基于源代码静态分析的 C++0x 泛型概念抽取

陈 林[1),2),3)] 徐宝文[1),2),3)]

[1)](东南大学计算机科学与工程学院 南京 210096)

[2)](南京大学计算机软件新技术国家重点实验室 南京 210093)

[3)](南京大学计算机科学与技术系 南京 210093)

摘 要 使用泛型概念对领域知识进行抽象是泛型程序设计方法的基础.在新的 C++0x 标准中泛型概念将成为一个新的语言设施,这将为设计可复用、可扩展的泛型软件提供坚实的基础.为了更好地利用 C++0x 的新特性,有必要识别 C++遗产代码中的泛型概念,并通过重构得到符合 C++0x 标准的代码.文中提出了一种基于代码静态分析的泛型概念自动识别方法,通过对泛型程序中类型参数的使用分析,从遗产代码中提取有效表达式约束和关联类型约束,进而推导出泛型概念.将该方法应用于 C++标准模板库 STL,可以识别出 STL 算法中绝大部分潜在的泛型概念,表明该方法有助于识别遗产代码中的泛型概念.

关键词 泛型程序设计;软件重构;泛型概念;C++0x;约束分析

中图法分类号 TP311 **DOI 号**: 10.3724/SP.J.1016.2009.01792

Using Static Analysis to Extract C++0x Concepts

CHEN Lin[1),2),3)] XU Bao-Wen[1),2),3)]

[1)](*School of Computer Science and Engineering, Southeast University, Nanjing* 210096)

[2)](*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing* 210093)

[3)](*Department of Computer Science and Technology, Nanjing University, Nanjing* 210093)

Abstract Using Concepts to abstract domain knowledge is the basis for generic programming. Concept will be a new language feature in C++0x, which provides solid base for developing reusable and extendable generic software. To make better use of new generic programming features of C++0x, it is necessary to identify Concepts in legacy C++ programs, and refactor them to C++0x programs. This paper presents an automatic method to identify Concepts with static analysis, analyzes the usages of type parameters in legacy programs to extract valid expression constraints and associate type constraints. Concepts can be inferred by these constraints. The approach is applied to the C++ Standard Template Library to identify most of the potential Concepts in STL. The study convinces that this method is helpful to identify Concepts in legacy C++ programs.

Keywords generic programming; software refactoring; Concept; C++0x; constraint analysis

1 引 言

泛型程序设计方法以抽象和效率为目标,已经在业界得到越来越广泛的使用.它强调根据类型需要满足的约束进行分类和抽象,其基本抽象单位称为"泛型概念"(Concept).泛型程序设计就是泛型概念的设计[1-6],通过认知抽象约束,对抽象约束进行

收稿日期:2009-04-14;最终修改稿收到日期:2009-07-24.本课题得到国家自然科学基金重大研究计划(90818027)与重点项目(60633010)、国家"八六三"高技术研究发展计划目标导向类项目(2009AA01Z147)资助.**陈 林**,男,1979 年生,博士研究生,主要研究方向为软件分析、软件重构.E-mail: lchen@nju.edu.cn.**徐宝文**,男,1961 年生,博士,教授,博士生导师,主要研究领域为程序设计语言、软件工程、并行与网络软件.

合理的分类,从而设计和定义泛型概念,进而根据泛型概念撰写程序.目前,支持泛型程序设计方法的主要语言设施是参数化类型,如 Java 泛型、C++模板和 Ada 类属等.参数化类型设施为复用提供了良好基础,然而,它无法为泛型程序设计提供进一步的支持.首先,泛型概念表达的是作用于类型上的抽象约束,参数化类型不能很好地支持这一抽象的表达.其次,现有参数化类型实现无法对泛型概念所表达的约束进行检查,增加了程序设计的困难.为了提供对泛型概念更好的表达和正确性检查,新的 C++标准 C++0x 将把泛型概念作为一个语言设施[5].

识别遗产软件中潜在的泛型概念,将代码重构为符合 C++0x 标准的软件,能提高遗产软件的可扩展性和可维护性,具有重要的现实意义.首先,遗产软件将能利用 C++0x 的新特性,可以对原有软件进行分析验证,确保泛型概念在其中的使用是否正确.其次,加强软件理解.遗产软件中类型参数仅作为占位符存在,以 T、E 等简单字符为类型参数命名,抽取出泛型概念后,有助于理解.再次,抽取出泛型概念,有利于认知遗产软件中的抽象组件,梳理遗产软件的体系结构,可以基于泛型概念实现新的算法、新的用户定义类型等,有利于不同软件组件(包括遗产软件和新开发软件组件)之间的交互,提高软件质量.

目前研究如何将遗产代码重构为泛型代码的工作中,大多从参数化类型的角度考虑,基于类型分析研究如何为类或方法引入参数化类型,将其泛型化.由于类型和泛型概念所表达的抽象语义不同,基于类型分析的重构方法难以从代码中识别泛型概念,不适用于将原有 C++程序重构为符合 C++0x 标准的代码.

我们曾对泛型程序源代码静态分析做过比较深入的研究[7-9],本文是在以前工作的基础上,通过对类型参数的使用分析,抽取泛型概念.首先,介绍了泛型程序设计的基本概念.然后,提出了基于静态分析的泛型概念抽取方法,通过从源代码中抽取有效表达式约束和关联类型约束,构造约束集,并由约束集推导出泛型概念.最后,对 C++标准模板库 STL 进行了分析以验证本文方法的有效性.

2　C++0x 和泛型概念

C++使用基于结构化约束签名匹配的方式表达模板参数的约束,并以此表达泛型概念,虽然具有较大的灵活性,然而,这种方式对泛型概念的表达是隐晦的,不容易理解,并且不支持即时的类型检查,由此引发编译信息晦涩,难以对错误进行定位等问题.

