文章编号:1000 - 5862(2019)05 - 0454 - 08

Apla 与程序设计语言泛型特性比较研究

左正康[1],刘志豪[1],黄 箐[1],游 珍[2],王昌晶[1*],石海鹤[1],胡启敏[2],陶小明[1]
(1. 江西师范大学计算机信息工程学院,江西 南昌 330022;
2. 江西师范大学江西省高性能计算重点实验室,江西 南昌 330022)

摘要: 面向对象编程(OOP)是以对象为编程核心,而泛型程序设计(GP)是根据一个类型所需要的操作需求进行分类和抽象,即 GP 是一种以类型需求为核心的程序设计范式. 描述完整的 GP 类型需求包含静态语法约束和动态语义约束,主流程序设计语言已经支持静态语法约束,但受语言其他方面的限制,抽象层次较低,定义的泛型概念难以描述及验证基于动态语义的复杂约束需求问题,与完整实现 GP 尚有距离. 该文综合比较了 Java、C#、C++、Concept 4 种支持泛型程序设计的语言,基于抽象程序设计语言 Apla 提出了离完整实现 GP 更近一步的泛型机制,通过同典型泛型实例对各种程序设计语言泛型特性进行分析,验证抽象泛型程序设计语言 Apla 中该泛型机制的优越性.

关键词: 泛型程序设计; 动态语义约束; 抽象程序设计语言
中图分类号: TP 311 **文献标志码:** A **DOI:** 10.16357/j.cnki.issn1000-5862.2019.05.03

0 引言

泛型是程序设计语言的一种特性[1-2],泛型(Generic)又称为参数化类型,即把数据变量的类型作为参数来进行传递. 1968 年 M. D. McIlroy[3] 提出可重用软件部件的概念,首次提出了可重用部件要力求泛型(generality). 20 世纪 70 年代,M. Zalewski 等[4] 和 J. C. Reynolds[5] 提出 System F,在 System F 中引入了多态性演算,该演算在类型引入时首创性地提出了全称量词的机制,形式化地描述了参数化多态的概念. 20 世纪 80 年代,D. Musser 等正式提出了泛型程序设计的概念及其规则,在此阶段出现了支持参数化的类型机制设计语言,如 Ada、C++等. 20 世纪 90 年代,A. Stepanov 等在原来 Ada 程序库基础上,改用 C++语言的模板机制重新设计了一个程序库,即标准模板库(standard template library,简称 STL)[6],该库包含了诸多计算机科学领域常用的基本数据结构和算法. STL 广义上分为 3 大类:algorithm(算法)、container(容器)和 iterator(迭代器)[7]. 泛型在 C++中的应用均采用了模板类和模板函数的方式,更好地提供了代码复用的机会. 2004 年 2 月 Sun 公司在 J2SE1.5 版本中加入了泛型,允许对所操作的数据类型进行抽象为一个参数类型,在使用时指定具体的类型,使强类型在编译时进行类型检查. C++可以通过模板技术对集合元素类型指定,而 Java 在 1.5 版本以前是没有相应功能的,在引入泛型特性[8-9] 后它允许集合指定元素类型. Java 泛型的使用也具有某些限制,比如不能创建泛型数组、不可以将泛型用在异常处理中等[10]. C# 在 2.0 版本中引入几项语言扩展,其中就包括泛型. C#和 Java 一样,泛型已经广泛应用在泛型类、泛型方法、泛型接口和其他容器以及对容器操作的一些方法中,泛型类和泛型方法集软件部件复用、类型安全和高效率于一身,与之前的非泛型编程比较,泛型编程给代码编写带来了极大的便利.

1 相关工作

C. Hall 等[11] 指出许多现代编程语言支持泛型,并综合比较了 8 种编程语言 C++、标准 ML、Objective Caml、Haskell、Eiffel、Java、C#和 Cecil 的泛

收稿日期:2019-01-08
基金项目:国家自然科学基金(61862033,61462039,61762049,61662035,61662036)和江西省教育厅科技课题(GJJ160329)资助项目.
通信作者:王昌晶(1977-),男,江西南昌人,教授,博士,主要从事软件形式化方法、形式规格说明方法和 Web 服务方面的研究. E-mail:wcj771006@163.com

型编程特性,通过在每种语言中实现一个重要的示例,说明泛型编程中的基本角色是如何在每种语言中进行表示的,进一步确定 8 种语言属性,支持泛型编程这种广泛的观点,尤其是在 Haskell 中新的递归特性,这些特性对于避免笨拙的设计、糟糕的可维护性和冗长的代码是很有必要的[11].

