软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2015,26(6):1340−1355 [doi: 10.13328/j.cnki.jos.004628]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

Apla 中泛型约束机制研究*

左正康[1,2,3], 薛锦云[1,2]

[1](中国科学院 软件研究所 计算机科学国家重点实验室,北京 100190)

[2](江西省高性能计算技术重点实验室(江西师范大学),江西 南昌 330022)

[3](中国科学院大学,北京 100049)

通讯作者: 薛锦云, E-mail: jinyun@vip.sina.com

摘 要: 泛型程序设计可大幅提高程序的可重用性、可靠性和开发效率.泛型约束机制是对泛型参数进行形式描述,并对其合法性进行检测及验证,从而保证泛型程序的可靠性和安全性.分析总结多种主流语言的泛型约束特性,存在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;以抽象程序设计语言 Apla 为宿主语言,提出了基于代数结构及公理语义的泛型约束方法,给出了基本数据类型、自定义抽象数据类型和子程序的 3 类泛型约束机制,拓展了泛型程序设计约束的应用范围.同时,支持静态语法和动态语义层约束,提高了泛型约束的精确度;借助 Isabelle 定理证明器,设计了泛型约束匹配检测和验证算法;进一步设计了泛型约束机制在 PAR 平台的实现方案及其系统原型.实验部分给出了该泛型约束机制描述、检测及验证一系列复杂泛型约束问题的全过程,自动生成的 C++模板程序的可靠性和安全性得到显著提高.

关键词: 泛型约束机制;Apla 语言;代数结构;动态语义约束;安全性

中图法分类号: TP311

中文引用格式: 左正康,薛锦云.Apla 中泛型约束机制研究.软件学报,2015,26(6):1340−1355. http://www.jos.org.cn/1000-9825/4628.htm

英文引用格式: Zuo ZK, Xue JY. Research on generic constraints of Apla. Ruan Jian Xue Bao/Journal of Software, 2015,26(6): 1340−1355 (in Chinese). http://www.jos.org.cn/1000-9825/4628.htm

Research on Generic Constraints of Apla

ZUO Zheng-Kang[1,2,3], XUE Jin-Yun[1,2]

[1](State Key Laboratory of Computer Science, Institute of Software, The Chinese Academy of Sciences, Beijing 100190, China)

[2](Provincial Key Laboratory of High Performance Computing Technology (Jiangxi Normal University), Nanchang 330022, China)

[3](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: Generic programming has emerged as a paradigm for the development of highly reusable and safe software libraries. Generic constraints mechanism includes a collection of features for constraining generic parameters and verification of the validity of generic parameter instantiated, thereby guarantees dependability and safety of generic programs. This paper first reviews the current research status of generic constraints, exposing the difficulty of describing and verifying generic programs with dynamic semantic constraints. Based on a new description of generic constraints of Apla language, it then proposes three main types of generic constraints mechanism: constraints of basic data types, constraints of custom abstract data types and constraints of subroutines. Next, with the help of Isabelle theorem prover, the paper designs the generic constraints matching detection and validation algorithms and further gives the implementation scheme of generic constraints mechanism in PAR platform. It confirms that the proposed generic constraints mechanism can solve a series of complex generic constraints problems, and so markedly improves dependability and safety of generic programs.

* 基金项目: 国家自然科学基金(61462039, 61020106009, 61363012, 61363013); 江西省自然科学基金(20142BAB217023, 20142BAB217026, 20142BAB207026); 江西省教育厅科技项目(GJJ14268)

收稿时间: 2013-04-26; 定稿时间: 2014-03-27

Key words: generic constraints mechanism; Apla language; algebraic structure; dynamic semantic constraint; safety

1968 年,McIlroy 发表其著名论文《Mass Produced Software Components》[1],提出可重用软件部件的概念,即,像组装硬件部件生产计算机一样组装软件部件生产软件.该文首次提出可重用软件部件要力求泛型(generality),以应对软件部件规模庞大和正确性验证等问题.20 世纪 70 年代,多态类型系统得以实现,代表性工作包括 Girard[2]和 Reynolds[3]提出的 System F,以及 Milner[4]提出的 Hindley-Milner type system 等.20 世纪 80 年代, Musser 和 Stepanov 正式提出了泛型程序设计(generic programming,简称 GP)的概念和基本原则[5],在此阶段出现了支持参数化类型机制的泛型程序设计语言,例如 Ada 的类属、C++的模板等.20 世纪 90 年代,Stepanov 和 Lee 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:标准模板库(standard template library,简称 STL)[6],它是 C++标准程序库的重要组成部分.STL 是迄今为止最为成功的 GP 范例,获得了普遍关注和认同.泛型程序设计方法以抽象和效率为开发目标,从此在业界得到越来越广泛的应用.

C++模板(包括 STL)为复用提供了良好的基础,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两点[7]:

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

(2) 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.

本文以开发安全可靠的 C++模板程序(如 C++可重用部件库等)为目标,以抽象程序设计语言 Apla[8]为宿主语言,设计泛型约束机制和约束匹配检测和验证算法,同时支持静态语法和动态语义层约束,支持完善的模块化约束匹配自动检测及验证,拓展了泛型程序设计约束的应用范围;并进一步设计了泛型约束机制在 PAR 平台[8]的实现方案及其系统原型.实际使用效果表明:该泛型约束机制可解决一系列复杂泛型约束问题,用 Apla 编写的泛型程序,经过完善的约束匹配检测和验证,保证了其可靠性和安全性,因而经过系统自动转换生成的 C++模板程序的可靠性和安全性得到显著提高,对完整实现 GP 具有实际的推动作用.

