ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.20, No.10, October 2009, pp.2617−2627
doi: 10.3724/SP.J.1001.2009.03656

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种基于类型传播分析的泛型实例重构方法*

陈 林[1,2,3+], 徐宝文[1,2,3], 钱 巨[1,4], 周天琳[1,3], 周毓明[1,2]

[1](南京大学 计算机软件新技术国家重点实验室,江苏 南京 210093)

[2](南京大学 计算机科学与技术系,江苏 南京 210093)

[3](东南大学 计算机科学与工程学院,江苏 南京 210096)

[4](南京航空航天大学 信息科学与技术学院,江苏 南京 210016)

Refactoring Generic Instantiations Based on Type Propagation Analysis

CHEN Lin[1,2,3+], XU Bao-Wen[1,2,3], QIAN Ju[1,4], ZHOU Tian-Lin[1,3], ZHOU Yu-Ming[1,2]

[1](State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China)

[2](Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China)

[3](School of Computer Science and Engineering, Southeast University, Nanjing 210096, China)

[4](College of Information Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China)

+ Corresponding author: E-mail: lchen@nju.edu.cn

Chen L, Xu BW, Qian J, Zhou TL, Zhou YM. Refactoring generic instantiations based on type propagation analysis. *Journal of Software*, 2009,20(10):2617−2627. http://www.jos.org.cn/1000-9825/3656.htm

Abstract: Refactoring generic instantiation is valuable for improving reusability and type safety of software. Most of the existing approaches of refactoring legacy code are not suitable for on-line and persistent refactoring because of their complexity. This paper proposes an instantiation refactoring approach for Java programs based on an extended variable type analysis algorithm. A generic type propagation graph is constructed, and new constructs used to express generic type analysis are added to the graph, so it is suitable to do a generic variable field sensitive type analysis. The paper also discusses how to use alias information to improve the refactoring. The case study shows that the results are satisfactory.

Key words: generic; software refactoring; type propagation analysis; alias analysis; software reuse

摘 要: 重构泛型实例有利于提高软件的复用性和类型安全,但现有重构方法的时间复杂度较高,不适用于即时持续的重构.分析了变量类型传播分析方法在重构中的不足,提出了一种改进的泛型变量类型传播分析方法.该方法通过引入一种可以描述复杂参数化类型关系的泛型类型传播图,以复制节点的方式实现泛型变量属性敏感的类型分析,并通过解决别名问题来提高分析的精度.实例研究表明,可以在与程序规模呈近似线性增长的时间复杂度内实施重构,取得了较满意的效果.

关键词: 泛型;软件重构;类型传播分析;别名分析;软件重用

中图法分类号: TP311 **文献标识码**: A

* Supported by the National Natural Science Foundation of China under Grant Nos.90818027, 60633010 (国家自然科学基金); the National High-Tech Research and Development Plan of China under Grant No.2009AA01Z147 (国家高技术研究发展计划(863))

Received 2008-09-24; Revised 2009-2-16; Accepted 2009-05-21

泛型以类型参数化的形式提供了编译时的多态,易于实现各种类型安全的策略,得到了工业界的广泛认可.使用泛型库的 Java 程序,由编译器作静态类型检查来保证容器内元素的类型一致性,提高了程序的可读性和类型安全性.然而,仍然存在大量使用非泛型库的 Java 遗产软件,以面向对象的方式实现容器,在操纵容器内元素时,往往需要对元素对象做类型转换操作.这些代码不仅破坏程序的可读性,也易引发运行时类型错误.为了充分利用泛型的优点,提高程序的类型安全性,可以对使用非泛型容器的代码进行泛型实例的重构.重构泛型实例是指给定泛型类或泛型方法,确定使用它们的客户代码在声明泛型类变量及实例化泛型类对象时所用的类型实参,从而将客户代码重构为泛型代码.

然而,现有的重构方法多先使用上下文敏感的分析算法,再进行迭代的类型约束求解,才能获得泛型实例所使用的类型实参.这些方法的时间复杂度很高,均为指数级.本文提出了一种兼顾效率和精确度的泛型实例重构方法,通过构造泛型类型传播图进行泛型程序的类型分析.一方面,设计了表示类型参数的节点和表达类型约束的边,以便能够从图中直接确定泛型实例的类型实参;另一方面,通过复制节点的方式实现了泛型变量属性敏感的分析,在类型传播的同时进行别名分析,提高了类型分析的精度,可以在较低的时间复杂度内取得满意的效果.

1 泛型实例重构问题

使用泛型容器可以有效地提高程序的类型安全性,提高代码可读性,改善代码质量.图 1 给出了使用非泛型容器和泛型容器的不同实现代码,显示了两者的区别.

```
1   class Cell{
2     Object value;
3     Cell(Object value) {set(value);}
4     void set(Object value) {this.value=value;}
5     Object get() {return value;}
6     void replaceValue(Cell that) {
7     this.value=that.value;}
8   }
```

(a)

```
9   class Cell<V extends Object> {
10    V value;
11    Cell(V value) {set(value);}
12    void set(V value) {this.value=value;}
13    V get() {return value;}
14    <U extends V> void replaceValue(Cell<U>that) {
15    this.value=that.value;}
16  }
```

(b)

```
17  static void example() {
18    Cell c1=new Cell(new Float(0.0));
19    Cell c2=new Cell(c1);
20    Cell c3=(Cell)c2.get();
21    Float f=(Float)c3.get();
22    Float f2=(Float)c2.get(); //error but can be compiled
23    Cell c4=new Cell(new Integer(0));
24    c4.replaceValue(c1);
25  }
```