例如,图 1(a)是抽取自 STL 中的简化后的 fill 算法,符合 C++98 标准.算法要求类型参数 *Iter* 必须是前向迭代子(Forward Iterator).前向迭代子就是一个泛型概念,它表达的部分抽象约束如表 1 所示.

```
1. template〈typename Iter, typename T〉
2.
3. void fill (Iter first, Iter last, const T &value) {
4.     for (; first !=last; ++first) {
5.         *first=value;
6.     }
7.     //first--;
8. }
```

(a) 符合 C++98 标准的模板函数 fill

```
1. template〈MutableForwardIterator Iter, typename T〉
2.     where Assignable〈Iter::reference, T〉
3. void fill (Iter first, Iter last, const T &value) {
4.     for(; first !=last; ++first) {
5.         *first=value;
6.     }
7.     //first--;
8. }
```

(b) 符合 C++0x 标准的模板函数 fill

图 1　两个不同版本的模板函数 fill

表 1　前向迭代子的部分需求

分类	迭代子			
精化自	Input Iterator, Output Iterator			
关联类型	与 Input Iterator 相同			
符号	X	前向迭代子模型的一类		
	T	*X* 值的模型		
	i, *j*	类型 *X* 的对象		
	t	类型 *T* 的对象		
有效表达式	名称	表达式	类型需求	返回类型
	Preincrement	++*i*	…	*X*&
	Postincrement	*i*++	…	*X*
复杂度保证	前向迭代子的操作复杂度是平摊的常数时间			

在 C++98 标准下,表 1 所示约束只能通过类型参数所声明的变量的使用来保证.图 1 程序中类型参数 *Iter* 必须满足具有解引用、等价比较、前自增等操作,这些约束通过变量 *first* 和 *last* 的使用来表达.这种通过表达式的使用隐含在语句中的约束,称为有效表达式约束.编译器不检查类型参数 *Iter* 是否为一个前向迭代子,只有在函数被实例化时,才通过检查表达式使用来验证这一约束.例如,如果在图 1(a)第 7 行添加语句"*first*——",该操作并不满足前向迭代子的约束(前向迭代子不具有"自减"操

作),然而,编译器不会在 fill 函数的定义处报错.

下面先给出泛型概念的一个普遍定义.

定义 1(泛型概念). 泛型概念是一个二元组 $\langle R, A\rangle$. R 是一组约束,A 是一组抽象,并且满足一个抽象包含在 A 中当且仅当它满足 R 中的所有约束.

在 STL 中,抽象约束被具体化为 4 种约束:有效表达式约束、关联类型约束、不变式和复杂度约束.本文将采用 STL 中给出的约束对源代码进行分析.不变式和复杂度约束涉及到复杂语义,通常很难从源代码中识别,我们不予考虑,仅从源代码中提取类型参数的有效表达式和关联类型约束构造约束集.

使用模板的 C++遗产软件,尤其是以 STL 为代表的一些泛型软件库,如果能识别出类型参数所表达的泛型概念,将它们重构为符合 C++0x 标准的代码,一方面,可以提高编译器类型检查的能力,便于错误定位;另一方面,有利于程序理解,有助于软件的复用和维护.如图 1(b)的 fill 函数,明确规定了类型参数 T 必须满足可变前向迭代子(MutableForwardIterator)这一泛型概念,通过可变前向迭代子,给出了类型参数 *Iter* 必须满足的约束条件,编译器就能对"*first*——"语句进行检查并报错.同时,抽取出了可变前向迭代子和可赋值两个泛型概念,有助于理解 fill 函数(不能给 fill 传递一个时间复杂度低于常数时间的迭代子参数),迭代子等泛型概念的使用也有助于加强软件组件间的交互,使程序更易于复用和维护.

考察图 1 两个不同版本的 fill 函数,可以发现 C++0x 标准和 C++98 标准程序的主要区别:必须给出泛型概念的定义,供编译器检查;类型参数不再是占位符,必须显式指定它们必须满足的泛型概念.据此可给出将 C++98 标准的程序重构为 C++0x 标准程序的主要步骤:①设计和定义泛型概念,构建泛型概念库;②分析模板参数的使用,包括操作、类型、函数调用等,提取类型参数的约束,构造约束集;③进行泛型概念的推导,指定类型参数必须满足的泛型概念;④实施重构.

泛型概念的设计定义通常由有经验的程序员和专家完成.STL 和 C++0x 文档均预定义了一批设计良好的泛型概念,可以借助它们构建泛型概念库.约束集通过分析类型参数所声明的变量使用情况进行构造.例如对图 1(a)的 fill 函数分析,由 *Iter* 所声明的两个变量 *first* 和 *last*,可知 *Iter* 应该具有解引用、等价比较、前自增等约束.基于约束集的泛型概念推导,为指定类型参数必须满足的泛型概念提供了候选集.对于图 1 的 fill 函数,可由约束集推导出 *Iter* 必须满足前向迭代子泛型概念.

3 泛型概念抽取

抽取泛型概念所要解决的主要问题是:如何有效地分析模板参数的使用、提取约束集并进行泛型概念的推导.模板参数的使用情况可以通过分析模板参数所声明的变量使用情况来获得.本节将讨论如何针对这些变量从代码中抽取有效表达式约束和关联类型约束.

3.1 有效表达式约束

有效表达式约束指定了类型参数所声明的变量应该具有的操作.我们定义了一批有效表达式约束计算规则,如表 2 所示.对于给定的变量 a 及其使用,通过分析抽象语法树,根据规则从中提取有效表达式约束.例如,从抽象语法树中可以容易地获取表达式"$*first$",再根据指针表达式的约束计算规则,即可得到约束.本文计算的有效表达式约束包括赋值表达式、数值表达式、比较表达式、条件表达式、构造函数、函数调用、指针操作(解引用、取成员操作)、取属性域和成员函数调用等几种类型.

表 2 部分有效表达式及其满足的约束

分类	有效表达式形式	所满足的约束
赋值表达式	$a=expr$;	assignable$\langle a,expr\rangle$
数值表达式	$a+b$, 或 $b+a$	addable$\langle a,b\rangle$
	$a-b$, 或 $b-a$	subtractable$\langle a,b\rangle$
	$a*b$, 或 $b*a$	multiplicable$\langle a,b\rangle$
比较表达式	$a<b$, 或 $b<a$, 或 $a\leq b$, 或 $a\geq b$	lessthanComparable$\langle a,b\rangle$
	$a==b$, 或 $a!=b$	equalityComparable$\langle a,b\rangle$
逻辑表达式	$!a$, $a\&\&b$, $a\|\|b$	convertible$\langle a,$bool$\rangle$
构造函数	Type a;	defaultConstructible$\langle a\rangle$
	Type $a(b)$;	copyConstructible$\langle a,b\rangle$
指针表达式	$*a$	dereferenceable$\langle a\rangle$
调用表达式	$a()$	callable$\langle a\rangle$
前置自增表达式	$++a$	preIncrement$\langle a\rangle$
后置自增表达式	$a++$	postIncrement$\langle a\rangle$
前置自减表达式	$--a$	preDecrement$\langle a\rangle$
后置自减表达式	$a--$	postDecrement$\langle a\rangle$

在有效表达式约束的提取中,间接表达式也需要进行分析.因为形如 $*first=value$, !pred()等的表达式,若不分析间接表达式,则有些约束会被忽略,使结果不准确.间接表达式的分析我们将在关联类型约束的提取步骤中作进一步讨论.

根据表 2 中的规则,对图 1(a)的实例进行分析,可得到表 3 所示的有效表达式约束集.