本文第 1 节进行相关工作的比较.第 2 节提出泛型约束机制在 Apla 中的设计方案.第 3 节设计泛型约束匹配检测及验证算法,并通过一个实例展示具体设计和验证过程.第 4 节为泛型约束机制的系统实现及其分析.第 5 节分析本项研究的适用范围.第 6 节对全文进行总结.

1 相关工作

在程序设计语言领域,以公理语义和代数语义来对多态/泛型类型系统进行规约和验证有很长的历史,可溯及 20 世纪 70 年代的 System F 和 Hindley-Milner type system. System F 引入了一种多态性的λ演算,与简单的类型λ演算不同,它对类型引入了全称量词的机制,从而在程序设计语言里面形式化描述了参数化多态的概念,并构成 Haskell 和 ML 等语言参数化多态的理论基础.Hindley-Milner type system 是一种经典的参数化多态λ演算的类型系统,其最重要的特性是:其在无需类型签名或者程序员的其他提示情况下,有能力在给定的一段程序中推导出最泛的类型,并且已证明它是完备的.设计之初,Hindley-Milner type system 是 ML 语言类型系统的一部分,之后,Haskell 对其进行了扩充,形成了 type classes[9]语言机制.System F 支持更多的类型,Hindley-Milner type system 可看作是一种受限的 System F.

Tecton[10]是一类用泛型概念描述和进行形式化验证工作的语言,采用形式化方法描述泛型概念,强调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.对于许多应用而言,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.

C++语言的模板机制设计的 STL[6]是迄今为止最为成功的 GP 范例,模板机制也是广受关注的泛型机制.C++模板的定义与使用之间未脱离关联,所有的类型检测在泛型实例化之后进行[11],运行效率高、操作灵活,但缺乏对泛型约束的支持.它采用基于文档的形式表达模板参数的约束,对泛型约束的表达是隐晦的、不容易

© 中国科学院软件研究所 http://www.jos.org.cn

理解的,并且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.

Cardelli 等人在 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[12],这构成 Java,C#,Eiffel 等面向对象语言多态系统的理论基础[13].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[14].

ConceptC++是基于 concepts 概念的 C++模板扩展语言,旨在尽可能保持 C++模板高效运行的前提下,解决 C++模板存在的类型检测问题.Concepts 概念约束借助了更加简单和抽象的原操作语句,在 C++模板的基础上显式添加了描述泛型约束的形式语法.Concepts 对泛型约束的支持有明显提升,泛型需求更加明确.ConceptC++的研究有主要有两组团队[15]:Texas 团队借鉴了 Haskell 的 type classes,其设计的 concepts 支持多参数多态,但不支持关联类型;Indiana 团队则认为,关联类型是一个有效的泛型语言机制,可以减少泛型算法类型参数的数量.

Haskell 语言设计的 type classes 包含了一组类型所允许的操作[16],这一点与面向对象语言中的接口相似,但 type classes 本身不是类型.Type classes 设计之初的目的是为了支持重载功能[17],type classes 可以很好地支持泛型概念,提供方便的隐式实例化机制.其最大的特色是支持 Hindley-Milner 风格的类型推理[18],可在一个泛型函数中自动推理出它的约束需求.Haskell 在同一模块中的两个 type classes 中不能有相同的操作名,操作名的作用域均为全局的,给操作名的命名带来了麻烦.

Scala 是一种结合面向对象风范和函数式风范的多范型编程语言[19],它不仅拥有函数式语言所必要的特性(参数化多态、高阶函数等),还提供了面向对象语言最有用的特性(比如重载、继承、子类等等)[19],其 implicits 机制是 GP 领域近期研究的热点[9,19,20].Implicits 基于面向对象程序设计风格,借鉴了部分 type classes 机制来实现参数化多态,且支持关联类型,对泛型概念特性有着良好的支持[9].Implicits 最为出色的特性是支持任何类型实例化泛型约束,可以方便地使用标准面向对象的接口/类(例如常规类型)去实例化泛型概念[20].由于受到面向对象设计理念的限制,Scala 的 implicits 的类型推理机制、语法清晰度等方面不如 Haskell 的 type classes.

ML 的 signatures 属于结构化约束签名的范畴.ML 的 signatures 的参数化粒度比 concepts 和 type classes 更粗.在 ML 中,signatures 被设计用来约束函子(functors)而不是泛型函数,因此,每一个泛型函数不得不嵌入到函子中,并且函子必须在类型参数中显式实例化[18].

Siek 提出了一种原型语言 δ[21],其类型系统基于 System F,是一种命令式语言.δ语言将 type classes 和 concepts 的优点结合,以最大程度满足泛型概念要求,语法与 C++非常相似,既可直接编译运行,又可实现了从δ到 C++的转换工具.

David 和 Haveraaen[22]设计了程序转换技术,可以将 C++ concepts 转换到受约束的 C++模板程序[23].转换过程中不考虑 C++ concepts 的上下文含义,也不对 concepts 进行类型检测,而是设计转换规则将每一条 concepts 定义转换成一系列 C++模板定义,类型检测最终由 C++模板程序来完成.

Gibbons 等人倡导的数据类型泛型程序设计(datatype-generic programming,简称 DGP)[24]通过参数化构造类型的形式(例如 lists,trees 等)来编写程序,这与前面讨论的参数化多态泛型机制有所不同.例如,在“lists of integers”类型中,参数化多态泛型是抽象参数“integers”,而 DGP 是抽象参数 lists of.函数式语言 Haskell 既支持用参数化多态泛型机制(使用 type classes)实现泛型,也对 DGP 有良好的支持.