(c)

```
26  static void example() {
27    Cell<Float>c1=new Cell<Float>(new Float(0.0));
28    Cell<Cell<Float>>c2=new Cell<Cell<Float>>(c1);
29    Cell<Float>c3=c2.get();
30    Float f=c3.get();
31    Float f2=c2.get(); //compile error!
32    Cell<Number>c4=new Cell<Number>(new Integer(0));
33    c4.replaceValue(c1);
34  }
```

(d)

Fig.1 An example of refactoring generic instantiations

图 1 泛型程序重构实例

在非泛型容器实现中,使用 *Object* 声明容器内元素 *value*,成员方法 *set*(),*get*()和 *replaceValue*()用于操纵该元素,如图 1(a)所示.图 1(b)是相应的泛型版本容器实现.若使用非泛型容器 *Cell*,当从容器中获取元素时,均需使用类型转换操作(语句 20~语句 22),且编译器无法确定所取元素的类型,程序员必须对所取元素的类型负责,保证该类型转换操作的正确性,如图 1(c)所示.语句 22 能够通过编译器检查,然而这是一个错误,会抛出运行时异常.若使用泛型容器,则无此问题(语句 29~语句 31),如图 1(d)所示,不仅消除了类型转换操作,同时可由编译器保证取元素操作的类型正确性.语句 31 的错误可由编译器检测出来,从而提高了程序的类型安全性.

由实例也可看出,只要确定了泛型变量声明点及实例化点的类型实参,即可将它们重构为使用泛型容器的

代码.因此,可以采用类型分析的方法实施重构.重构是一种需要持续进行、及时反馈的行为,对效率的需求非常迫切.时间复杂度较低的方法,更容易被程序员所接受,有利于重构的有效实施.目前的泛型实例重构方法大都先使用传统的笛卡尔乘积算法(Cartesian product algorithm,简称 CPA)进行上下文敏感分析[1],其时间复杂度较高,为指数级.使用 CPA 分析后,还需迭代作类型约束求解,再选取合适的类型作为类型实参,效率较低.若要提高重构的效率,同时保证重构精度,就必须选用更合理的方案.变量类型传播分析(variable type analysis,简称 VTA)是一种复杂度与程序规模呈线性增长的高效类型分析方法[2],效率很高.我们在 VTA 基础上,提出了一种更为高效的泛型实例重构方法.

2 泛型程序类型传播分析

通常,精度高的类型分析方法效率较低,而效率高的方法往往精度无法满足用户的需求.本节主要介绍如何在高效的 VTA 方法上进行扩充,以使该方法适用于泛型实例重构.

2.1 变量类型传播分析

VTA 的主要思想是通过赋值语句(包括方法参数传递等隐式赋值)将类型从对象创建点传播至各个变量中,通常使用类型传播图(type propagation graph,简称 TPG)来实现[2].类型传播图使用节点表示属性、方法形参、临时变量和方法的返回值等,并关联它们的可能类型,使用边表示由于赋值、方法调用、方法返回等语句引起的类型传播,模拟了程序中所有可能的赋值.

定义 1. 称类型 A 能够到达变量 u,当且仅当程序中存在一条执行路径,始于形如 v=*new* A()的构造函数调用,通过某条形如 $x_1=v,x_2=x_1,\ldots,x_n=x_{n-1},u=x_n$ 的赋值链.这里,u,v,x_i 表示变量,赋值操作包括由方法调用或方法返回引起的隐式赋值.

定义 2. 在类型传播图中,经过类型传播后,图中每个节点 n 都将关联一组类型,称为 n 的可达类型,记作

$$\mathcal{R}\_type(n).$$

VTA 方法可以分为下面两个步骤:首先构造类型传播图 TPG,为类、属性域、方法、方法参数、返回值、局部变量等添加对应节点,并为所有赋值语句(包括方法参数传递等隐式赋值)生成对应的边;然后传播类型,使用实例对象类型对节点进行初始化,分两阶段进行类型传播:先查找并合并强连通图,其可达类型为构成强连通图的所有节点可达类型的合集,然后按拓扑顺序遍历类型传播图一次以传播类型.这样可以保证类型传播的时间复杂度与程序规模呈线性增长.

使用 VTA 方法对程序分析后,每个变量将关联一个可能类型集,即该变量的可达类型.

2.2 泛型程序类型传播分析

在泛型实例重构中使用 VTA 对泛型程序进行类型分析存在两个问题:首先,VTA 是针对非泛型程序的,不考虑类型参数的传播,使得在获得变量的可能类型集后,仍无法直接求解泛型变量的类型实参,需通过对泛型变量的每个方法调用点作进一步的类型约束求解[3],确定类型实参.然而,这个过程需要反复迭代,时间复杂度较高.其次,VTA 是一种基于属性的类型分析方法,同一类型的不同实例的同名属性共享一个节点,无法区分不同泛型变量使用的类型实参,会给重构的精度带来损失.