P. Maciol[12] 在时间和空间上进行泛型编程,指出元编程是专业化解决软件工程中抽象与性能权衡最有希望的候选方案之一. 他们将泛型编程的范围扩展到时间维度,由此产生的阶段多态性的概念,使他们在开发和探索程序生成器的设计中有了新的抽象. 并且呈现一种实现的可能性,在 Scala 中使用轻量级模块化分段(LMS)框架,并将其应用于 2 个重要的案例研究: 图像上的卷积和快速傅里叶变换[13].

薛锦云等[14] 介绍了 PAR 平台支持模型驱动的软件工程(MDE),包括算法建模语言 Radl、抽象程序建模语言 Apla、模型转换的一套规则和一套算法模型和程序模型的自动转换工具. 目前典型的编程语言的通用机制(比如 C++、Java 等)还不够,只有数据类型可以是通用的形式参数. PAR 平台的显著特征之一是敏捷的通用机制. 不仅是值、数据类型和计算操作可以是通用参数,而且自定义抽象数据类型(ADT)也可以作为通用参数. 文献[14]提出了类型区域、动作区域和 ADT 区域的新概念,可以明显提高通用软件的安全性,并对泛型编程(GP)给出了新的定义: GP 是一个参数化编程,其中参数可以是数据(值)、数据类型、操作(包括操作符、子程序、功能、方法和程序)、ADT、组件和服务等.

孙斌[15] 提出了命名类型约束机制,其选取 C++语言作为宿主语言,类型约束的语法设计与原始的 C++语言风格及其设计思想保持一致; 按照泛型程序设计的方法和新的语法原则设计了基础性类型约束库; 开发出一个编译器前端(集成到已有的 C++前端中),对涉及类型约束代码部分进行属性值计算和分析检测. 命名类型约束机制实现的基本策略是尽可能地重用现有的C++资源.

丁志义等[16] 总结了泛型程序设计的作用,在程序设计的过程中,根据需要将类型或者操作当作操作传递参数,并在此基础上编写通用性更高的设计性程序. 此外在该过程中还总结了泛型程序设计中的 4 种技术: 模板偏特化、常整数映射为型别、型别对型别的映射以及一种组件泛化仿函数.

徐文胜[17] 对泛型编程的核心思想和技术特征进行了较为深入的分析,介绍了泛型编程在语言实现上的现状与不足,着重论述了针对这些不足做出的改进工作,即对类型参数及其约束机制进行扩展以支持通用、高效的算法和数据结构的设计,并以 Java 语言作为实施例,详细介绍了如何通过现有对象技术来实现比较完整的泛型编程.

该文以泛型语言特性为研究目标,对比了 4 种支持泛型编程的语言 Java、C#、C++、Concept 的泛型特性,并且以抽象程序设计语言 Apla 为宿主语言,提出了自定义抽象数据类型的泛型约束机制,同时支持静态泛型语法层约束,拓展了泛型程序设计约束的应用范围.

2　泛型编程

泛型编程最初诞生于 C++中,由 A. Stepanov 等[18] 和 D. R. Musser[19] 创立,最初提出的动机很简单: 发明一种语言机制,帮助实现一个通用的标准容器库. 其语言支持机制就是模板(Template),模板的核心是参数化类型,即把原本特定于某个类型的算法或类当中数据类型信息抽象成一个模板参数 T. STL 是迄今为止最为成功和卓越的泛型编程范例,其本身是 C++的模板但其算法是泛型的,不与任何特定的数据结构或对象类型耦合在一起. 它以极高的抽象程度、通用性和出色的运行效率实现了 C++程序设计中涉及的数据结构和算法,其抽象能力和运行效率完全是在泛型编程的方法和原则指导下获得的.