北京大学孙斌提出了命名类型约束机制[14],其选取 C++语言作为宿主语言,约束机制的语法设计保持与 C++语言风格及其设计思想一致;设计了标准约束库,收录了计算机科学中常用的一组基本概念(包括离散数学、常用数据结构和算法中涉及到的一批概念);开发出一个编译器前端(集成到已有的 C++前端中),对涉及到类型约束代码的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.

本文设计的 Apla 泛型约束语言机制与 Tecton 语言相似,强调在高抽象层次上讨论泛型概念,可形式化地验证某个类型是否符合基于动态语义的泛型约束.本质区别是:本文以开发安全的 C++模板程序(如可重用部件库

© 中国科学院软件研究所 http://www.jos.org.cn

等)为目标,具有工具的支持,设计了泛型约束机制在 PAR 平台的实现方案及其系统原型,支持完善的模块化约束匹配自动检测,可自动转换生成安全可靠的 C++模板程序,以方便地用于软件开发.另外,除 Tecton 语言外,上述的其他语言均为可执行级语言,在语言层面上直接支持泛型概念.显然,受语言其他设施的限制,抽象层次较低,所定义的泛型概念难以支持动态语义约束描述[18,25],难以用于软件需求、分析和测试阶段,对于动态语义约束部分也难以在编译时进行检验[14].

2 泛型约束机制在 Apla 的设计

薛锦云等人一直致力于研究泛型程序设计[26],经过深入的研究,我们给出了泛型约束的定义:

定义 1(**泛型约束**). 泛型约束是在泛型程序设计中对每类泛型参数构成域的精确描述.

例如:若泛型参数为数据,其泛型约束即为对数据域的精确描述,即类型,这是最低级的泛型约束;若泛型参数为数据类型,其泛型约束即为对数据类型的精确描述;若泛型参数为子程序,其泛型约束即为对子程序的精确描述.泛型约束是保证泛型程序设计安全性的重要机制,也是构造可信软件的关键技术.

本节给出的泛型约束包括数据类型和子程序两大类泛型参数的构成域的精确描述.数据类型约束进一步细分为基本数据类型约束和自定义抽象数据类型(abstract data type,简称 ADT)约束.

2.1 基本数据类型约束

基本数据类型定义了一组数据的集合,一些简单类型如 integer,boolean,real,char 等、枚举类型、自定义简单类型集合归类为基本数据类型.因而,在基本数据类型约束的定义中不涉及抽象数据类型的定义.针对此类约束,我们使用一阶谓词逻辑公式的形式来刻画.

例 1:Comparable 可比较类型约束.

```
define constraint Comparable;
    generic ⟨sometype elem⟩;
    someop⊕(x,y:elem):boolean;
        (x,y: x̄, ȳ =elem:∃(⊕∈{>,<,=,≠,≥,≤}));
enddef;
```

例 2:EqualityComparable 等价比较类型约束.

```
define constraint EqualityComparable;
    generic ⟨sometype elem⟩;
    where (Comparable⟨elem⟩);
    someop⊕(x,y:elem):boolean;
        (x,y: x̄, ȳ =elem:¬(∃(⊕∈{>,<,≥,≤})));
enddef;
```

Comparable 约束刻画了一组可比较类型,为了有效描述可比较类型约束的需求,需要额外定义一些与约束相关的操作.someop⊕(x,y:elem):boolean 就是可比较类型约束的关联操作部分.EqualityComparable 约束刻画了一组等价比较类型,这组类型只允许等价比较操作,而不允许其他比较操作.where (Comparable⟨elem⟩)是此约束的约束精化关系部分,指明等价比较类型是可比较类型的一个精化.约束体部分$\neg(\exists(\oplus\in\{>,<,\geq,\leq\}))$明确约定等价比较类型不允许$\{>,<,\geq,\leq\}$比较操作.

2.2 自定义抽象数据类型约束

自定义抽象数据类型定义了数据的集合以及定义在这个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.在代数学中,用公理系统来研究代数系统,不必涉及操作(运算)的对象;用代数方法描述抽象数据类型也无需涉及数据类型的具体表示.一个具体的抽象数据类型可以视为一个具体的代数系统.

代数结构是抽象的代数系统,诸如群、环、域、格等,可以刻画一类代数系统的共性.利用或构造适当的代

数结构把一类问题统一于一个数学模式之中并构成可重用程序部件库,是研究泛型程序设计的一条有效途径.本文基于代数结构来描述自定义抽象数据类型约束,使用代数结构规范语言来描述一类抽象数据类型.

代数结构规范语言用等式代数公理描述操作的语义,其描述的规范说明由一系列的 Theory 组成,每个 Theory 又由语法定义、语义定义和导入 Theory 这 3 部分组成.

定义 2(Theory).

```
Theory theory 名;
    sorts:类型名表;
    opers:操作名:类型名表→类型名;
        …
    where:导入 theory 名表
    eqns:for  量词  变量说明
        等式左部=等式右部
        …
EndTheory;
```

在语法定义部分(sorts,opers),操作名后的类型名表给出了该操作定义域的类型名,其后的类型名为该操作的结果类型名.语义定义部分(eqns)采用代数等式刻画操作的语义,等式右部还允许条件表达式和递归.