图 2 给出了使用 VTA 方法为泛型类 *Cell* 构造的类型传播图.属性节点 *value* 标记为 *Cell.value*.对于每个方法 *Cell.m*,其形参 *p* 标记为 *Cell.m.p*,*Cell.m.ret* 表示该方法的返回值.*Cell.this* 用于表示方法的隐含参数.图中的边对应了赋值语句和方法参数传递.对应于图 2 所示的泛型类 *Cell*,语句

$$Cell\ c1=new\ Cell(new\ Float(0.0).$$

经过 VTA 类型传播分析,可以得到变量 $c1$ 和 $c1.value$ 的可达类型:

$$\mathcal{R}\_type(c1)=\{Cell\},$$

$$\mathcal{R}\_type(c1.value)=\{Float\}.$$

在图 2 所示泛型类 *Cell* 的类型传播图中,并没有类型参数 V 对应的类型节点,无法从中直接确定 V 的类型,

需要对泛型变量的每个方法调用点进行类型约束分析.

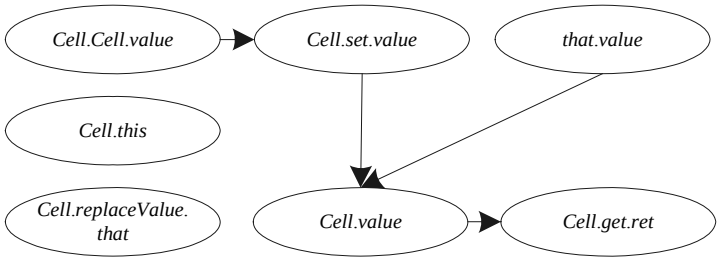

Fig.2 A type propagation graph constructed by VTA

图 2 使用 VTA 方法构造的类型传播图

为此,我们在 VTA 的基础上提出了一种改进的泛型程序类型传播图(generic type propagation graph,简称 GTPG),并使用该传播图进行类型分析.该方法的主要技术难点在于:① 如何表达类型参数的传播,从而直接确定类型实参的类型,避免迭代约束求解;② 如何避免基于属性的分析不区分不同泛型变量使用的类型实参的问题,提高分析精度.

为了解决第 1 个问题,我们在泛型类型传播图中给泛型变量及其类型参数设计了相应的节点和边.首先,在泛型类型传播图中增加表示类型参数的节点.除了每个引用类型的变量 a 对应节点外,为泛型类 C 的每个类型参数 E 均增加一个节点,记做 $C.E$.经扩充后,在以下分析讨论过程中将类型参数节点与变量节点统一处理.然后,增加边以表达类型参数的类型传播.在 VTA 类型传播图中,所有边均是由赋值对应产生的.注意到,在增加了类型参数节点后,声明语句是唯一将变量和类型参数联系起来的设施,且声明语句也反映了变量和类型参数之间直接的类型约束关系.因此,为这些约束关系也添加了相应边.给定泛型类 C 及其类型参数 T,我们讨论 3 种情况:一是形如 $T\ a$ 声明语句,一旦 a 的类型确定,a 所在的泛型类实例变量的类型参数 T 的实际类型也即确定.因此,引一条从 a 到 T 的边;二是形如 $C'\langle T\rangle\ a$ 的声明,$a.T$ 的类型将影响泛型类 C'的实例变量的 T 的类型,添加一条从 $a.T$ 到 $C.T$ 的边;三是 $C.T$ 也是 C 的父类 C'使用的类型参数,反映了父子类的类型参数间的约束关系,添加一条从 $C'.T$ 到 $C.T$ 的边.

为了解决第 2 个问题,我们为每个泛型变量分配一个泛型类的复制节点(包括属性节点和方法节点),而不是共享类节点,从而可以在泛型变量上作属性敏感的分析,提高精度.

由此,得到泛型类型传播图的构造方法:首先在 VTA 类型传播图的基础上添加类型参数节点及表达变量和类型参数之间的类型约束关系泛的边,为泛型类构造类型传播子图;然后为客户端代码构造类型传播子图,其中要为泛型变量增加复制节点;最后,连接所有子图构成完整的泛型类型传播图.具体过程见算法 1.

算法 1. 构造泛型类型传播图 constructGTPG().

输入:Java 程序 P;

输出:程序 P 对应的泛型类型传播图.

```
begin
  for 每个泛型类 C do
    以 VTA 方式添加节点;
    for 每个类型参数 T do
      添加 T 对应的节点 C.T;
      if 类声明形式为 C⟨T⟩ extends C′⟨T⟩ then 添加一条 C′.T 到 C.T 的边;
      if 存在声明语句 T a then 添加一条从 a 到 C.T 的边;
      if 存在声明语句 C′⟨T⟩ a then 添加一条从 a.T 到 C.T 的边;
    endfor
```

```
    endfor
    for 每个非泛型类 C do
        for C 中的每个变量 v do
            if v 是一个泛型变量 then
                为 v 添加一个复制节点;
                将标签 C.f 改为 v.f;
            endif
            以 VTA 方式添加节点;
        endfor
    endfor
    以 VTA 方式添加边;
end
```

图 3 为泛型类 *Cell* 的泛型类型传播图.与图 2 所示的 VTA 类型传播图相比,增加了 *Cell.V* 节点表示类 *Cell* 的类型参数 *V*,以及从各变量节点到该节点的边,表达该类型参数和各变量之间的类型约束关系.