A. Stepanov 等[18] 开始探讨一种新的角度来研究计算机算法和数据结构,发现根据数据类型需求的基本条件来进行分类,可以极大地简化程序,设计出同时具备高度抽象性、通用性和高效率的数据结构和算法. 泛型编程从某种角度来说是面向对象编程的进一步扩展,都是在软件开发时剔除不稳定因素,用具有普遍性和通用性的稳定基础构建稳定系统. GP 提供了一种比面向对象抽象性更为高层次的方法(根据一个类型操作所需满足的需求来分类并抽象). GP 的抽象单位称之为“概念(Concept)”,表示满足一组类型需求的对象类型的集合. 为此“概念”的定义可以表示为[15-21]: 设 T 为一个抽象类型的集合,GP 中一个概念 C 为 T 中满足某个需求集合 D 的所有元素集合,即

$$C(D \mid T) = \{ t \in T \mid \forall d \in D, t(D) = \text{true} \}.$$

在某种支持泛型编程的程序语言中,T 为该语言所允许的所有类型集合,D 为某个类型需求条件的集合,概念 C 则是抽象类型集合 T 中满足一组类

型需求 D 的所有元素的集合. 因此 GP 概念是一组由类型需求确定的类型集合,其实质是对类型需求的抽象. 由此可见 GP 是一种以类型需求为核心的程序设计模式.

3 程序设计语言泛型特性对比研究

通过分析同一典型泛型实例在各种程序设计语言的实现方式来描述各种程序设计语言的泛型实现机制.

3.1 Java 泛型特性

(i) 定义泛型约束 Comparable.

```
interface Comparable <T>{ boolean best( T x) ;}
```

(ii) 定义泛型方法 sort,该方法要求其类型参数 T 必须实现泛型约束 Comparable.

```
public class select{
    public static <T extends Comparable <T> >
      T [] sort( T a []) {
    for ( int i =0; i <a. length -1; i + +)
    { for ( int j = i +1; j <a. length; j + +)
        { if(! a [j]. best( a [i]) )
    { T temp = a [i]; a [j] = a [i];
    a [j] = temp; } } }
    return a; } }.
```

(iii) Dog 类实现泛型约束 Comparable.

```
public class Dog implements Comparable <Dog>{
    public int height;
    public boolean best( Dog y)
    { return height > y. height; } }.
```

(iv) Cat 类实现泛型约束 Comparable.

```
public class Cat implements Comparable <Cat>{
    public String name;
    public boolean best( Cat y) {
      if( name. compareTo( y. name) >0)
      return true; else
      return false; } }.
```

(v) 分别用 Dog 类和 Cat 类对象实例化泛型方法 sort.

```
public class Main {
    public static void main( String [] args) {
      int n; String m;
      Dog a [] = new Dog [n];
      select. sort( a) ;
      Cat b [] = new Cat [m];
      select. sort( b) ; } }.
```

代码在编译时检查参数的类型是否符合继承自接口 Comparable 的类型参数 T,只有参数类型满足泛型约束 Comparable 的类型才能通过编译执行. Java正因为引进这一特性使得代码在编译时期及时发现并处理错误,提高了执行代码的安全和可靠性. 但是这只是一种狭义的约束,主要是一种基于子类型约束的多态机制,这类约束描述的泛型需求过于狭隘,不能完整地描述泛型需求.

3.2 C#泛型特性

(i) 在 C#语言中同样是定义泛型约束 Comparable 和类型参数为 T 的 best 方法.

```
public interface Comparable < T > { bool best
( T x) ; }.
```

(ii) 定义泛型方法 sort,该方法要求其类型参数 T 必须实现泛型约束 Comparable. 其泛型约束调用是通过关键字 where 实现.

```
public static class select{
public static T sort < T >( T [] a) where T:
    Comparable < T >
{ for( int i =0; i < a. Length -1; i + + ) {
    for ( int j = i +1; j < a. Length; j + + ) {
    if(! a [j]. best( a [i]) )
    temp = a [i]; a [j] = a [i]; a [j] = temp; } }
    return a; } }.
```

(iii) Dog 类实现泛型约束 Comparable.

```
class Dog: Comparable < Dog > {
    public int height;
    { return height > y. height; } }.
```

(iv) Cat 类实现泛型约束 Comparable.

```
class Cat: Comparable < Cat >{
    public string name;
    public bool best( Cat y) {
      if ( name. CompareTo( y. name) >0)
      return true; else
      return false; } } }.
```

和 Java 的实例化相同,只有满足 Comparable 约束条件的类型参数方可通过编译执行,这样大大地提高了代码的灵活性,显著缩短了开发时间. C#使用 where 关键字实现约束调用,其约束类型可以是一个类或接口类型的列表,也是基于子类型约束的多态制,所以只能称之为狭义的约束.