例 3:半环的代数系统规范描述.

```
Theory Semi-ring;
    sorts: item;
    opers: ⊗: item×item→item;
           ⊕: item×item→item;
    where: Abelian-monoid(item,⊗)∧Monoid(item,⊗);
    eqns: for (∀x,y,z:item:x⊗(y⊕z)=(x⊗y)⊕(x⊗z)∧(y⊕z)⊗x=(y⊗x)⊕(z⊗x));
EndTheory;
```

例 3 中,类型 *item* 和操作(⊗,⊕)符合半环 Theory 的充分必要条件是:类型 *item* 和操作⊕必须符合阿贝尔独异点 Theory,类型 *item* 和操作⊗必须符合独异点 Theory,并且操作语义必须满足:

$$\text{for } (\forall x,y,z{:}item{:}x\otimes(y\oplus z)=(x\otimes y)\oplus(x\otimes z)\wedge(y\oplus z)\otimes x=(y\otimes x)\oplus(z\otimes x)).$$

基于上例半环的代数系统规范描述,结合 Apla 语言泛型机制,我们给出 Apla 半环泛型约束定义.

例 4:Apla 半环泛型约束定义.

```
define constraint Semi-ring;
define ADT T(sometype elem);
    someop⊕(a,b:elem):elem;
    someop⊗(a,b:elem):elem;
enddef;
generic 〈someADT T〉;
where (Abelian-monoid(T(elem,⊕))∧Monoid(T(elem,⊗)));
    (x,y,z: x̄, ȳ, z̄ =elem:x⊗(y⊕z)=(x⊗y)⊕(x⊗z)∧(y⊕z)⊗x=(y⊗x)⊕(z⊗x));
enddef;
```

根据上述语言设施,我们还设计了一个基础性的 Apla 泛型约束库,并严格按照泛型程序设计的方法和新的语法规则来开发,其中主要收录计算机科学中常用的一组基本概念(包括离散数学、常用数据结构和算法中涉及到的一批概念)的泛型需求,例如 Comparable、EqualityComparable、Basetype、Basebinaryop、幂等结构、交换结构、可逆结构、广群、半群、独异点、阿贝尔独异点、群、阿贝尔群、环和半环等一系列约束.主要设计

© 中国科学院软件研究所 http://www.jos.org.cn

任务包括对这些类型约束进行分类、类型需求的详细/正确的描述、精化关系的确定.用户可以基于此泛型约束库,依据自身需求自定义更多约束.通过对这一系列类型需求的定义,我们对类型约束机制进行了较为全面的测试运用.结果表明,其描述能力和适用性已达到了设计目标.

2.3 子程序约束

在 Apla 中,子程序包括函数和过程两部分.泛型程序允许将子程序作为参数,这样可以使得子程序泛化,具体子程序可以对子程序参数进行实例化.子程序约束是对子程序参数构成域的精确描述,对于子程序参数而言,究竟哪些具体子程序可以实例化它,必须给出精确的描述.若考虑子程序参数功能性部分,本文采用基于 Hoare 公理语义的操作规约来描述子程序约束.操作规约由前置断言和后置断言组成,前置断言表示子程序执行之前应该满足的条件,而后置断言表示子程序终止时应该满足的条件.为了将操作规约表达清楚,必须使用一些符号、联结词和其他表示成份,这些成份共同构成操作规约描述语言,形式化地描述子程序所实现的功能.设 AQ 表示子程序的前置断言,AR 表示子程序的后置断言,子程序约束就是由前后置断言$\{AQ\}$和$\{AR\}$来描述的,即:对于任意实例子程序 S,若能判定$\{AQ\}S\{AR\}$成立,则 S 满足子程序约束.子程序约束充分体现 GP 定义及其设计思想,是本文设计的一种新的约束形式.

操作规约是判定子程序约束匹配的基础,子程序约束是否精确反映问题及用户需求,直接影响最终约束匹配的正确性验证.在书写操作规约,有两点说明:

(1) 必须指明操作规约中哪些标识符的值是可变的,哪些是不可变的:子程序参数中有些标识符,它们的值在子程序执行之前从外部获得,且在子程序执行过程中始终保持不变,即,这些标识符的值不随子程序状态的改变而变化.我们称这类标识符为输入参数,用 in 表示;子程序中还有另一类标识符,其值随程序的执行而不断变化,称为输出变量,用 out 表示;

(2) 引入辅助参量.为了描述子程序变量取值的变化规律,在书写操作规约时,有时还需要引入一些辅助参量,说明子程序执行前、后变量值之间的关系.辅助参量只能用于操作规约的描述,不得出现在子程序中,使用 aux 说明.

例 5:排序类子程序[27]约束定义.

define constraint Sort;

generic ⟨someop⊗(value result a:array[0…n−1,integer])⟩;

|[in n: integer; out a[0:n−1]:array of integer]|

AQ: $n \geqslant 0$;

AR: sort(a,0,n−1)≡($\forall w$:1≤w<n:$a[w] \leqslant a[w+1]$)

enddef;

3 约束匹配检测和验证算法

在约束例化阶段,基于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,约束匹配就是对这一过程进行匹配检查.本文将约束匹配分为两部分:约束匹配检测和约束匹配验证.其中:约束匹配检测可判定形式参数和实例化参数是否满足约束的静态语法需求,此过程是基于 PAR 平台完全自动完成;而约束匹配验证则是判定实例化参数是否满足约束的动态语义需求,此过程为部分自动化,需要手工推演出可验证的谓词逻辑公式,并验证其正确,部分逻辑公式借助 Isabelle 定理证明器进行自动验证[28].

3.1 约束匹配检测

定义 3(泛型过程定义).