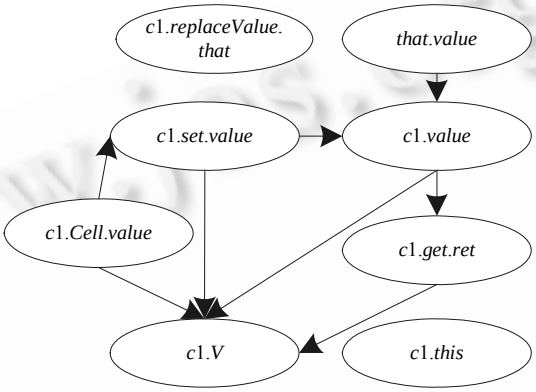

Fig.3 Generic type propagation graph for *Cell*

图 3 类 *Cell* 的泛型类型传播图

由于添加了表示类型参数的类型约束的节点和边,在泛型类型传播图中会出现一些因泛型变量属性间别名引起的隐式类型传播路径;并且,尽管使用复制节点区分了不同泛型变量的属性,仍无法确定使用类型变量进行泛型变量声明的属性域的可能类型,采用传统的类型传播分析方法会造成精度上的损失.为此,首先查找所有的对象创建语句,为变量赋予初始类型;然后,沿赋值产生的边进行类型传播,在传播分析的同时,进行变量的别名分析.具体过程见算法 2.通过别名分析,可以消除隐式类型传播路径,提高分析的精度.

算法 2. 类型传播分析算法.

输入:Java 程序;

输出:程序中各变量的可达类型集$\mathcal{R}_type(v)$,变量等价类划分$[v]_{alias}$.

```
begin
    constructGTPG(P);   //构造类型传播图
    合并强连通图;
    for 程序中每个对象实例化创建点 do
        用实例化对象的类型初始化节点;
    endfor;
    for 类型传播图中的每个节点 v do
```

```
    [v]alias={v};
    ℛ_type([v]=Type(v);   //Type(v)是 v 的静态类型
  endfor;
  for 类型传播图中的每条边 Edge(v1,v2) do
    if MayAlias(v1,v2)==true then   //合并别名变量及其对应属性的别名等价类
      [v1]alias=[v2]alias=[v1]alias∪[v2]alias;
      ℛ_type([v2]alias)=ℛ_type(v1);   //从 v1 向 v2 别名等价类中节点传播类型
      for [v2]alias 中的任意元素的属性 f do
        [v1.f]alias=[v2.f]alias=[v1.f]alias∪[v2.f]alias;
        ℛ_type([v1.f]alias)=ℛ_type([v2.f]alias)=ℛ_type([v1.f]alias)∪ℛ_type([v2.f]alias);
      endfor
    endif
  endfor
end
```

3 别名问题

别名关系是指程序中两个表达式访问相同的内存空间.在 Java 程序中,任意一个引用变量上的非空赋值都会造成别名.给定引用变量 x 和 y,令 f_1 是它们的合法属性域,f_2 是 f_1 类型的一个属性域,则 $x=y$ 会造成 $x.f_1$ 和 $y.f_1$ 的别名,而 $x.f_1=y$ 的赋值会造成 $x.f_1.f_2$ 和 $y.f_2$ 的别名,$x=y.f_1$ 的赋值会造成 $x.f_2$ 和 $y.f_1.f_2$ 的别名.VTA 方法用单个节点来表示一个类所有实例的属性域 f,隐式地假设了任一对属性域 f 的访问都可能存在别名关系.本文方法由于用不同类型节点表示不同泛型变量的属性,上述假设不再成立,必须引入新的方式来解决别名问题,否则会影响重构的精确性.

首先,别名会造成一些隐式传播路径,在类型传播图中不能显式表示.考虑第 1 种别名方式:$x=y$.如果 x 和 y 有属性域 f,则 $x.f$ 和 $y.f$ 也是别名.从 $x.f$ 到 $y.f$ 有一条隐式的类型传播路径,反之亦然.如果再加上其他方式的别名,则会形成更为复杂的情况.考虑如下语句序列:$x1=x2;x2.f=x;x3=x1.f$,$x1.f$ 和 $x2.f$ 是别名,从而,x 的类型通过赋值路径 $x2.f=x$ 能够到达 $x1.f$ 和 $x3$.在泛型类型传播图中,这种隐式的传播路径不能直接表示出来.

通过别名分析能够识别这些隐式类型传播路径,并尽可能地消除其影响.上例中,如果能够确定 $x1$ 和 $x2$,$x1.f$,$x2.f$,x 和 $x3$ 是别名,就能识别出 $x2.f$ 到 $x1.f$ 的类型传播路径,进一步识别出 x 到 $x1.f$ 和 $x3$ 的类型传播路径.

其次,对于泛型类中以类型参数声明的变量,由于无法判断其是否为泛型变量,在构造类型传播图时不会为其分配复制节点,因此无法从类型传播图中确定它们的属性域的可能类型.例如,对于图 1 中的语句序列 29,30,31,可知 $c3$ 为泛型变量,而 $c2.get()$的声明为 V,尽管由图 4 的类型传播图可以确定 $c2.V$ 的可达类型为{*Cell*},然而,由于没有为其分配复制节点,不清楚其属性域的可达类型,当类型传播至 $c3$ 时,也只能确定 $c3$ 的可达类型是{*Cell*},而不能确定 $c3.V$ 的可达类型.

这一问题可以根据别名分析的结果得以解决.由别名分析结果可知,$c3$ 和 $c2.value$,$c1$ 存在等值关系,如果知道 $c2.value$ 或者 $c1$ 的类型,$c3$ 及 $c3.V$ 的类型也就可知.从类型传播图中可以确定 $c1$ 的可达类型为{*Cell*⟨*Float*⟩},从而能够确定 $c3$ 的可达类型也为{*Cell*⟨*Float*⟩},也就能确定 $c3.V$ 的可达类型为{*Float*}.