表 3　从实例程序 fill 抽取的有效表达式约束集

表达式	有效表达式约束
Iter first, *last*	copyConstructible⟨*first*⟩, copyConstructible⟨*last*⟩
first!=*last*	equalityComparable⟨*first*, *last*⟩, equalityComparable⟨*last*, *first*⟩
++*first*	preIncrement⟨*first*⟩
**first*	deferenceable⟨*first*⟩
**first*=*value*	assignable⟨**first*, *value*⟩

3.2　关联类型约束

在上一节所分析的表达式约束中,有一个特殊约束:assignable⟨**first*,*value*⟩.该约束反映了一个信息:*first* 和 *value* 两个变量之间存在一定联系.对应的,类型参数 *Iter* 和 *T* 之间也存在一定的联系.这种联系,我们称之为关联类型约束.

关联类型是 STL 中非常重要的一种约束,反映了泛型概念间的约束关系,对提高组件间交互能力和提高组件的可复用性具有重要作用.关联类型约束可以通过有效表达式约束进行计算.为便于讨论,我们先给出如下定义.

定义 2(关联对象).　给定变量 v,称通过直接作用在 v 上的操作所得结果,为 v 的关联对象.

定义 3(变量的关联类型).　给定变量 v,称 v 的关联对象的类型为 v 的关联类型.

关联对象的具体类型,是泛型概念的关联类型实例化后的具体类型.关联类型约束并不是指关联对象的具体类型,然而,可以通过关联对象的类型可以推导出关联类型约束.实际上并不需要确定关联对象的具体类型,关联对象的具体类型只能在使用实际类型对类型参数进行替换,即实例化后,才可能推导出来.

下面先给出计算关联类型约束的基本思路:给定变量 v 及其表达式约束,先计算 v 的所有关联对象,记作 $ao_1,\cdots,ao_n$,若关联对象出现在有效表达式约束中,则为该关联对象分配类型占位符,记作 $at_1,\cdots,at_n$,并将其代入有效表达式约束中;然后,根据关联类型约束匹配规则,确定占位符所代表的关联类型;最后,确定关联类型约束.

例如,对于图 1 的示例程序,变量 *first* 满足有效表达式约束 assignable⟨**first*,*value*⟩,其中 **first* 是 *first* 的关联对象.我们给关联对象 **first* 分配类型类型占位符 *first*::*associated_type*,代入有效表达式约束可得 assignable⟨*first*::*associated_type*, *value*⟩.

关联类型约束是和特定的泛型概念相关联的,因此,在未确定 *first* 对应的类型参数 *Iter* 所满足的泛型概念之前,不能进一步求解上述约束.对于待定关联类型约束 assignable⟨*first*::*associated_type*, *value*⟩,假定最后确定 *first* 对应的类型参数 *Iter* 满足泛型概念 InputIterator,由于 *first*::*associated_type* 是 *first* 的关联对象 **first* 对应的关联类型占位符,根据表 4 所示关联类型约束匹配规则,该占位符应该对应关联类型 reference,从而关联类型约束为 Assignable⟨*Iter*::reference,*T*⟩.这一推导过程我们将在第 4 节做进一步的讨论.

表 4　部分关联类型约束匹配规则

泛型概念	关联类型	表达式形式	匹配规则
Assignable	result_type	operator=(*T*&,*U*)	若 *T* 和 *U* 均为 Assignable,则表达式结果为 *T*::result_type
Dereferenceable	reference;	operator*(*T*)	若 *T* 为 Dereferenceable,则表达式返回类型为 *T*::reference
Addable	result_type	operator+(*T*, *U*)	若 *T* 和 *U* 均为 Addable,则表达式结果为 *T*::result_type
InputIterator⟨*X*⟩	value_type	…	…
	difference_type	…	…
	reference	operator*(*X*);	若 *X* 为 InputIterator,则表达式结果为 *X*::reference
	pointer	operator->(*X*)	若 *X* 为 InputIterator,则表达式结果为 *X*::pointer
	postincrement_result	operator++(*X*&, int)	若 *X* 为 InputIterator,则表达式结果为 *X*::postincrement_result
MutablaForwardIterator	value_type	…	…
	reference	operator*(*X*&)	若 *X* 为 MutablaForwardIterator,则表达式结果为 *X*::reference
	postincrement_result	operator++(*X*&, int)	若 *X* 为 MutablaForwardIterator,则表达式结果为 *X*::postincrement_result

不是所有关联类型都值得关注,目前我们只考虑在 STL 中定义的关联类型.表 4 给出了部分泛型概念的关联类型及其计算规则.

3.3　构造约束集

类型参数的约束,是由所有类型参数声明的属性域、变量[①](包括方法参数、返回值)等的约束决定的.例如图 1 所示程序中,类型参数 *Iter* 的约束,是由 *first* 和 *last* 两个变量的约束决定的.因此,我们先计算所有这些变量的约束,然后将类型参数代入求解,来获得类型参数的约束.本节将详细介绍约束

① 为便于讨论,以下在不会造成误解的情况下统称变量.

集的构造过程.

提取有效表达式约束和关联类型约束构造约束集,可能受到以下两个因素的影响:继承关系和函数调用关系.继承是面向对象程序的一个主要特征,在泛型程序中,继承关系同样是普遍存在的.考虑图 2 所示的一个简单继承关系:类 Derived 是类 Based 的子类.若只考虑子类 Derived,其类型参数 T 只需满足约束:可赋值为整型数.然而,如果使用 int 对子类 Derived 进行实例化显然会出错,因为所实例化的对象调用了从父类 Based 继承来的方法 foo(),而该方法要求类型参数满足可赋值为字符串类型这样的约束.因此,Derived 的类型参数 T 的约束集显然还应包括 Based 的类型参数的约束集.

```
1.  template<typename T> class Based {
2.    public:
3.    T i;
4.    void foo(){
5.      i="hello";
6.    };
7.  };
8.  template<typname T> void test(T i){
9.    //use of i;
10. };
11. template<typename T> class Derived: Base<T>{
12.    public:
13.    T t;
14.    void bar(){
15.      t=1;
16.      test(t);
17.    };
18. };
19. //instantiation of Derived
20. Derived<int>d; //error!
21. d.foo();
```

图 2 继承关系和函数调用影响类型参数的约束

可见,受继承关系的影响,类型参数必须满足在同一类型继承体系中所有父类的对应类型参数的约束.分析时应从类继承体系中当前被分析的类型参数所在类开始,逐层向上分析.