⟨泛型过程定义⟩::=⟨泛型过程头⟩⟨过程体⟩

⟨泛型过程头⟩::=⟨泛型参数表⟩⟨过程头⟩[⟨约束调用⟩]

⟨泛型参数表⟩::=generic ⟨someADT ⟨类型参数⟩{;someADT ⟨类型参数⟩}⟩

© 中国科学院软件研究所 http://www.jos.org.cn

〈过程头〉::=procedure 〈泛型过程名〉;|procedure 〈泛型过程名〉(〈形参部分〉{;〈形参部分〉});
〈形参部分〉::=result 〈参数组〉|value 〈参数组〉
〈参数组〉::=〈标识符〉{,〈标识符〉}:〈类型〉
〈过程体〉::=[〈常量说明〉][〈类型说明〉][〈变量说明〉] begin 〈复合语句〉 end;
〈泛型过程名〉::=〈标识符〉
〈复合语句〉::=〈语句〉{〈语句〉}

定义 4(泛型函数定义). 与泛型过程定义类似,可参照泛型过程定义,限于篇幅,略.

泛型过程或函数的过程体或函数体的语句是由各类表达式构成,在这些表达式中,我们把它们分为依赖性表达式和非依赖性表达式,其中,依赖性表达式是包含泛型参数的表达式,非依赖性表达式则与泛型参数无关.在形式参数检测中,我们只需检测依赖性表达式是否符合约束需求即可.约束匹配检测可以在 PAR 平台中完全自动完成.

形式参数检测算法(如图 1 所示).

Step 1. 根据定义 3 和定义 4,泛型函数(过程)可分为 4 部分:泛型参数表、函数头(过程头)、约束调用、函数体(过程体).
Step 2. 扫描泛型参数表部分,得到类型参数 *T*.
Step 3. 扫描约束调用部分,得到此泛型函数(过程)所调用的约束名 *N*.
Step 4. 根据约束名 *N*,扫描 *N* 的约束定义.
Step 5. 依据约束定义规则,进一步生成类型参数 *T* 的操作集合 *S*.
Step 6. 扫描函数体(过程体)部分,提取与形式类型参数 *T* 相关的依赖性表达式 *E*.
Step 7. 扫描所有依赖性表达式 *E*,自动生成 *E* 的所有操作集 *P*.
Step 8. 判定操作集 *P* 是否属于操作集合 *S* 的子集,若是则检测通过;若不是则调用出错处理.

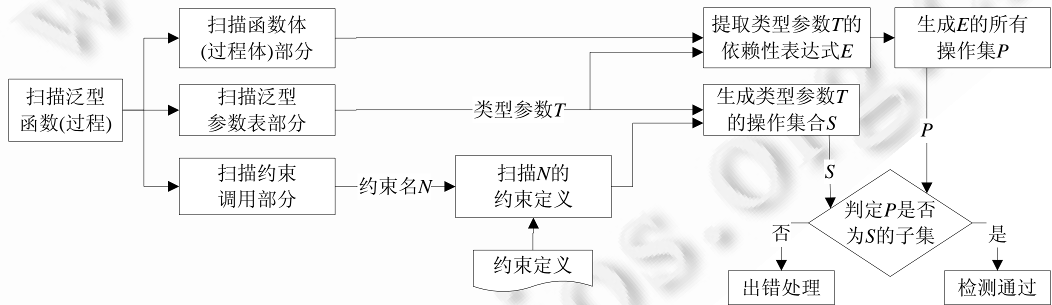

Fig.1 Formal parameter constraints matching detection
图 1 形式参数约束匹配检测

实例化参数检测算法(如图 2 所示).

Step 1. 扫描约束例化部分,得到实例化所对应的约束名 *N*、实例化类型参数 *T* 和实例化操作参数 *P*.
Step 2. 根据约束名 *N*,扫描 *N* 的约束定义.
Step 3. 依据约束定义规则,自动生成抽象数据类型数据域的类型集合 *X* 和操作域的操作集合 *Z*.
Step 4. 判定实例化类型参数 *T* 是否属于集合 *X*.
Step 5. 判定实例化操作参数 *P* 是否属于集合 *Z*.
Step 6. 判定 Step 4 和 Step 5 是否均为真:若是,则检测通过;若不是,则调用出错处理.

© 中国科学院软件研究所 http://www.jos.org.cn

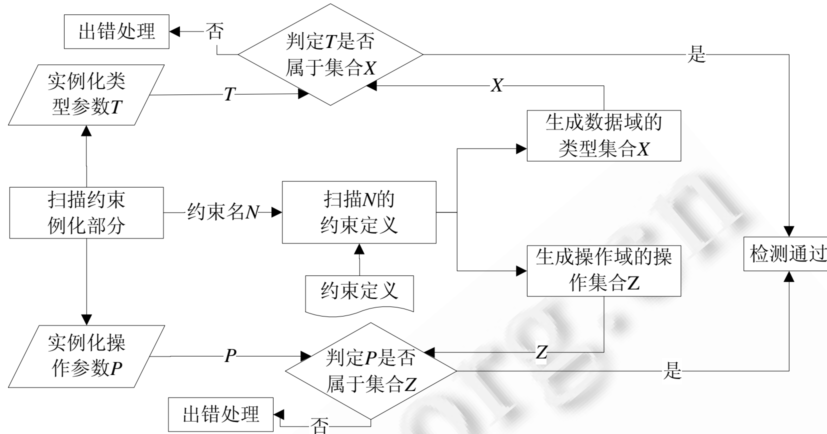

Fig.2 Instantiation parameter constraints matching detection

图 2 实例化参数约束匹配检测

3.2 约束匹配验证

抽象数据类型定义了数据的集合以及定义在这个数据集上的一组操作,可以看作对应于一系列元素和在它们之上定义的代数操作.对于自定义抽象数据类型的约束,我们有如下定义和结论:

定义 5. 设$\langle A,f_1,\ldots,f_m,a_1,\ldots,a_k\rangle$,$\langle B,g_1,\ldots,g_m,b_1,\ldots,b_k\rangle$是两个同类的代数系统,$\sigma$是 A 到 B 的映射.如果:

(1) 任意 $1\leqslant i\leqslant m,(\forall x_1,\ldots,x_n\in B)(\sigma(f_i(x_1,\ldots,x_n))=g_i(\sigma(x_1),\ldots,\sigma(x_n)))$;

(2) 任意 $1\leqslant j\leqslant k,\sigma(a_j)=b_j$,

则称σ是$\langle A,f_1,\ldots,f_m,a_1,\ldots,a_k\rangle$到$\langle B,g_1,\ldots,g_m,b_1,\ldots,b_k\rangle$的同态映射,称$\langle B,g_1,\ldots,g_m,b_1,\ldots,b_k\rangle$同态于$\langle A,f_1,\ldots,f_m,a_1,\ldots, a_k\rangle$,记作 $A\approx B$.把$\langle \sigma(A),g_1,\ldots,g_m,b_1,\ldots,b_k\rangle$称为$\langle A,f_1,\ldots,f_m,a_1,\ldots,a_k\rangle$的一个同态象.

第 2.2 节提出了使用代数结构规范刻画了一类代数系统的基本属性,而一个代数系统可以为一具体的抽象数据类型系统,它由对象集和对象间的函数映射构成.一个代数系统满足代数结构规范说明,是指对代数结构规范说明中的任意等式,无论其中的变量取域中何值,该等式在此代数系统中均成立,此时称这个代数系统为该代数结构规范说明的一个模型(model).往往有许多不同的模型满足同一规范说明,这些不同模型间存在着同态映射关系.因此,若已知一个模型 A 满足代数结构规范说明,则模型 A 的同态象均可以保证满足其代数结构规范说明,或者存在另一模型 B 与模型 A 有同态映射关系,则模型 B 也满足代数结构规范说明.

基于上述理论,我们得出约束匹配验证的算法.

Step 1. 用实例类型替换类型约束定义中的泛型参数表,并据此进一步生成实例约束精化和实例约束体.

Step 2. 基于类型约束定义中的约束精化关系,将实例约束精化进一步展开成新的实例约束精化和实例约束体.

Step 3. 判断是否还存在未展开的实例约束精化:

Step 3.1. 若存在,则继续展开,直到不能展开为止,即,展开的实例约束精化均为实例约束体;

Step 3.2. 若已不存在未展开的实例约束精化和实例关联约束,则执行 Step 4.

Step 4. 所有展开的实例约束体即为谓词逻辑公式,利用谓词逻辑变换规则可以对已得到谓词逻辑公式进行化简和分化,得到 Isar 证明脚本.

Step 5. 用 Isabelle 定理证明器判定 Isar 证明脚本是否可验证[28].

Step 6. 若验证合法,则检测通过;否则,提示错误信息.

Step 7. 若有两组实例类型 A 和 B,且已证明实例类型 A 已匹配约束,将 A 和 B 转换为代数系统 A'和 B',若能证明 A'和 B'之间存在同态映射关系,则可得到实例类型 B 匹配约束的结论.

3.3 泛型Kleene算法的设计及其闭半环约束验证

3.3.1 泛型 Kleene 算法实例

本节选取泛型Kleene算法[29]对本文提出的Apla泛型约束设计进行解释说明.Kleene泛型算法是一个$O(n^3)$的算法,解决了通用的路径问题,该泛型算法以代数结构闭半环[30]作为其泛型约束.泛型Kleene算法的设计分为约束定义、约束调用和约束例化这 3 个步骤.

(1) 约束定义

给出闭半环代数结构的 Apla 约束定义:

```
define constraint Closed-semiring;
define ADT T(sometype elem);
      someop⊕(a,b:elem):elem; someop⊗(a,b:elem):elem;
enddef;
generic ⟨someADT T⟩;
where (Basetype(T.elem)∧Basebinaryop(T.⊕)∧Basebinaryop(T.⊗)∧Semi-ring(T(elem,⊕,⊗)));
```

$$\left(x,y:\overline{x},\overline{y}=integer;\overline{a[x]},\overline{a[y]}=elem:\left(\bigoplus_{x=0}^{\infty}\right)a[x]\otimes\left(\bigoplus_{y=0}^{\infty}\right)b[y]=\left(\bigoplus_{x,y=0}^{\infty}\right)(a[x]\otimes b[y])\right);$$

```
enddef;
```

其中,Basetype,Basebinaryop 和 Semi-ring 都归属于 Apla 泛型约束库,Basetype 定义了基本类型约束,Basebinaryop 定义了基本二元操作类型约束,Semi-ring 定义了半环约束,本例在自定义约束 Closed-semiring 中直接调用了它们.

(2) 约束调用

```
generic ⟨someADT T⟩;
procedure Kleene (n:integer;c:array[1…num,array[1…num,elem]]);
where (Closed-semiring(T(elem,⊕,⊗)));                    //调用闭半环约束
var
      i,j,k:integer;
begin
k:=1; do (k≤n)→foreach(i,j:1≤i,j≤n:c[i,j]:=c[i,j]⊕(c[i,k]⊗c[k,j]);); k:=k+1; od;
end;
```

Kleene泛型过程的泛型参数为一个自定义抽象数据类型参数T,其中T必须符合Closed_semiring闭半环约束,where (Closed-semiring(T(*elem*,⊕,⊗)))为约束调用语句.

