typings 方法的应用[4]，但是，这种方法会引起程序量的大幅度增加；而且，子类的增加可能会引起父类的修改。限

于篇幅，本文不详细介绍。

3.4 参数化类的方法

David L. Shang 认为，子类中对协变的要求实际上是因为父类的定义中没有描述正确的类型依赖，因而通过采用参数化的类来描述程序中的各种类型依赖关系，协变问题就可被完全解决，从而保证类型安全[5]。

举例说明：某个引擎类对应于某个燃油类，该引擎中的添加燃油方法（AddFuel）只能添加该类燃油，则该方法的形参与引擎类间有类型依赖关系：

```
class Engine#(type FuelType:Oil)
{  AddFuel(fuel:FuelType);}
```

类参数 FuelType 表示 Engine 类所能接受燃油的类型，对类参数 FuelType 进行重受限将产生新的子类，形参 fuel 和类 Engine 之间的类型依赖关系被准确地描述出来了，这样，就不会产生类型不匹配的错误了。

4. ND-Polya 语言中的解决方案

上节介绍了四种保证类型安全的办法及它们各自的优缺点。我们认为，一种较实用的办法是在现有语言的基础上，添加一定的机制，来达到我们的目的。

Matching 方法和参数化类的方法都提出了一些现有 O-O 语言所没有的全新的概念，需借助崭新的语言来实现。precise typings 方法在定义子类时，往往要修改父类（可能还有父类的父类），程序量也较大。Multi-methods 方法可以通过对已有 O-O 语言进行改造来实现，较为可行。在 multi-methods 方法的两种实现办法中，又以封装 multi-methods 方法为佳。

从实用的角度出发，我们在 ND-Polya 语言中给出了我们的解决方案。ND-Polya 语言是基于目前较为流行的 Java 语言的。Java 语言为避免类型不安全，采用了不变重定义规则，语义不够自然，对程序员的限制较大，从图3的例子中可以看出。为了达到类型安全且表达自然的目的，我们仅对 Java 语言的类型系统及方法搜索进行了修改，有利于用户掌握和使用。

4.1 基本思想

首先应明确的是，协变反映了程序设计的精化思想，反变反映了动态计算中的行为子类型替换。基于这点，我们在 ND-Polya 语言中给出了两套类型系统。

（1）静态类型系统：变量的静态类型为声明该变量的类类型。我们允许方法协变重定义，符合分类精化的程序设计思想。

（2）动态类型系统：变量 a 的动态类型是一个类型链，它从生成 a 所代表对象的类类型开始，沿继承链上溯，到 a 的静态类型终止，由这条链上所有的类类型构成。在该类型链中，所有同名方法实际上构成了一个 multi-methods 方法的各个分支。由3.2节的讨论可知，ND-Polya 的动态类型系统是类型安全的。

程序编制时，程序员使用静态类型系统，运用协变重定义规则来定义方法，符合人们的思维习惯，表达自然，程序运行时，对多态变量 a 的方法选择是根据消息的实参在它的动态类型上进行方法搜索完成的：从 a 的当前类型开始，用实参选择方法，若在当前类型中匹配到方法，则成功；否则沿动态类型的链向上搜索，直至找到匹配的方法。

4.2 举例说明

```
class Food……
class Meat extends Food ……
class Animal {
    void eat(Food f)……
}
class Tiger extends Animal {
    void eat(Meat m)……
}
public class test {
    public static void main(String argv[]) {
        Animal x=new Animal();
        Tiger y=new Tiger();
        Meat m=new Meat();
        x=y;
        x.eat(m)
    }
}
```

图3 Java 语言和 ND-Polya 语言对比

Java 语言中，Tiger 类从父类不变地继承了 eat(Food f)方法，与 eat(Meat m)形成了方法重载，x.eat(m)在编译时进行选择，调用的是 eat(Food f)方法，这显然是语义不精确的。

ND-Polya 语言中，x=y 后，x 的动态类型是由 Animal 和 Tiger 组成的一个类型链，当前类型是 Tiger。执行 x.eat(m)时，方法的选择是在类型链中向上搜索而得，因而选择 Tiger 中的 eat 方法，实现了程序设计精化的思想。

我们提供一个预编译器把 ND-Polya 程序转换为 Java 程序，在预编译过程中，对 ND-Polya 程序进行一些改动，以实现后面的方法搜索。在此不详细讨论了。