在函数调用中,类型参数所声明的变量可能作为实际参数、返回值等传递,因此,类型参数也要满足被调用函数的潜在需求.例如,图 2 所示程序中,类 Derived 的 bar()函数中有对函数 test()的调用,变量 t 作为实参传递.显然,Derived 的类型参数 T 必须同时满足函数 test 所用类型参数的约束.当类型参数所声明的变量被作为参数进行传递时,必须进行模块间分析,要构造调用图.

为此,约束集的构造算法要构造调用图以及类继承层次图,以确定类型参数的可达作用域.下面先提出该算法的整体框架,然后再对具体的步骤进行分析讨论.给定一个模块 M 及其类型参数 T,我们用 $T\_variables$ 表示决定 T 的约束的变量集合,则 T 的约束集构造的基本过程为

1. 初始化;
   (a) 构造调用图和类继承层次图;
   (b) 确定 T 的可达传播范围;
   (c) 在 T 的可达传播范围内收集所有由 T 声明的变量,加入集合 $T\_variables$;
   (d) 对于 $T\_variables$ 中的每一个变量,分析其别名,也加入集合 $T\_variables$;
2. 分析 $T\_variables$ 中的每一个元素;
   (a) 提取有效表达式约束;
   (b) 提取关联类型约束;
3. 根据有效表达式和关联类型约束构造约束集.

具体算法如算法 1 所示.

算法 1. 类型参数约束集构造算法.

```
Gen_Constraints(T,M)
 Begin
  Construct_Class_ Hierarchy();
  Construct_Call_Graph();
  //Collect_T_Chain()的作用是确定 T 的可达传播范围
  T_variale=Collect_T_Chain();
  Constriants=∅;
  while T_variale≠∅
    从集合 T_variale 中取出一个元素 v;
    Constraints=Constraints∪Analyze_Constraint(v);
  endwhile;
  return Constraints;
 end;
```

首先,通过调用分析函数 Collect_T_Chain 得到变量集合 $T\_variable$,分两步:首先确定类型参数 T 的可达传播范围,解决由于继承关系造成的对类型参数的约束集构造的影响;然后确定类型参数 T 所声明的所有变量的可达传播范围,解决别名引用以及函数调用的影响.经过 Collect_T_Chain 分析,得到变量集合 $T\_variable$.该集合包含了所有由类型参数 T 所声明的变量以及它们的别名.

然后,从集合 $T\_variable$ 中取出一个元素 v,逐条扫描语句,若有对 v 的引用,则进行约束分析.Analyze_Constraint(v)分析变量的使用,并从中提取约束.可以将变量的使用分为 3 类:直接引用、别名引用和参数传递对象引用.

直接引用. 若语句 s 有对变量 v 的直接引用,则分析 v 的有效表达式约束,并将其加入 v 的约束集中.

直接引用比较简单,可以直接从抽象语法树中

直接提取表达式约束,并加入约束集.例如,图 1 实例语句 5 中 $*first$ 就是对变量 $first$ 的直接引用.

别名引用. 对于变量 v,若 u 是 v 的别名,则把 u 的约束也加入 v 的约束集中.

由于我们在构造约束集之前进行了别名的计算,因此,所有使用别名进行引用而产生的约束,都能被加到约束集中.

参数传递引用. 对于变量 v,若有函数调用 $call(\cdots,v,\cdots)$,且 $para\_v$ 是 v 的对应形参,则把 $para\_v$ 的约束也加到 v 的约束集中.

当变量被作为函数调用的参数时,引用情况变得更为复杂.被调用的函数形参在调用开始时是作为实参使用的变量的别名,然而,在函数调用过程中,二者未必具有别名关系.不过,即使形参和实参不再具有别名关系,C++语言对于模板参数约束的实现机制要求仍然需要将形参的约束也加到实参变量的约束集中.因此,必须跟踪函数调用图,分析所对应的函数形参的使用情况.考虑如下例子:

```
void foo(A a){
    //use of a;
    …
    a=new A();
    //use of a1;
}
…
x=new A();
foo(x);
…
```

在函数 foo()中,语句 a=new $A()$之后变量 a 和 x 显然不具有别名关系,然而,foo()体内所有由于 a 的使用而产生的约束,都应该加到 x 的约束集中.

4　泛型概念匹配

计算出变量的有效表达式约束和关联类型约束后,我们将类型参数进行回代,确定类型参数的约束,并进一步确定其满足的泛型概念.

以类型参数 $Iter$ 和 T 分别替换表 3 所示约束集中对应的变量,可以得到如表 5 所示的类型参数 $Iter$ 和 T 的有效表达式约束集.同样,将类型参数 $Iter$ 和 T 代入对应变量的待定关联类型约束,可得关联类型约束:assignable$\langle Iter::associated_type,T\rangle$.

表 5　实例程序 fill 的类型参数 *Iter* 和 *T* 对应的约束集

类型参数	有效表达式约束
$Iter$	copyConstructible$\langle Iter\rangle$, equalityComparable$\langle Iter,Iter\rangle$, preIncrement$\langle Iter\rangle$, deferenceable$\langle Iter\rangle$, assignable$\langle *Iter,T\rangle$
T	assignable$\langle *Iter,T\rangle$

下面将介绍基于约束集推导类型参数应该满足的泛型概念的过程.一个简单直接的思路是:在泛型概念库中查找是否有泛型概念具有约束集中的所有约束,若有,则匹配成功;若没有,可以由约束集构造新的泛型概念,并加入泛型概念库.

先给出如下几个定义.

定义 4(精化关系). 给定泛型概念 C 和 C',若对于所有约束 $c\in C$,有 $c\in C'$,则称 C' 是 C 的一个精化,记作 $C'\mathcal{R}C$.

泛型概念通过精化关系可以形成一个泛型概念框架.例如,STL 中就定义了一个算法和数据结构的泛型概念框架,包括容器、算法、迭代子、适配器、函数对象等多种泛型概念.

定义 5(最小泛型概念). 给定泛型概念框架及其中泛型概念 C,某个约束 c,且 $c\in C$,若除了自身外,不存在其它泛型概念 C' 具有约束 c 且 C 是 C' 的精化,则称 C 是具有约束 c 的一个最小泛型概念.若 c 是空约束,则 C 是该泛型概念框架中的顶层元素.

定义 6(最大泛型概念). 给定泛型概念框架及其中的一个泛型概念 C,某个约束 c,且 $c\in C$,我们称 C 是具有约束 c 的一个最大泛型概念,当且仅当除了自身外不存在其它泛型概念 C', $c\in C'$ 且 C' 是 C 的精化.

定义 7(最小公共精化泛型概念). 给定某泛型概念框架中的泛型概念 $C_1,\cdots,C_n$,若存在 C', $C'\mathcal{R}C_1,\cdots,C'\mathcal{R}C_n$,且对于任意 $C''\neq C'$,$C''\mathcal{R}C_1,\cdots,C''\mathcal{R}C_n$,有 $C''\mathcal{R}C'$,则称 C' 是 $C_1,\cdots,C_n$ 的最小公共精化泛型概念.