可见,进行别名的计算可以尽可能地消除隐式类型传播路径,提高泛型实例重构的精度.本文在进行类型传播的同时计算别名,方法是进行别名等价类的划分,这样的计算方法简单且效率较高.

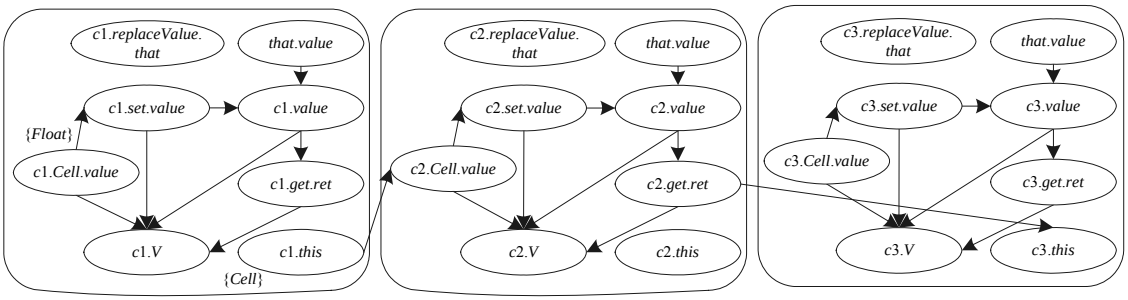

Fig.4 Generic type propagation graph for statement 29, 30 and 31 in Fig.1

图 4 与图 1 中实例程序语句序列 29,30,31 对应的泛型类型传播图

我们假设:如果两个变量之间存在直接或者间接的赋值关系,则认为这两个变量为别名.这是计算别名的重要依据,根据这个假设,所计算的别名是可能别名,即在部分程序执行中成立的别名关系,所得别名是一个保守但安全的结果.根据上述假设,可以给出如下判别函数:

$$MayAlias(v1,v2)\equiv Subtypes(Type(v1))\cap Subtypes(Type(v2))\neq\varnothing\wedge Edge(v1,v2)\in GTPG,$$

其中:$Type(v)$是变量 v 的静态类型;$Subtypes(T)$代表类型 T 的子类型集,包含 T 自身.当两个变量之间存在赋值关系,即在 GTPG 中是否有一条边连接两个变量 $v1$ 和 $v2$,且它们的静态类型的子类型集存在交集时,该函数为真.

如算法 2 所示,初始化时,每个变量自身作为一个别名等价类.令$[v]_{alias}$表示变量 v 所在的别名等价类,初始化为$[v]_{alias}=\{v\}$.在类型传播过程中,如果两个变量 $v1$ 和 $v2$ 满足上述判别函数,则合并它们的别名等价类:$[v2]_{alias}=[v1]_{alias}=[v1]_{alias}\cup[v2]_{alias}$.此外,我们认为,如果两个变量 $v1$ 和 $v2$ 在一个别名等价类中,则 $v1$ 及其任意属性和 $v2$ 及其任意属性对应地存在别名关系,也应合并相应的别名等价类.这样做可以消除泛型类型传播图中更多的隐式传播路径所带来的影响.

考虑本节给出的语句序列,$c1$,$c2.value$ 和 $c3$ 存在别名关系.由别名关系带来的隐式类型传播路径:

$$\{Float,c1.Cell.vallue,c1.set.value,c1.value,\ldots,c3.value\}$$

却并不能直接从 GTPG 中得到.通过别名分析,进行别名等价类划分后,可以得到等价类集合:

$$\{c1,c2.value,c3\},\{c1.value,c3.value\},$$

可知 $c1.value$ 的可达类型同样会传播给 $c3.value$,从而确定 $c3.V$ 的可达类型.

4 类型实参的选择

由于在泛型类型传播图中模拟了类型参数的传播,从类型传播图上即可得到泛型变量类型参数的可达类型,无须复杂的类型约束分析.经过类型传播分析后,泛型变量及其类型参数的可达类型存在如下几种情况:

(1) 泛型变量的可达类型不为单一类型.在这种情况下,该变量可能用于多个泛型类的实例化,不进行重构.这种情况在实际程序中很少发生;

(2) 泛型变量的可达类型只包含一种类型,但其类型参数的可达类型包含多种类型.在这种情况下,又可细分为如下两种情形:一是类型参数的可达类型没有最小公共父类;二是类型参数的可达类型有一个最小公共父类.前者我们不实施重构,对于后者,选择其最小公共父类作为重构所用的类型实参.例如,在如图 1 所示的实例中,$\mathcal{R}\_type(c4)=\{Cell\}$,$c4$ 的可达类型为单一类型,进一步判断$\mathcal{R}_type(c4.V)=\{Float,Integer\}$,$c4.V$ 的可达类型不唯一,但 *Float* 和 *Integer* 有一个最小公共父类 *Number*,因此可以选择 *Number* 作为类型实参;

(3) 泛型变量及其类型参数的可达类型均为单一类型.这种情况最为简单,我们只需选择类型参数的可达类型作为该泛型变量的类型实参即可.

所选择的类型实参的类型也可能是泛型类.例如在图 1 所示的实例中,$\mathcal{R}\_type(c2)=\{Cell\}$,若直接选择 *Cell* 作为 $c2$ 的类型实参不够精确,更精确的结果应该是 $Cell\langle Float\rangle$.我们可以根据别名分析的结果来确定嵌套的实参类型.