（下转第40页）

步2　选择交配对使在平均性能之上的个体得到更多的子代；

步3　交叉和变异，产生新子代；

步4　如果不满足终止准则，则回到步3；否则，停止进行。

这里，具体的遗传算法的各种算子设置为：

1)编码方式：采用排列编码方式。

2)交叉算子：采用修改的分类匹配算子，其工作原理：它从捐送串上随机选取一个子串，接着把包含在被选子串上的元从接受串上删除，然后再把这个子串拷贝到接受串上。图3是这个交叉算子的一个实例。交叉概率设定为0.80。

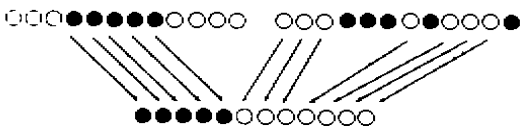

图3　修改的分类匹配的实例

3)变异算子：应用交换排序变异算子，在染色体串上随机选定两个位置，将这两个位置间的顺序逆排。变异概率设定为0.01。

4)选择机制：适应值函数 f 定义为：

$$f=f_{max}+(f_{max}-f_{min})/M \qquad (3)$$

其中，f_{max}和 f_{min}分别表示当前代中染色体的最大距离和最小距离，M 是群体规模。选择概率定义为：

$$P(i)=(f-f(i))/(M\times f-\sum_{j=1}^{M}f(j)) \qquad (4)$$

其中，$f(i)$是染色体 i 的路径距离。

5)群体规模定为 $M=30$。

6)结束准则：不同规模的 TSP 问题，采用不同的最大迭代代数。

4　仿真结果与分析

本节针对不同规模的 TSP 问题，通过大量的比较实验研究，来考察具有搜索空间平滑技术的遗传算法的搜索能力，以下给出几个典型的仿真结果，图4(略)是两种遗传算法在相同的参数设置下，针对 OliverTSP 问题[10]的性能比较，一种是本文所提出的具有空间平滑技术的遗传算法(GAS)，另一种是文[11]所提出的2-opt 与遗传算法所构成的混合遗传算法(HGA)。图5(略)是 GAS 所求得的 OliverTSP 问题的最优解的城市遍历图。图6(略)和图7(略)分别是48城市的 TSP[11]的寻优结果和最优解的城市遍历图，而图8(略)和图9(略)分别是 Grot442城市 TSP[11]的寻优结果和最优解的城市遍历图。从仿真结果不难看出，GAS 具有较强的搜索能力，它能够有效地跳出"局部"最优解的陷阱，获得问题的近优解。

参 考 文 献

1　Chen Xiong, Xu Xinhe. New Heuristics with Search Space Smoothing for Rolling Planning. In: Proceeding of CWCICIA, Xian, 1997

2　Gu J. Local Search for Satisfiability Problem IEEE Trans. on System, Man and Cybernetics, 1993, 23(4): 1108～1129

3　Lin S. Computer Solutions of the Traveling Salesman Problem. Bell System Tech. J., 1965, 44: 2245～2269

4　Lin S, Kernighan, B W. An Effective Heuristic Algorithm for the Traveling Salesman Problem. Operations Research, 1973, 21: 498～516

5　Oliver I M D, et al. A study of permutation Crossover Operators on the TSP. In: Proc. 2nd Int. Conf. on Genetic algorithm and Their Applications. Cambridge, MA, 1987. 224～230

6　陈雄：[博士论文]. 东北大学，1998

(上接第18页)

ND-Polya 语言中的这种机制具有如下一些优点：(1)它提供了协变多态计算模型，语言的表达能力较强，符合人们的思维习惯，使用自然、灵活。(2)它保证了类型安全。(3)方法在类型格上的搜索由系统自动完成，程序员无额外负担。(4)保证了修改子类时，不须改动父类。(5)保证了语言的模块性。

参 考 文 献

1　Boyland J, Castagna G. Type-Safe Compilation of Covariant Specialization: A Practical Case. Nov. 1995

2　Castagna G. Covariance and Contravariance: Conflict without a Cause. ACM Transactions on Programming Languages and Systems. 17(3)

3　Bruce B, et al. Subtyping is not a good"Match" for object-oriented languages. Nov. 1996

4　Bruce K, et al. On Binary Methods. Theory and Practice of Object Systems 1(Number: 3)

5　Boyland J, Castagna G. Parasitic Methods, An Implementation of Multi-Methods for Java.

6　谢高严，徐永森，詹志远，单成. TransFrame 语言中类型依赖关系及其实现. 微型计算机，1997, 17(增刊)

7　McDermud J A. Software Engineer' s Reference Book. Butterworth-Heinemann Ltd, 1991