若一个泛型概念具有某个约束,则其所有精化泛型概念都具有该约束.因此,在推导泛型概念时我们只需查找具有该约束的最小泛型概念.

根据定义 4~6,我们可以得到实例程序 fill 的约束集及其对应的最小泛型概念,如表 6 所示.

表 6　实例程序 fill 的类型参数 *Iter* 和 *T* 的约束及其对应的最小泛型概念

有效表达式约束	对应的最小泛型概念
copyConstructible$\langle Iter\rangle$	CopyConstructible
equalityComparable$\langle Iter\rangle$	EqualityComparable
preIncrement$\langle Iter\rangle$	InputIterator, OutputIterator, BasicOutputIterator, Integal
dereferenceable$\langle Iter\rangle$	Dereferenceable
assignable$\langle *Iter,T\rangle$	Assignable

图 3 列出了与实例程序 fill 相关的泛型概念框架的一部分. 虚线框内是表 6 所示包含有相应有效表达式约束的最小泛型概念. 从图中可以看出,同时具有类型参数 *Iter* 应满足的所有约束的最小泛型概念是 MutableForwardIterator.

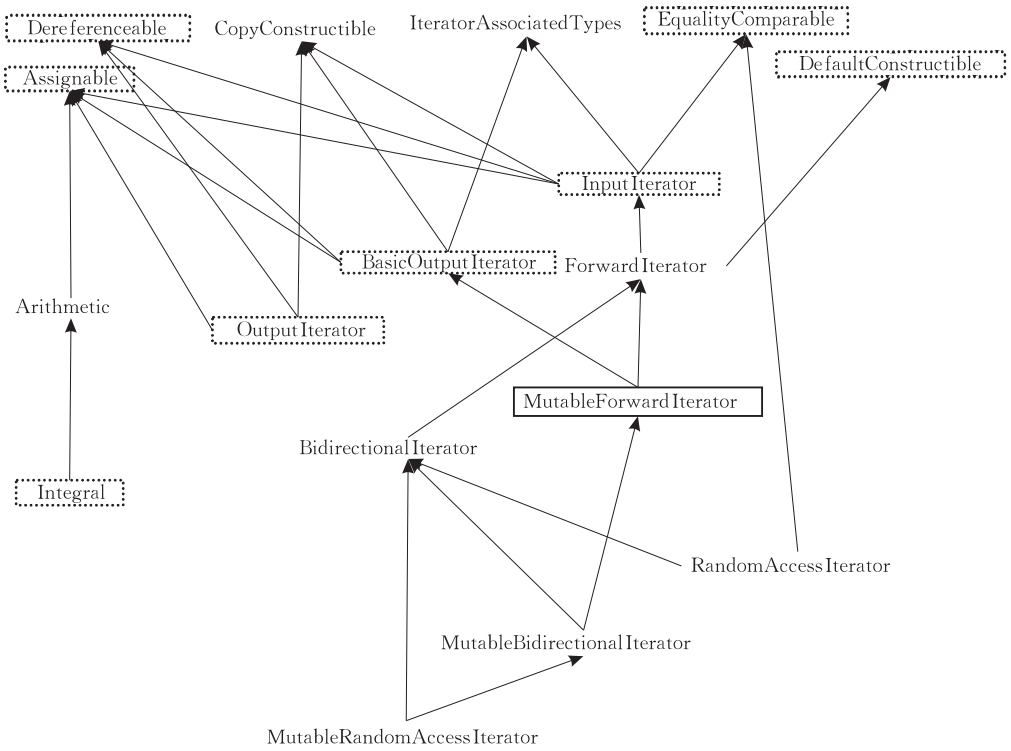


图 3 C++0x 中定义的部分概念层次结构

我们的目标是希望找到如 MutableForwardIterator 一般的最小泛型概念,能同时包含所有约束. 一个直接的方法是查找所有约束对应的最小泛型概念的最小公共精化泛型概念. 然而,观察图 3 可以发现:① 最小公共精化泛型概念有时候是不存在的. 如图 3 所示,Integral 和 OutputIterator 不存在公共精化泛型概念. ② 并非所有最小泛型概念都是有用的. 例如,Assignable 是包含约束 assignable〈*Iter*, *T*〉的最小泛型概念,然而,由于包含约束 preIncrement〈*Iter*〉的最小泛型概念 BasicOutputIterator 同时也是 Assignable 的精化,即它也同时包含约束 assignable〈*Iter*, *T*〉,因此,不需要同时考虑 BasicOutputIterator 和 Assignable. 此外,OutputIterator 与 BasicOutputIterator 同样是具有约束 assignable〈*Iter*, *T*〉的最小泛型概念,两者只需选择其一.

针对上述问题,我们提出了一种泛型概念匹配算法,基本步骤如下:

1. 对于每个约束,找到具有该约束的最小泛型概念集,记作 $C_1,\cdots,C_n$;

2. 从每个约束对应的最小泛型概念集中各取一个泛型概念 $c_1,\cdots,c_n$;

3. 查找 $c_1,\cdots,c_n$ 的最小公共精化泛型概念;

4. 重复步 2 和步 3.

步 2 从每个约束对应的最小泛型概念集中各取一个泛型概念,保证了最后匹配的泛型概念能满足所有约束.

步 3 查找最小公共精化泛型概念,其结果可能为空. 例如,Integral 和 OutputIterator 不存在公共精化泛型概念. 为了避免最后返回结果为空,我们在匹配过程中保存中间结果,并允许程序员从中选择. 例如,从表 6 中取出泛型概念集合

{CopyConstructible, EqualityComparable,
Integal, Dereferenceable, Assignable},

其中{CopyConstructible, EqualityComparable, Dereferenceable, Assignable}具有最小公共精化泛型概念 MutableForwardIterator,然而,它们和 Integal 没有公共精化泛型概念. 我们在匹配过程中记录下

MutableForwardIterator&&Integal

供程序员检查选择.

表 7 列出了实例程序 fill 的泛型概念匹配结果.

表 7　实例程序 fill 的类型参数 *Iter* 的泛型概念匹配结果

最小泛型概念	最小公共精化泛型概念
CopyConstructible, EqualityComparable, InputIterator, Dereferenceable, Assignable	MutableForwardIterator
CopyConstructible, EqualityComparable, OutputIterator, Dereferenceable, Assignable	OutputIterator && MutableForwardIterator
CopyConstructible, EqualityComparable, Integal, Dereferenceable, Assignable	MutableForwardIterator
CopyConstructible, EqualityComparable, Integal, Dereferenceable, Assignable	MutableForwardIterator && Integral

从表 7 可以看出,虽然 MutableForwardIterator 是最佳结果,其它结果在一定程度上也反映了原程序的设计意图,记录下它们有利于理解程序.