给定某个泛型变量 v 的类型变量 T,若所选 T 的类型实参为泛型类.根据前面别名分析的结果,如果两个引用变量为别名,则其对应属性也为别名.假定 u 是 $[T]_{alias}$ 中的某个元素 u,且 u 有属性 f,则

$$\mathcal{R}\_type(T.f)=\mathcal{R}\_type(u.f).$$

如果所选 $u.f$ 的类型实参仍是泛型类,重复上述分析过程,直到某个 $u.f$ 的类型实参不是泛型类,或者不可选定合适的类型实参.例如,$c2.V$ 的可达类型为{*Cell*},查找 $[c2.V]_{alias}$={$c1$,$c3$,…},其中有 $c1$ 的类型实参为 *Float*,因此,最终可以确定 $c2.V$ 为 *Cell*⟨*Float*⟩.

表 1 列出了图 1 实例中的泛型变量{$c1$,$c2$,$c3$,$c4$}的可达类型和最终选择的类型实参.

Table 1 Reachable types of the generic variables and their type variables

表 1 实例程序中泛型变量及其类型变量的可达类型

No.	Variable	$\mathcal{R}\_type()$	Type argument	No.	Variable	$\mathcal{R}\_type(v)$	Type argument
1	$c1$	{*Cell*}	*Float*	5	$c3$	{*Cell*}	*Float*
2	$c1.V$	{*Float*}		6	$c3.V$	{*Float*}	
3	$c2$	{*Cell*}	*Cell*⟨*Float*⟩	7	$c4$	{*Cell*}	*Number*
4	$c2.V$	{*Cell*}		8	$c4.V$	{*Integer*,*Float*}	

此外我们注意到,在实际程序中,泛型变量在声明点及其实例化点极少使用不同的类型实参.因此,本文不对声明点及其实例化点类型实参分别作推导,而是采用简单的策略,认为两处使用相同的类型实参.

5 实验分析

为了考察方法的有效性,我们在 Soot 分析框架[4]的基础上实现了一个工具原型用于实验分析.Soot 是一种开源的程序分析框架,它可以将 Java 源代码转换成 3 种中间表示,并提供了在这些中间表示之上实现控制流、数据流分析操作的接口.在考察的基准程序中,JUnit 是一个测试框架,JLex 是一个 Java 的词法解析生成器,Htmlparser 是用于对 html 文件进行解析的开源软件,vpoker 是一个视频扑克游戏引擎.Java 的容器位于 java.util 包中,其中,ArrayList,Vector(两者的不同仅在于 Vector 是可同步的)是最常用的两种容器类,占了所有程序中容器类使用的 90%以上.Junit,JLex 和 vpoker 基本上仅使用了这两种容器.在 Htmlparser 中,则也使用了 Hashtable, HashSet 和 HashMap 等.我们对这些容器都进行了考察.

泛型实例重构能够消除原程序中由于对 raw-type(指原有的非泛型类型)引用导致的类型转换操作,因此,类型转换的消除情况是对泛型实例重构效果的一个很好的评价标准[3,5−8].我们在实验中,主要考察了所重构的实例化点数量和消除的因 raw-type 引用导致的类型转换操作数.表 2 列出了实验结果,程序规模列给出了各个程序的代码行数,第 3 列是被分析程序中使用的容器类的实例化点数量,第 4 列是采用我们的方法所识别出的可被重构为泛型容器实例化点数量,第 5 列给出了原程序中由于 raw-type 引用导致的类型转换操作,最后一列给出了重构后可以消除的类型转换数目.

Table 2 Experiment result of analyzing four open source software

表 2 对几个开源软件分析的实验结果

Benchmark	Size (Code lines)	Instantiation	Refactored instantiation	Type conversion due to raw-type reference	Type conversion elimination
JUnit	3 081	27	21	68	37
JLex	8 605	35	32	57	53
vpoker	4 401	12	10	31	23
Htmlparser	26 060	62	58	33	24

表 3 给出了相关工作的一个对比,Donovan 等人的方法[3]和 Tip 等人的方法[5−7]是比较典型的两个相关工作.表中列出了对几个基准程序实施重构后,被重构的实例化点的数量和可消除的类型转换操作数量. Donovan 等人没有给出可被重构的实例化点数量.Tip 等人给出了可消除的类型转换操作数,然而,这些类型转换操作是指原程序中所有的类型转换操作,而不仅是由于 raw-type 引用导致的类型转换操作.表 3 中以“–”表示未给出的项.

我们将实验结果与基准程序进行了手工检查,以分析基准程序中不能被重构的代码.首先,容器数组未能被

重构.Java1.5 不支持容器数组的泛型化,因此,所有的容器数组都只能保留 raw-type 的引用方式;其次,异质容器是不能被重构的.异质容器可以容纳多种类型的元素,显然是不能、也不应该被泛型化的.最后,一些程序会对容器对象调用 *clone*()方法.*clone*()方法的返回值声明类型为 *Object*,在调用 *clone*()方法时会丢失类型信息,也导致有些情况不能识别.