3.3 C + +泛型特性

(i) 定义泛型参数为 T 的 sort 函数模板.

```
template <class T> const T* sort( T a [],int
```

```
    length) {
  for ( int i = 0; i < length - 1; i + + ) { s = i;
  for ( int j = i + 1; j < length; j + + ) {
    if ( a [s] > a [j] ) s = j; }
    if ( s! = i) {
    temp = a [i]; a [i] = a [s];
    a [s] = temp; } }
    return a; } .
```

(ii) Dog 和 Cat 类中都实现了返回类型为 boolean 类型的 best 函数.

```
class Dog{ public: int height;
    Dog( int i) : height( i) { } }
bool best( Dog x,Dog y)
    { return x. height > y. height; }
class Cat{ public: string name;
    Cat( string str) : name( str) { }
bool best( Cat x,Cat y)
{ if ( x. name. compare( y. name) > 0)
    return true; else
    return false; } .
```

在所有支持泛型特性的程序设计语言中,C + + 模板对泛型约束的支持是最弱的. 如上述实例的文档约束,就要求 Dog 和 Cat 的实例化类型 x、y 必须实现返回类型为 boolean 类型的 best 函数,否则在实例化时报类型匹配错误. C + + 以文档的形式表达模板的约束,在语法中完全没有体现,全靠程序员的经验从文档中来判断对模板参数的约束,不容易理解,容易引起错误[22].

3.4 Concept 泛型特性

(i) 定义 BigthanComparable 的泛型约束,并定义 best 比较方法,其泛型参数类型为 T.

```
Autoconcept BigthanComparable < typename T >
{ bool best( T,T) ; }
```

(ii) 通过关键字 template 和 where 实现泛型约束的调用,并定义类型参数 T 必须满足约束条件的 sort 排序方法.

```
template < typename T > where BigthanComparable < T >
T*  sort( T a [],int length)
{ for ( int i = 0; i < length - 1; i + + ) { s = i;
    for ( int j = i + 1; j < length; j + + )
    { if( ! best( a [j],a [i]) ) s = j; }
      if ( s! = i)
        { temp = a [i]; a [i] = a [s];
        a [s] = temp; } }
    return a; } .
```

(iii) Dog 类实现实现泛型约束 BigthanComparable.

```
class Dog{ public: int height;
    Dog( int i) : height( i) { } }
bool best( Dog x,Dog y) {
    return x. height > y. height; } .
```

(iv) Cat 类实现实现泛型约束 BigthanComparable.

```
class Cat{ public: string name;
    Cat( string str) : name( str) { }
bool best( Cat x,Cat y) {
    if ( x. name. compare( y. name) > 0)
      return true; else
      return false; } .
```

实例化过程中将实例参数转换成可比较的 Dog 和 Cat 类型,必须满足约束条件的类型方可通过编译,否则报错. Concept 语言和 C + + 语言相比,虽然显式地增加了泛型约束语句,但由于其抽象程度较低,而且其约束仍然只停留在静态语法层约束上,对动态语义层约束暂时无法描述,故其对泛型参数的约束也是不完整的[23].

4 Apla 语言提出及其泛型特性研究

4.1 抽象泛型程序设计语言 Apla

Apla 语言是本研究团队开发的一种基于对象的抽象泛型程序设计语言[24-27],支持完整的静态语法泛型约束和动态语义泛型约束的描述及验证,设计目标为尽可能完整地实现 GP 思想. Apla 是一种高度抽象的程序设计语言,充分地体现了功能抽象和数据抽象等现代程序设计思想,成功地实现了类型参数和操作参数的泛型机制. Apla 通过 constraint 关键字来自定义泛型约束,泛型约束机制通过判定形式参数和类型约束需求之间的关系、实例化参数和操作参数集合之间的关系,对泛型参数的合法性进行检测,从而使得软件的可靠性和安全性得到显著提高. Apla 泛型程序设计语言较其他语言能更好地将泛型约束需求通过显式的语句在程序中表达出来,其约束类型不像其他程序设计语言一样单一,用户可以根据自己的需求不断地更新约束库中的约束类型. Apla 和其他程序设计语言的静态语法约束一样,可以通过 where 关键字来实现泛型约束的调用.