(3) 约束例化

Kleene 泛型算法统一了一系列图的路径算法问题,包括最短路径算法、传递闭包算法和最大容量路算法.选取适当的闭半环结构,通过实例化语句替换泛型过程中的自定义抽象数据类型参数 T,就可以生成解决不同问题的具体算法.例如,选取闭半环($I^+\cup\{+\infty\}$,MIN,+,+∞,0),执行实例化语句:

```
ADT A1: new T(integer;min;+);
procedure floyd: new kleene(instantiation Closed_semiring(A1));
```

可以生成计算有向图 G 所有顶点对之间最短路径的子程序.

又如,选取闭半环({0,1},∨;∧,0,1),执行实例化语句:

```
ADT A2: new T(boolean;∨;∧);
procedure close_set: new kleene(instantiation Closed_semiring(A2);
```

可以生成计算有向图 G 传递闭包的子程序.

再如,选取闭半环($I^+\cup\{+\infty\}$,MAX,MIN,0,+∞)执行实例化语句:

© 中国科学院软件研究所 http://www.jos.org.cn

ADT *A*3: new *T*(integer;max;min);

procedure capcity: new kleene(instantiation Closed_semiring(*A*3);

可以生成计算有向图中所有顶点间最大容量的算法.

可以证明:凡是满足闭半环特性的问题,都可以通过对 Kleene 泛型算法的实例化求解,这就使得这一泛型算法成为一类算法的抽象[26].

3.3.2 泛型 Kleene 算法闭半环约束验证

本节给出泛型 Kleene 算法的约束匹配验证过程(约束匹配检测过程见第 4 节),即,验证约束例化符合闭半环约束定义.

(1) 用实例化抽象数据类型(integer;min;+),(boolean;∨;∧)和(integer;max;min)替换闭半环约束定义中的泛型参数表.

(2) 用实例化类型替换后,将闭半环约束展开成约束例化展开式 1~展开式 3,如下所示:

- 约束例化展开式 1:实例约束精化 R1{Basetype(integer),Basebinaryop(min),Basebinaryop(+),Semi-ring(integer;min;+)}和实例约束体 A1:
$$\forall\left(x,y:\overline{x},\overline{y}=integer;\overline{a[x]},\overline{a[y]}=integer:\left(\overset{\infty}{\underset{x=0}{\oplus}}\min\right)a[x]+\left(\overset{\infty}{\underset{y=0}{\oplus}}\min\right)b[y]=\left(\overset{\infty}{\underset{x,y=0}{\oplus}}\min\right)(a[x]+b[y])\right);$$
- 约束例化展开式 2:实例约束精化 R2{Basetype(boolean),Basebinaryop(∨),Basebinaryop(∧),Semi-ring(boolean;∨;∧)}和实例约束体 A2:
$$\forall\left(x,y:\overline{x},\overline{y}=integer;\overline{a[x]},\overline{a[y]}=boolean:\left(\overset{\infty}{\underset{x=0}{\oplus}}\vee\right)a[x]\wedge\left(\overset{\infty}{\underset{y=0}{\oplus}}\vee\right)b[y]=\left(\overset{\infty}{\underset{x,y=0}{\oplus}}\vee\right)(a[x]\wedge b[y])\right);$$
- 约束例化展开式 3:实例约束精化 R3{Basetype(integer),Basebinaryop(max),Basebinaryop(min),Semi-ring(integer;max;min)}和实例约束体 A3:
$$\forall\left(x,y:\overline{x},\overline{y}=integer;\overline{a[x]},\overline{a[y]}=integer:\left(\overset{\infty}{\underset{x=0}{\oplus}}\max\right)a[x]\min\left(\overset{\infty}{\underset{y=0}{\oplus}}\max\right)b[y]=\left(\overset{\infty}{\underset{x,y=0}{\oplus}}\max\right)(a[x]\min b[y])\right).$$

下一步,将实例约束精化 R1,R2 和 R3 分别展开,以展开 R1 为例.R1 展开可得到 4 个实例约束精化:Semi-ring (integer;min;+),Basetype(integer),Basebinaryop(min),Basebinaryop(+)和 1 个实例约束体 A1.将此 4 个实例约束精化继续展开,以 Semi-ring(integer;min;+)的展开过程为例,如图 3 所示,最终得到的所有实例约束体为 B1~B8.

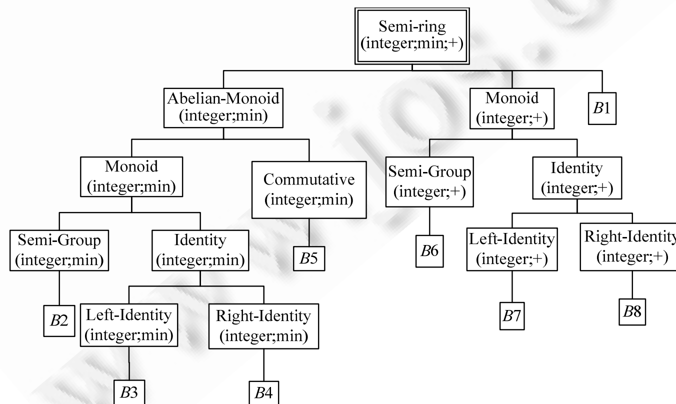

Fig.3 Unfolding tree of Semi-ring(integer;min;+)

图 3 Semi-ring(integer;min;+)的展开树

限于篇幅,我们给出 *B*1 和 *B*2 的谓词逻辑式,*B*3~*B*8 类似,略.

$B1:\forall(x,y,z:\overline{x},\overline{y},\overline{z}=integer:x+(\min(y,z))=\min((x+y),(x+z))\wedge(\min(y,z))+x=\min((y+x),(z+x)));$

© 中国科学院软件研究所 http://www.jos.org.cn

$B2$: $\forall(x,y,z:\overline{x},\overline{y},\overline{z}=integer:\min(\min(x,y),z)=\min(x,\min(y,z)))$.