Table 3 Experiment result comparison for the refactored instantiation and the type conversion elimination

表 3 被重构的实例化点及消除的类型转换操作比例实验结果对比

Benchmark	Refactored instantiation			Type conversion elimination		
	Donovan	Tip (%)	Chen (%)	Donovan (%)	Tip	Chen (%)
JUnit	–	100	78	62	–	54
JLex	–	94	91	98	–	93
vpoker	–	100	83	77	–	74
Htmlparser	–	97	94	78	–	73

我们假设泛型变量声明点和实例化点具有相同的类型实参,有两个缺点:一是不能识别声明类型的实参使用通配符的情况;二是无法确定使用接口进行声明的变量的类型实参.原因都在于它们没有实例化点:Java 不允许使用通配符作为类型实参进行实例化,接口也不能实例化.然而,通过手工检查所考察的程序,使用接口声明的变量经由具体类实例化创建对象,其声明所用实参和具体类实例化点所用类型实参是一致的.

需要指出的是,如果仅使用实例对象的类型初始化类型传播的节点,那么对于如下形式的代码:

```
void foo(Type1 f){
  Vector v=new Vector();
  v.addElement(f);
}
```

如果在所有对方法 *foo*()的调用中均没有实例对象作为参数传递,则不能确定变量 *v* 的类型实参.通过对不同的基准程序考察,这种形式的代码在框架程序中大量存在,且框架程序通常是作为服务端代码,缺乏创建对象的语句.不考虑这个因素,将会大大降低重构的效果.我们在分析时,让变量的声明类型也参与类型传播分析.这样,上面的实例就可以重构为 *Vector*⟨*Type*1⟩,只有 *Type*1 或其子类型的对象才能放入容器中.

6 相关工作

本文使用类型分析技术实现了泛型实例重构,因此,我们主要关注两方面的相关工作:类型分析和重构.

类型分析是有助于程序分析、理解和维护的重要技术.其中,0-CFA 是一种上下文不敏感的算法,其时间复杂度为 $O(n^3)$,在 0-CFA 基础上提出的改进的上下文敏感算法 *k*-CFA 虽然精度提高,但复杂度也随 *k* 呈指数级增长[9].CPA 是目前使用较多的一种上下文敏感分析算法,但其时间复杂度为指数级[1].VTA 是一种时间复杂度较低的上下文不敏感分析算法[2],在 Java 调用图构造等领域中得到了很好的应用.然而,由于其不区分不同对象的属性,不能区分泛型类的实例化点,因此不能直接用于泛型程序的类型分析和重构.

Donovan 等人提出了一种基于指向分析的泛型实例重构方法[3].他们先用上下文敏感的分析算法 CPA 确定对象分配点的可能类型,然后使用基于集合约束求解的上下文无关分析选择对分配点和声明点一致的类型.von Dincklage 等人在 IlWith 重构系统中同时考虑了泛型程序参数化重构与实例重构[8].他们的方法也是基于 CPA 算法先进行分析,再进一步确定待重构代码.其方法可以为程序员提供有用的参考.Tip 等人提出的泛型实例重构方法先用 CPA 计算方法的上下文,再将其用于约束的生成与求解[5−7],从而得到重构结果.他们的方法能够推导出继承自类库类的用户自定义类型的使用.他们也考虑了参数化和实例重构问题.其方法通过推导程序中各元素的类型约束确定重构代码.

现有的重构方法基本上都首先使用上下文敏感的 CPA 方法,其时间复杂度是指数级的.在上下文敏感分析的基础上,再使用迭代的类型约束求解来进一步确定类型实参,通常,该步骤的时间复杂度是 $O(n^3)$[10].如果不考虑上下文,则精确度将大为降低.

与现有工作相比,本文方法的主要优点是利用类型分析技术在较低的时间复杂度内实现泛型实例重构,且能取得较好的效果.在算法效率方面,本文方法与VTA方法的时间复杂度处于一个量级,与程序代码的规模近似呈线性增长.根据VTA算法构造的类型传播图所包含的节点数为$2M+P+L+F$.其中,M是方法数,P是参数的总个数,L是局部变量的总个数,F是属性的总个数.这样,节点个数的增长是与程序代码规模增长呈线性关系的.由于每个节点可能有多条边与其相连,因此,边的计算要复杂一点,最多可能有 $O(C\times M_C)$条边.其中,C是类的个数,而M_C是方法调用的个数[2].与 VTA 算法相比,我们方法的节点数较多,因为要给每个泛型变量分配复制节点.最坏情况下,假设程序中全是泛型变量,则节点数为 $2M+G\times(P+L+F)$.其中,G 是泛型类节点的平均个数.在实际程序中,G是一个常数,并不随所分析程序规模的增长而增长.

为了提高分析的精度,我们通过别名分析消除了部分隐式类型传播路径,同时解决了因使用类型变量进行泛型变量声明而无法确定其属性域可能类型的问题.别名分析采用在类型传播的同时基于赋值合并别名等价类的方法,这实际上也是一种基于合并约束的方法[11],可以得到安全保守的结果,并且是一种复杂度近似线性的分析方法.

7 结 论

重构是改善代码结构、提高软件质量的一种重要手段.由于重构往往是连续而实时的行为,且需要给予程序员即时的反馈,因此,时间复杂度较低的重构方法更易于被程序员所接受[12].从20世纪90年代起,我们就在软件重构方面展开研究,针对Ada83,C++,Java等语言特点,提出了一系列重构技术改善软件质量[13−16].在已有研究的基础上,针对现有泛型实例重构方法时间复杂度较高、不利于即时持续重构的问题,我们提出了一种基于改进的VTA类型传播分析的泛型实例重构方法.提出了一种泛型程序类型传播图.基于泛型类型传播图,提出了一种泛型变量属性敏感的类型分析,并对泛型实例重构中的别名问题作了进一步探讨;提供了一种比较简单、易行、效率较高的泛型变量实参选择方式.