抽象泛型程序设计语言 Apla 的泛型约束描述机制如下:

(i) 约束定义.

define constraint <程序名>;

define ADT <通配符名>(<关键字名> <参数名>);

someop <操作符 1>(参数 1,参数 2…: <参数名>): <参数名>;

someop <操作符 2>(参数 1,参数 2…: <参数名>): <参数名>;

…

enddef;

generic <someADT <通配符名> >;

where{函数(过程)体};

enddef.

(ii) 约束调用.

generic <someADT <通配符名> >

procedure <函数名> <参数 1,参数 2…,泛型参数 1: 通配符名,泛型参数 2: 通配符名…>

where <约束名> <通配符名> <参数 1,参数 2…,泛型参数 1: 通配符名,泛型参数 2: 通配符名…>

{函数(过程)体};

(iii) 约束实例化.

define ADT <类型名>

{函数(过程)体};

implement ADT <类型名>

{函数(过程)体};

procedure <实例函数名>: new <函数名>(instantiation <约束名>(类型名)).

定义 1 泛型约束是在泛型程序设计中对类型参数化的高度概括. 若泛型参数是一个集合,则其泛型约束就是对集合的高度概括; 若泛型参数是函数,其泛型约束就是对函数的高度概括; 若泛型参数是操作,其泛型约束就是对操作的高度概括. 泛型约束就是泛型程序设计中安全性的重要保障.

Apla 语言的泛型约束包含数据类型和操作类型 2 大泛型参数的高度概括[28],数据类型约束进一步分为基本数据类型约束和自定义抽象数据类型(abstract data type,ADT)约束.

定义 2 简单类型(如 integer, boolean, real 等)、枚举类型、自定义简单类型集合归为基本数据类型.

定义 3 自定义抽象数据类型定义了数据的集合以及定义在该数据集上的一组操作,可以对应于一系列元素和在其之上定义的代数操作.

通过实现本文第 3 节中的同一静态语法约束实例,提出自定义抽象数据类型 ADT 并使用 Apla 语言对其静态语法约束进行如下描述:

(i) 约束定义.

```
define constraint Comparable;
generic <sometype T>
    function best(x: T,y: T): boolean;
enddef;
```

(ii) 约束调用.

```
generic sometype <T>
type lst = array [0…100,T]; procedure sort(lst a, int num) where Comparable <T>;
var i,j,k: integer,tt: T;
i: =0;
do i≤num -1→ k: =i; j: =i +1;
    do j≤n→if best(a [j],a [k]) →k: =j; fi;
    j: =j +1; od;
    if i≠k→tt: =a [i];
        a [i]: =a [k];
        a [k]: =tt;
    fi;
    i: =i +1;
od.
```

(iii) 约束实例化.

```
define ADT Dog;
    procedure Dog(i: integer);
    function best(x: Dog,y: Dog): boolean;
enddef;
implement ADT Dog;
    var height: integer;
    procedure Dog(i: integer);
    begin
        height: =i;
    end;
    function best(x: Dog,y: Dog): Boolean;
    begin
        best: =x. height > y. height;
    end;
endimp;
define ADT Cat;
    procedure Cat(str: string);
    function best(x: Cat,y: Cat): Boolean;
enddef;
implement ADT Cat;
    var name: string;
    procedure Cat(str: string);
```

```
        begin
            name: = str;
        end;
        function best( x: Cat, y: Cat) : Boolean;
        begin
            best: = Compare( x. name, y. name) ;
        end;
    endimp;
    procedure Sort_Dog: new sort( instantiation Comparable ( Dog) ) ;
    procedure Sort_Cat: new sort( instantiation Comparable ( Cat) ) .
```