在推导出候选泛型概念后,还要进一步求解其关联类型约束.对于图 1 所示实例,应该对关联类型约束 assignable〈*Iter*::*associated_type*,*T*〉做进一步求解并在重构后程序中指定.

在关联类型约束 assignable〈*Iter*::*associated_type*,*T*〉中,*Iter*::*associated_type* 只是个占位符,一旦确定了 *Iter* 所匹配的候选泛型概念,就可以根据表 4 所示的关联类型约束匹配规则,确定该占位符的类型.对照表 4,如果 *Iter* 匹配 MutableRandomAccessIterator,根据表达式形式 **Iter*,那么可以确定 *Iter*::*associated_type* 应该为 *Iter*::reference.

5　实例研究

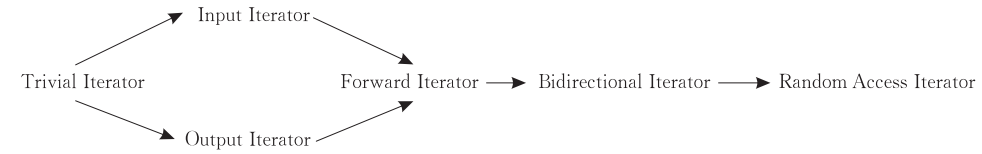
STL 是泛型程序设计方法的一个典范,且 C++0x 草案中已经对 STL 中的所有算法都使用泛型概念进行了重新定义.因此,我们将 STL 作为实例进行了分析,并将泛型概念的识别结果和现有其他工作以及 C++0x 草案中的标准定义进行了对比.

待重构代码

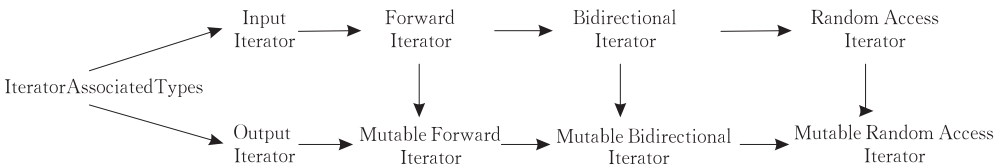
我们选取了 STL 中的算法作为待分析代码,这些代码的主要特点是:模块化程度很高,除了函数调用的参数引用外,没有其它别名存在;算法都是以函数代码形式出现,没有类定义.因此,不需要分析继承关系来跟踪分析类型参数的约束.函数调用在 STL 算法代码中仍是比较普遍的操作,必须跟踪调用图对参数做进一步分析.例如,mismatch 算法中调用了 pair 的构造函数,find_end 算法先调用了_find_end 函数,在_find_end 内部又调用了 search 算法,这些调用都必须逐步跟踪.

泛型概念库

泛型概念库的构建是影响分析精度的一个重要因素.例如,在 STL 中定义的迭代子泛型概念框架如图 4(a)所示.在 STL 的定义中,前向迭代子的有效表达式集合和输入迭代子的是一样的,输出迭代子的有效表达式集合包含了输入迭代子的所有有效表达式集合.这种有效表达式约束定义和迭代子精化框架定义存在矛盾,不能有效区分输出迭代子和前向迭代子.



(a) STL中的迭代子框架



(b) C++0x中的迭代子框架

图 4　两个不同的迭代子框架

我们参照 C++0x 标准草案 N2082、N2083、N2084、N2085 等构建了泛型概念库①,这里定义的泛型概念和原来 STL 中的泛型概念有些不同,如图 4(b)所示.首先,增加了若干可变迭代子泛型概念;其次,泛型概念的有效表达式约束集与 STL 中的定义不一样,例如,输出迭代子除了包含前向迭代子的有效表达式集外,还包括"解引用后赋值"表达式.根据这些定义构建的泛型概念库中的概念更明确,能明显区分输出迭代子和前向迭代子.

目前我们构建的泛型概念库的主要局限在于:

① C++ Standards Committee Papers. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/

约束集仅包含了有效表达式约束和关联类型约束，C++0x 定义的其它一些约束没有在泛型概念库中给予定义. 例如图 5 所示为输入迭代子的完整定义，其中诸如

SameType〈Assignable〈*X*〉::result_type, *X*&〉

```
concept InputIterator〈typename X〉
  :IteratorAssociatedTypes〈X〉, CopyConstructible〈X〉,
   EqualityComparable〈X〉{
  where Assignable〈X〉 &&
        SameType〈Assignable〈X〉::result_type, X&〉;
  where SignedIntegral〈difference_type〉&&
        Convertible〈reference, value_type〉&&
        Convertible〈pointer, const value_type*〉;
  typename postincrement_result;
  where Dereferenceable〈postincrement_result〉&&
        Convertible〈Dereferenceable〈postincrement_result〉::
                    reference, value_type〉;
  pointer operator->(X);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
  reference operator*(X);
};
```

图 5 C++0x 中输入迭代子的完整定义

这样的嵌套约束我们没有考虑. 主要是因为这个约束考虑的是 Assinable 的关联类型 result_type 和类型参数 *X* 之间的关系，我们目前无法识别 *X* 的某个关联类型就是 Assinable 的关联类型 result_type. 在对 STL 的实例研究中，我们发现不考虑这种约束并不影响识别的最终效果.

方法效果

表 8 列出了我们的方法和现有方法的部分结果对比，所列算法包含了 STL 的主要算法种类：处理单一容器的算法、处理多个容器的算法、涉及函数对象的算法以及进行数据处理的算法等. 算法一栏列出了我们所分析的 STL 算法名称和该算法携带的类型参数，Sutton 一栏是 Sutton 等人的方法分析结果[10]，Chen 代表本文方法，C++0x 一栏列出的是 C++0x 草案中对该算法的标准定义. Sutton 等人的方法基于类型参数的使用分析，从 C++ 函数模板中识别泛型概念. 在表 8 中，在不影响理解的前提下使用了泛型概念的简写，此外，用“&&”连接符连接所有约束.