将重构工具集成到常见的IDE中,提供半自动化或自动化的重构支持,能够极大地提高软件开发的生产力.目前,我们的工作仍处于原型开发阶段,在未来的工作中,我们将对更多的实验对象展开分析,以验证本文方法的效果.同时开发实用重构工具,并将其集成到常用的开发环境中以利于程序员使用.

致谢 在此,我们向对本文工作给予支持和建议的同行表示感谢.同时,对审稿人提出的有益建议表示感谢.

References:

[1] Agesen O. Concrete type inference: Delivering object-oriented applications [Ph.D. Thesis]. Stanford University, 1996.

[2] Sundaresan V, Hendren L, Razafimahefa C, Vallee-Rai R, Lam P, Gagnon E, Godin C. Practical virtual method call resolution for Java. In: Proc. of the Conf. on Object-Oriented Programming: Systems, Languages, and Applications. New York: ACM, 2000. 264−280. http://portal.acm.org/citation.cfm?id=354222.353189

[3] Donovan A, Kieżun A, Ernst MD. Converting Java programs to use generic libraries. In: Proc. of the ACM Symp. on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA). New York: ACM, 2004. 15−34. http://portal.acm.org/citation.cfm?id=1028979

[4] Vallée-Rai R, Hendren L, Sundaresan V, Lam P, Gagnon E, Co P. Soot: A Java optimization framework. In: Proc. of the IBM Centre for Advanced Studies Conf. Mississauga: IBM Press, 1999. 125−135. http://portal.acm.org/citation.cfm?id=782008&dl=

[5] Tip F, Fuhrer R, Dolby J, Kieżun A. Refactoring techniques for migrating applications to generic java container classes. Technical Report, RC23238 (W0406-045), IBM Research Division, 2004.

[6] Kieżun A, Ernst MD, Tip F, Fuhrer R. Refactoring for parameterizing Java classes. In: Proc. of the 29th Int'l Conf. on Software Engineering. Washington: IEEE Computer Society, 2007. 437−446. http://portal.acm.org/citation.cfm?id=1248820.1248876

[7] Fuhrer R, Tip F, Kiezun A, Dolby J, Keller M. Efficiently refactoring Java applications to use generic libraries. In: Proc. of the 19th European Conf. on Object-Oriented Programming. Glasgow, 2005. 71−96. http://www.springerlink.com/content/7fkjxm1lcvec9qr1/

[8] von Dincklage D, Diwan A. Converting Java classes to use generics. In: Proc. of the ACM Symp. on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA). New York: ACM, 2004. 1−14. http://portal.acm.org/citation.cfm?id=1028978

[9] Shivers O. Control-Flow analysis of higher-order languages [Ph.D. Thesis]. School of Computer Science, Carnegie Mellon University, 1991.

[10] Nielson F, Nielson HR, Hankin C. Principles of Program Analysis. New York: Springer-Verlag, 1999.

[11] Steensgaard B. Points-To analysis in almost linear time. In: Proc. of the Symp. on Principles of Programming Languages. New York: ACM, 1996. 32−41. http://portal.acm.org/citation.cfm?id=237721.237727

[12] Fowler M. Refactoring: Improving the Design of Existing Code. Boston: Addison-Wesley, 1999.

[13] Zhou TL, Shi L, Xu BW. Refactoring C++ programs physically. Journal of Software, 2009,20(2):256−270 (in English with Chinese abstract). http://www.jos.org.cn/1000-9825/561.htm

[14] Chen L, Xu BW, Zhou XY, Cao J. Refactoring generic Java programs based om type inference. ACTA ELECTRONICA SINICA, 2007,35(12A):185−191 (in Chinese with English abstract).

[15] Zhou YM, Xu BW. An object-extracting approach using module cohesion. Journal of Software, 2000,11(4):557−562 (in Chinese with English abstract). http://www.jos.org.cn/ch/reader/view_abstract.aspx?flag=1&file_no=20000421&journal_id=jos

[16] Li BQ, Xu BW. An approach for transforming Ada83 serving tasks to Ada95 protected objects. Journal of Software, 2000,11(6):836−840 (in Chinese with English abstract). http://www.jos.org.cn/ch/reader/view_abstract.aspx?flag=1&file_no=20000620&journal_id=jos

附中文参考文献:

[14] 陈林,徐宝文,周晓宇,曹璟.一种基于类型约束的泛型 Java 程序重构方法.电子学报,2007,35(12A):185−191.

[15] 周毓明,徐宝文.一种利用模块内聚性的对象抽取方法.软件学报,2000,11(4):557−562. http://www.jos.org.cn/ch/reader/view_abstract.aspx?flag=1&file_no=20000421&journal_id=jos

[16] 李帮清,徐宝文.一种 Ada83 服务性任务向 Ada95 保护对象变换的方法.软件学报,2000,11(6):836−840. http://www.jos.org.cn/ch/reader/view_abstract.aspx?flag=1&file_no= 20000620&journal_id=jos

陈林(1979－),男,广东兴宁人,博士生,主要研究领域为软件分析,软件重构.

周天琳(1982－),女,博士生,主要研究领域为程序分析与重构.

徐宝文(1961－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为软件工程,并行与网络软件.

周毓明(1974－),男,博士,教授,主要研究领域为软件度量.

钱巨(1981－),男,博士,讲师,主要研究领域为程序分析.