Apla 语言支持静态语法泛型约束,具有完整的约束定义、约束调用和约束实例化. 以上述代码为例: (i) 约束定义,定义一个可比较类型的 Comparable 泛型约束,用户可以根据自己的需求定义多种类型的约束库; 其中 best 为一个泛型参数为 T 的比较函数; (ii) 约束调用,在约束调用过程中,通过调用关键字 where,使得类型参数 T 必须符合泛型约束 Comparable,为可比较类; (iii) 约束实例化,Dog 及 Cat 类型是实例化的自定义抽象数据类型 ADT. 和其他语言的类型相比较,ADT 更加快捷、高效和易于实例化. 用户可以根据自己的需求转换成多种类型,然后将抽象数据类型 Dog 和 Cat 类型或是其他自定义抽象类型 ADT 代入可比较 Comparable 泛型约束中的 best 函数进行判断,是否符合约束定义. 若所代入的类型参数符合可比较类型约束定义,则通过实例化语句调用泛型函数 sort,将输入数据进行排序输出.

基于设计的抽象泛型程序设计语言 Apla 的泛型约束描述机制,本文用 Apla 语言较好地实现了一典型的静态语法约束程序实例. 同时,若在 Apla 泛型约束描述机制中加入基于代数结构的一阶谓词逻辑描述方法,则可以较好地实现动态语义约束. 限于篇幅原因,在此不再赘述,具体请参见文献 [28] .

4.2　Apla 与其他程序设计语言对比研究

泛型程序设计(GP) 与面向对象编程(OOP) 都是针对解决开发过程中的一个重要问题,即如何将系统中不稳定的成分隔离出来,从而用相对稳定的、具有普遍性和共性的成分为基础来构造稳定的可扩展的系统. 与 OOP 不同的是,GP 是根据一个类型所需要的操作需求来进行分类和抽象,即 GP 是一种以类型需求为中心的程序设计范式. 而现在主流的程序设计语言都是 Programming with Generics (or Templates) ,它们是通过某种变相的手段实现了泛型程序设计,但 Generic Programming ≠ Programming with Generics(or Templates) ,不能真正意义上地定义或表示完整的 GP 概念 [15] .

描述完整的 GP 类型需求需要从静态语法约束和动态语义约束出发,大部分主流程序设计语言已经支持静态语法约束,但是受语言其他方面的限制,抽象层次较低,定义的泛型概念难以描述及验证基于动态语义的复杂约束需求问题,与完整实现 GP 尚有距离 [28] .

以 Java、C#、C + + 为例,F. Cardelli 等在 System F 的基础上对其进行扩展,设计了基于子类型约束的扩展 F 受限的参数多态机制,这构成 Java、C# 等面向对象语言多态系统的理论基础. 此机制通过类之间的子类型关系或继承关系,强制所需要的实例类型必须由某个基类(或接口) 派生,支持独立的模块化类型检测. 这种设计方法的优点是易于实现、理解. 但是由于派生类型过于紧耦合的关系,此类约束机制无法支持完整的泛型概念,约束描述的泛型需求过窄,只能称之为窄义的约束.

C + + 模板(包括 STL) 为泛型程序设计提供了良好的基础,然而它无法为泛型约束提供进一步的支持,主要原因体现在如下 2 点:

(i) 泛型概念表达的是作用于类型上的抽象约束,C + + 模板不能较好地支持这一抽象的表达;

(ii) C + + 模板无法对泛型概念所表达的约束进行检查,缺乏用于描述泛型约束的形式语法,是一种欠安全的泛型语言. 其对泛型约束的表达是隐晦的、不容易理解的,不支持模块化的类型检测,由此引发编译信息晦涩、难以对错误进行定位等一系列问题,并导致程序的安全性和可靠性无法保证 [28] .

Apla 泛型程序设计语言是一种高度抽象的语言,不仅支持静态语法泛型约束,还支持动态语义泛型约束,提高了泛型约束的精确度. 泛型参数和泛型约束是松耦合的关系,灵活度高、可操作性强,充分体现了功能抽象、数据抽象等现代程序设计思想,简单易用,便于程序开发. 由于 Apla 语言高度抽象的特点,在描述静态语法约束方面,提出的自定义抽象数据类型 ADT 更加方便高效实用; 在描述动态语义的复杂约束需求问题方面,其基于代数结构及公理语义的泛型约束方法可以完整地描述出其他泛型程序设计语言所不能具体刻画的复杂动态语义泛型约束问题,具有巨大的天然优势,距离真正意义上实现以类型需求为核心的泛型程序设计更近一步!