表 8 相关工作效果比较

算法	Sutton	Chen	C++0x
fill 〈*Iter*, *T*〉	MutFwdIter〈*Iter*〉	MutFwdIter〈*Iter*〉&& Assignable〈*Iter*::ref, *T*〉	MutFwdIter〈*Iter*〉&& Assign〈*Iter*::ref, *T*〉
find 〈*Iter*, *T*〉	MutFwdIter〈*Iter*〉	InIter〈*Iter*〉&& Equal〈*Iter*::ref, *T*〉	InIter〈*Iter*〉&& Equal〈*Iter*::ref, *T*〉
find_if 〈*Iter*, *Pred*〉	MutFwdIter〈*Iter*〉&& CopyCtor〈*Pred*〉	InIter〈*Iter*〉&& (Predicate〈*Pred*〉&& CopyCtor〈*Pred*〉)	InIter〈*Iter*〉&& Predicate〈*Pred*〉
find_end 〈*Iter*1, *Iter*2〉	MutFwdIter〈*Iter*1〉&& MutFwdIter〈*Iter*2〉	FwdIter〈*Iter*1〉&& FwdIter〈*Iter*2〉&& Equal〈*Iter*1::ref, *Iter*2::ref〉	FwdIter〈*Iter*1〉&& FwdIter〈*Iter*2〉&& Equal〈*Iter*1::ref, *Iter*2::ref〉
for_each 〈*Iter*, *Func*〉	MutFwdIter〈*Iter*〉&& CopyCtor〈*Func*〉	InIter〈*Iter*〉&& (Callable〈*Func*〉&& CopyCtor〈*Func*〉)	InIter〈*Iter*〉&& Callable〈*Func*〉
accumulate 〈*Iter*, *T*〉	MutFwdIter〈*Iter*〉&& RanAccessIter〈*T*〉	InIter〈*Iter*〉&& (Callable〈*T*〉 ‖ BinaryOp〈*T*〉)	InIter〈*Iter*〉&& BinaryOp〈*T*〉

与 Sutton 等人方法相比，我们的方法在识别效果上更佳：

(1) Sutton 等人方法所识别的类型参数若不包含“解引用后赋值”操作，则无法区分迭代子的实际类型，因此，在其分析结果中，除了 fill 算法，其他结果均为 MutableforwardIterator，显然不精确. 我们的分析在类型参数不包含“解引用后赋值”操作的情况下，利用最小泛型概念的查找算法能得到更精确的结果.

(2) 在泛型概念定义中，Predicate 是 Callable 的精化概念，前者比后者多了一项表达式约束：类型可转换为 bool 类型. Sutton 等人的方法无法利用这一表达式约束识别出 Predicate 概念. 而我们的方法通过分析关联对象是否存在比较操作，可识别出这一有效表达式，从而抽取出 Predicate 概念.

(3) 我们的方法考虑了关联类型约束，并将其抽取出来，结果更接近 C++0x 的标准代码. 如算法 find、find_end 等.

与 C++0x 的标准定义的比较反映出我们的方法仍存在一些有待改进的地方.

首先，有些有效表达式在程序中普遍存在，导致某些诸如 DefaultConstructibale、Integral 等最小泛型概念出现的概率很大. 因此，如果泛型概念库包含有较多不相干的泛型概念，匹配效果会较差. 如第 4 节所述，fill 匹配的中间结果包含{Integral && ForwardIterator}，显然 Integral 和 ForwardIterator 是两个语义相差较远的泛型概念，仅仅由于前自增有效表达式约束，使得同时抽取出两个泛型概念.

其次,在分析 find_if 算法时,发现 Pred 同时具有 copyConstrutor 和 call 两个表达式约束,然而,对应的泛型概念 CopyConstructible 和 Callable 在 C++0x定义中没有最小公共精化泛型概念,只能得到(Predicate && CopyConstructible)这样的结果.出现该结果的原因是:虽然满足 Callable 的类型参数未必满足 CopyConstructible,然而,由于 CopyConstructible 在程序中普遍存在,绝大部分类型参数都满足该泛型概念,因此,在很多情况下,不必把该约束写到算法定义中.也就是说,虽然 find_if 算法使用的类型参数必须满足 CopyConstructible 泛型概念,然而不必要将其写入算法定义中.算法 for_each 也存在同样问题.

再次,在某些情况下无法区分 Callable 和 BinaryOperation 两个泛型概念,如算法 accumulate 的结果.当被分析的有效表达式带两个参数时,这两个泛型概念包含的有效表达式形式是相同的,均为形如"operator ($x1$,$x2$)"的操作,因此无法区分.

最后,在实现中我们使用了 srcML 从源代码中提取有效表达式[11],由于该工具的限制,无法识别某些有效表达式,如操作符重载、函数调用和类型转换操作等.在这种情况下,需要一定的人工介入.

6　相关工作

目前支持泛型程序设计方法的主要语言设施是参数化类型.C++0x 直接在语言层面支持泛型概念,提供泛型概念的检查,能够大大提高编译器的能力,有利于泛型程序设计方法的使用.除了 C++0x,目前尚未见其他类似实现.

泛型概念库的构建依赖于 C++0x 的语言定义.直接在语言层面支持泛型概念,受语言其他设施的约束,抽象层次降低,所定义的泛型概念不容易用于软件需求、分析和测试阶段.因此,用于泛型概念表示的形式化或半形式化方法可以借鉴用于泛型概念库的构建,如 Tecton 和 Caramel. Tecton 是一个用于泛型概念描述和进行形式化验证工作的系统[12],采用形式化方法描述泛型概念,可以在高抽象层次上讨论泛型概念,形式化地证明某个类型是否符合特定的泛型概念,从而验证算法的正确性.然而对于许多应用,这种形式化方法过于严格,缺乏工具的支持,不能方便地用于软件开发.Caramel 是一个泛型概念表示标记语言系统[13],使用 XML 表示泛型概念,可以由 XML 文档生成泛型概念检查类(用于编译时检查泛型概念实现是否和需求一致)、泛型概念原型类(用于测试算法)和文档,对算法实现进行验证.然而,由于 XML 文档本身不含有语义信息,因此这种表示可能存在二义性,并且不能实现语义信息的推理.

现有对类型参数的使用分析方法大多基于类型分析,所处理的语言为 Java 语言[14-17].基于类型分析的方法,适用于对程序进行泛型化重构,即分析程序中变量使用,为非泛型程序添加类型参数,使程序更具有通用性,类型安全性更高.如文献[14-17]均是通过指向分析和类型分析技术确定类型参数.我们也提出了一种基于类型分析的类型参数确定方法,并用于 Java 泛型程序重构[8].

Siff 等人给出了一种将 C 语言程序重构为 C++模板程序的方法[14].这也是一种基于类型分析的泛型化重构方法.通过对类型参数的使用分析,提取类型参数,达到"去类型化"的效果.

文献[7]提出了一种利用泛型概念对 C++模板程序实施重构的方法,其目的与本文相似,均期望从源代码中抽取出泛型概念.然而,该方法需要较多的人工参与,自动化程度较低,更多的是为泛型概念抽取提供指导作用.