5 结束语

本文通过同一典型泛型实例来对各种程序设计语言泛型特性进行对比研究,并基于 Apla 语言提出了一种新的泛型机制,通过一个基本数据类型实例来描述泛型约束特性,表现出 Apla 语言,同时兼具静态语法和动态语义泛型约束的能力.

本文较为详细地讨论和分析了 Java、C#、C++、Concept 以及新提出的 Apla 5 种程序设计语言的泛型特性,发现 Java 和 C#都是基于子类型约束的多态机制,这种类型约束描述泛型需求不完整;C++语言采用基于文档的形式表达模板参数的约束,对泛型约束不具体;Concept 语言在 C++模板上显式添加描述泛型约束的语句,其抽象程度较低.因为前 4 种语言其他设施的影响,所以其泛型约束只能称为狭义的约束.相比其他几种面向对象的设计语言,Apla 在这几方面有较大的优势,其泛型需求描述清晰,用户可以根据自己的需求自定义类型的约束,且其抽象程度高,把基于文档形式的隐式约束需求显式化,提高了代码的安全性和可靠性.在描述动态语义泛型约束时,更具优势.Apla 不是 Programming with Generics 而是 Generic Programming.Apla 泛型约束在团队开发的 PAR 平台上可以对静态语法泛型约束进行匹配检测,对动态语义泛型约束进行匹配验证;用 Apla 编写的泛型程序,经过平台系统地约束匹配检测和验证后,可以保证其源程序的可靠性和安全性,经过平台系统转换出的其他可执行程序设计语言的可靠性和安全性也将得到显著地提高.

下一步工作,在泛型约束实例化阶段,基于泛型约束用具体类型将泛型参数实例化,其中要求实例化参数必须满足约束需求,这就需要对泛型参数和泛型约束之间进行匹配检测.

6 参考文献

[1] Garcia R, Jarvi J, Lumsdaine. An extended comparative study of language support for generic programming [J]. Journal of Functional Programming, 2007, 17(2): 145-205.

[2] Jeremy Siek, Jeremiah Willcock. A comparative study of language support for generic programming [J]. Acm Sigplan Conference on Object-oriented Programing, 2003, 38(11): 115-134.

[3] McIlroy M D. Mass-produced software components [C]// Naur P, Randell B. International Conference on Software Engineering, Brussels: NATO, 1968: 1-9.

[4] Zalewski M, Schupp S. A semantic definition of separate type checking in C++ with concepts [J]. Journal of Object Technology, 2009, 8(5): 105-132.

[5] Reynolds J C. Towards a theory of type structure [EB/OL]. [2019-01-12]. http://logcom.oxfordjournals.org/external-ref?access_num=10.1007/3-540-06859-7_148&link_type=DOI.

[6] Austern M H. Generic programming and the STL [M]. Beijing: China Electric Power Press, 2003.

[7] 周卫星,左正康,王昌晶,等. 泛型编程在面向对象语言中的对比研究 [J]. 江西师范大学学报: 自然科学版, 2018, 42(3): 304-310.

[8] 吴国凤,方珏. 基于 Java 语言中的泛型研究 [M]//徐枞巍,韩江. 全国计算机技术与应用. 北京: 电子工业出版社, 2008.

[9] 田芳,石海鹤,左正康. 一种抽象泛型机制的新型 Java 实现 [J]. 江西师范大学学报: 自然科学版, 2016, 40(1): 77-82.

[10] 薛锦云,朱小征,等. Apla→Java 程序生成系统中泛型机制实现方法研究 [J]. 江西师范大学学报: 自然科学版, 2017, 41(1): 52-55.

[11] Hall C, Hammond K, Jones S P. Type classes in Haskell [M]. Berlin: Springer-Verlag, 1994: 241-256.

[12] Piotr Maciol. Application of metaprogramming and generic programming in multiscale modelling [EB/OL]. [2019-01-12]. https://www.computer.org/csdl/magazine/cs/5555/01/08500322/null.

[13] Georg Ofenbeck. Staging for generic programming in space and time [EB/OL]. [2019-01-12]. http://spiral.ece.cmu.edu:8080/pub-spiral/pubfile/paper_284.pdf.

[14] Xue Jinyun. Genericity in PAR platform [EB/OL]. [2019-01-12]. https://link.springer.com/chapter/10.1007%2F978-3-319-31220-0_1.

[15] 孙斌. 面向对象、泛型程序设计与类型约束检查 [J]. 计算机学报, 2004, 27(11): 1492-1504.

[16] 王千文. 泛型程序的形式验证问题的研究 [D]. 宁夏: 宁夏大学, 2015.

[17] 徐文胜. 泛型编程与 Java 实现 [J]. 江西师范大学学报: 自然科学版, 2007, 31(5): 471-474.

[18] Stepanov A, Lee M. The standard template library [J]. C/C++ Users Journal, 1999, 13(12): 10-20.

[19] Musser D R. The tecton concept description language [J]. Technical Communication, 1998, 34(2): 119-120.

[20] 陈林,徐宝文. 基于源代码静态分析的 C++0x 泛型概念抽取 [J]. 计算机学报, 2009, 32(9): 1792-1803.

[21] Reis G D, Stroustrup B. Specifying C++ concepts [M]. New York: ACM Press, 2006: 295-308.

[22] David Pfander, Malte Brunn. AutotuneTMP: auto-tuning in

C + + with runtime template metaprogramming [EB/OL]. [2019-01-12]. https://www.researchgate.net/publication/324867388_AutoTuneTMP_Auto-Tuning_in_C_With_Runtime_Template_Metaprogramming.

[23] Bernardy J P, Jansson P, Zalewski, et al. A conparison of C + + concepts and Haskell type classes [M]. Canada: ACM, 2008.

[24] 石海鹤,石海鹏,薛锦云,等. 一种形式化开发非递归算法的方法 [J]. 计算机应用研究,2007,24(11):203-205.

[25] 薛锦云. 程序设计方法学 [M]. 北京: 国家高等教育出版社,2001.

[26] 游珍,薛锦云,应时. Apla 语言中并发分布式机制的研究 [J]. 计算机科学,2012,39(1):104-108.

[27] 徐华珍,薛锦云,朱小征. Apla→Java 程序生成系统中泛型的机制实现方法研究 [J]. 江西师范大学学报: 自然科学版,2017,41(1):52-55,92.

[28] 左正康. Apla 中泛型约束机制研究 [J]. 软件学报,2015,26(6):1340-1355.

The Comparative Study on the Generic Features of Apla and Programming Languages

ZUO Zhengkang[1], LIU Zhihao[1], HUANG Qing[1], YOU Zhen[2], WANG Changjing[1*], SHI Haihe[1], HU Qimin[2], TAO Xiaoming[1]

(1. College of Computer Information Engineering, Jiangxi Normal University, Nanchang Jiangxi 330022, China;
2. Provincial Key Laborary High Performance Computing, Jiangxi Normal University, Nanchang Jiangxi 330022, China)

Abstract: Object-oriented Programming (OOP) takes objects as the core of programming, while Generic Programming (GP) classifies and abstracts according to the operational requirements required by a type, that is, GP is a programming paradigm with type requirements as the core. Describing the complete type of GP demand needs to include static syntax constraints and dynamic semantic constraints. Most mainstream programming languages have supported the static syntax constraints, however, limited by language's other facilities, the level of abstraction is too low, and the defines of the generic's concepts are hard to describe and verify the problems with complicated constrained demand that based on the dynamic semantics, and complete implementation GP still have a distance. In the paper, the four generic programming languages, such as Java, C#, C + + and Concept are compared. Based on the abstract programming language Apla, a generics mechanism is proposed to further realize GP. By analyzing the generics characteristics of various programming languages with the same typical generic example, and the superiority of the generic mechanism in abstract generic programming language Apla is verified.

Key words: generic programming; dynamic semantic constraints; Apla

(责任编辑: 冉小晓)