目前我们已知的与本文工作最为相似的是 Sutton 等人的研究[10].他们尝试将 C++模板函数重构为 C++0x 标准的代码,对 STL 进行了分析.他们先从表达式中提取约束,然后利用形式化泛型概念分析方法对所得约束进行分析,以确定约束集所对应的泛型概念.与 Sutton 等人的方法相比,本文工作的主要不同在于:(1)考虑了关联类型约束,提高了约束分析的精度,使得结果更接近于 C++0x 标准的代码;(2)分析了由于继承、函数调用、别名等因素造成的约束,使我们的方法更通用,不仅同时适用于函数模板和类模板,也提高了约束分析的精度;(3)我们通过查找最小泛型概念和最小公共精化泛型概念匹配结果,并记录了匹配的中间过程,在一定程度上更全面地反映了泛型程序的设计意图.

7　结　语

本文提出了一种基于源代码静态分析的 C++0x 泛型概念抽取方法,通过分析遗产代码中类型参数的使用情况,从程序中提取有效表达式约束和关联类型约束,构造约束集,进而推导出合理的泛型概念,可以指导将遗产 C++模板代码重构为符合

C++0x 标准的代码.与现有工作相比,方法的主要优势在于:方法不仅适用于模板函数,也适用于模板类;分析了函数调用、类继承关系以及别名关系的影响,提高了约束集构造的精度;考虑了关联类型约束,使得识别效果更接近于 C++0x 标准代码;使用 C++0x 标准构建泛型概念库,更利于指导泛型概念的识别和重构的实施.

未来工作包括进一步发展软件工程领域的泛型概念,完善泛型概念库.泛型概念库的建立,不仅有利于指导实施重构,也能支持泛型概念和实例查询与验证以及其他软件分析活动,如识别代码的坏味道、软件度量等.

参 考 文 献

[1] Austern M H. Generic Programming and the STL. Professional Computing Series. Boston: Addison-Wesley, 1999

[2] Andrei Alexandrescu. Modern C++ Design: Generic Programming and Design Patterns Applied. Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 2001

[3] Plauger P, Stepanov A A, Lee M, Musser D R. The C++ Standard Template Library. Prentice Hall PTR, 2000

[4] Gregor D, Järvi J, Siek J, Stroustrup B, DosReis G, Lumsdaine A. Concepts: Linguistic support for generic programming in C++//Proceedings of the Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06). Portland, Oregon, USA, 2006: 291-310

[5] Siek J, Lumsdaine A. Language requirements for large-scale generic libraries//Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE'05). Tallinn, Estonia, 2005. New York: ACM, 2006: 405-421

[6] Garcia R, Järvi J, Lumsdaine R et al. A comparative study of language support for generic programming//Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, Anaheim, California, USA, 2003: 115-134

[7] Chen Lin, Xu Bao-Wen, Zhou Tian-Lin, Shi Liang, He Yan-Xiang. Refactoring C++ programs with concepts//Proceedings of the 9th IASTED International Conference on Software Engineering and Applications. Phoenix, AZ, USA, 2005

[8] Chen Lin, Xu Bao-wen, Zhou Tian-lin, Zhou Yu-Ming. Applying generalization refactoring to Java generic programs//Proceedings of the 1st IEEE International Workshop on Semantic Computing and Systems. Huangshan, China, 2008: 35-39

[9] Zhou Yu-Ming, Xu Bao-Wen. An object-extracting approach using module cohesion. Journal of Software, 2000, 11(4): 557-562(in Chinese)
(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)

[10] Sutton Andrew, Maletic Jonathan I. Automatically identifying C++0x concepts in function templates//Proceedings of the 24th IEEE International Conference on Software Maintenance (ICSM 2008). Beijing, China, 2008: 57-66

[11] Collard M L, Kagdi H, Maletic J I. An XML-based lightweight C++ fact extractor//Proceedings of the 11th IEEE International Workshop on Program Comprehension. Portland, Oregon, USA, 2003: 134-143

[12] Musser D R. The tecton concept description language. Universität Tübingen, South Dakota, 1998

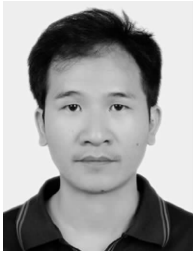
[13] Willcock Jeremiah, Siek Jeremy, Lumsdaine Andrew. Caramel: A concept representation system for generic programming//Proceedings of the 2nd Workshop on C++ Template Programming. Tampa Bay, FL, USA, 2001

[14] Siff M, Reps T. Program generalization for software reuse: From C to C++//Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering. New York: ACM, 1996: 135-146

[15] von Dincklage D, Diwan A. Converting Java classes to use generics//Proceedings of the ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications. Vancouver, Canada, 2004: 1-14

[16] Donovan A, Kiezun A, Ernst M D. Converting Java programs to use generic libraries//Proceedings of the ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications. Vancouver, Canada, 2004: 15-34

[17] Kieżun A, Ernst M D, Tip F, Fuhrer R. Refactoring for parameterizing Java classes//Proceedings of the 29th International Conference on Software Engineering (ICSE'07). Minneapolis, MN, USA, 2007: 437-446



CHEN Lin, born in 1979, Ph. D. candidate. His current research interests include software analysis and software refactoring.

XU Bao-Wen, born in 1961, Ph. D., professor, Ph. D. supervisor. His current research interests include programming language, software engineering (software methodology, software analysis, software metrics and software testing), and Web technology.

Background

Generic programming has become more and more popular in software development area. The basis of generic programming is to use Concepts to abstract domain knowledge and provide solid base for developing reusable and extendable generic software. Concept will be a new language feature in C++0x. Refactoring legacy programs to use generics is an important issue in reverse engineering. There are two different works in refactoring codes to use generics. One is introducing type parameters to programs. This work is called parameterization. It is usually done using type inference and pointer analysis. Another work is refactoring legacy C++ programs to use Concepts. This work can be seen as a deeper work based on parameterization. Usually a Concept contains two kinds of constraints: valid expression constraints and associate type constrains. The existing works failed to extract associate type constraints from codes, so the results are not very precise.

This paper presents an automatic method to identify Concepts based on static analysis. The paper analyzes the usages of type parameters in legacy programs to extract valid expression constraints and associate type constraints. Concepts can be inferred by these constraints. The authors apply the approach to C++ Standard Template Library (STL) to identify most of the potential Concepts in STL. The study convinces that this method is helpful to identify Concepts in legacy C++ programs.

The work in this paper is partially supported by the National Natural Science Foundation of China (90818027, 60633010), the National High Technology Development Program (863 Program) of China (2009AA01Z147), and the MOE-INTEL Information Technology Special Research Program (MOE-INTEL-08-12).