(3) Basetype(integer),Basebinaryop(min)和 Basebinaryop(+)的展开过程略.

(4) 将未展开的实例约束精化进一步展开成新的实例约束精化和实例约束体,直至所有的实例约束精化均展开成实例约束体的形式.

(5) 将所有实例约束体用 Isar 语言描述,得到 Isar 证明脚本.借助 Isabelle 定理证明器,我们可判定所有 Isar 证明脚本是成立的[28].限于篇幅,证明过程略.

(6) 将抽象数据类型(integer;min;+)转换为与其同构的代数系统 $A\langle I,\min,+\rangle$.通过步骤 5 的验证过程,可得出代数系统 $A\langle I,\min,+\rangle$已满足闭半环约束的结论.

(7) 将抽象数据类型(boolean;∨;∧)转换为与其同构的代数系统 $B\langle\{0,1\};\vee;\wedge\rangle$;将抽象数据类型(integer;max;min)转换为与其同构的代数系统 $C\langle I,\max,\min\rangle$.同理可证,代数系统 $C\langle I,\max,\min\rangle$满足闭半环约束.

定理 1. 设 f 是代数系统$\langle A,+,*\rangle$到代数系统$\langle B,\oplus,\otimes\rangle$的一个同态映射.如果$\langle A,+,*\rangle$是闭半环,且$\langle B,\oplus,\otimes\rangle$是关于同态映射 f 的同态象,那么$\langle B,\oplus,\otimes\rangle$也是一个闭半环.

证明:由于$\langle A,+,*\rangle$是闭半环,$\langle A,+,*\rangle$一定是半环.由于$\langle A,+\rangle$是阿贝尔独异点,$\langle A,*\rangle$是独异点,容易证明:$\langle B,\oplus\rangle$也是阿贝尔独异点,$\langle B,\otimes\rangle$也是独异点.

对于任意的 $b_1,b_2,b_3\in B$,必有相应的 a_1,a_2,a_3,使得 $f(a_i)=b_i(i=1,2,3)$.

于是:

$$\begin{aligned}b_1\otimes(b_2\oplus b_3)&=f(a_1)\otimes(f(a_2)\oplus f(a_3))\\&=f(a_1)\otimes f(a_2+a_3)\\&=f(a_1*(a_2+a_3))\\&=f(a_1*a_2+a_1*a_3)\\&=f(a_1*a_2)\oplus f(a_1*a_3)\\&=(f(a_1)\otimes f(a_2))\oplus(f(a_1)\otimes f(a_3))\\&=(b_1\otimes b_2)\oplus(b_1\otimes b_3).\end{aligned}$$

同理可证:$(b_2\oplus b_3)\otimes b_1=(b_2\otimes b_1)\oplus(b_3\otimes b_1)$.

因此,$\langle B,\oplus,\otimes\rangle$也是一个半环.对任意的 $x_1,x_2,\dots,x_\infty,y_1,y_2,\dots,y_\infty\in B$,必有相应的 $a_1,a_2,\dots,a_\infty,b_1,b_2,\dots,b_\infty\in A$,使得:

- $f(a_i)=x_i$ $(i=1,2,3,\dots,\infty)$;
- $f(b_j)=y_j$ $(j=1,2,3,\dots,\infty)$.

于是:

$$\begin{aligned}\left(\bigoplus_{i=0}^{\infty}\right)x[i]\otimes\left(\bigoplus_{j=0}^{\infty}\right)y[j]&=\left(\bigoplus_{i=0}^{\infty}\right)f(a[i])\otimes\left(\bigoplus_{j=0}^{\infty}\right)f(b[j])\\&=(f(a[0])\oplus f(a[1])\oplus\dots\oplus f(a[\infty]))\otimes(f(b[0])\oplus f(b[1])\oplus\dots\oplus f(b[\infty]))\\&=f(a[0]+a[1]+\dots+a[\infty])\otimes f(b[0]+b[1]+\dots+b[\infty])\\&=f((a[0]+a[1]+\dots+a[\infty])*(b[0]+b[1]+\dots+b[\infty]))\\&=f\left(\sum_{i,j=0}^{\infty}(a[i]*b[j])\right)\\&=\left(\bigoplus_{i,j=0}^{\infty}\right)(x[i]\otimes y[j]).\end{aligned}$$

因此,$\langle B,\oplus,\otimes\rangle$也是一个闭半环. □

(8) 由于代数系统 $B\langle\{0,1\};\vee;\wedge\rangle$与代数系统 $C\langle I,\max,\min\rangle$具有同态映射关系 $f:I\to\{0,1\}$如下:

$$f(n)=\begin{cases}1, & 若n\geqslant 0,n\in I\\0, & 若n<0,n\in I\end{cases}.$$

根据定理 1,可得到代数系统 $B\langle\{0,1\};\vee;\wedge\rangle$也满足闭半环约束.

(9) 综上所述,抽象数据类型(integer;min;+),(boolean;∨;∧),(integer;max;min)均满足闭半环约束. □

© 中国科学院软件研究所 http://www.jos.org.cn

4 泛型约束机制的系统实现

PAR 平台 C++程序生成系统[8]的目标是将 Apla 语言描述的程序自动转换成 C++程序.Apla 到 C++程序生成系统的系统架构如图 4 所示.本研究对原系统进行了扩充,主要实现了两项功能:一是泛型约束匹配检测,二是将检测通过的 Apla 泛型程序自动转换为 C++模板程序.

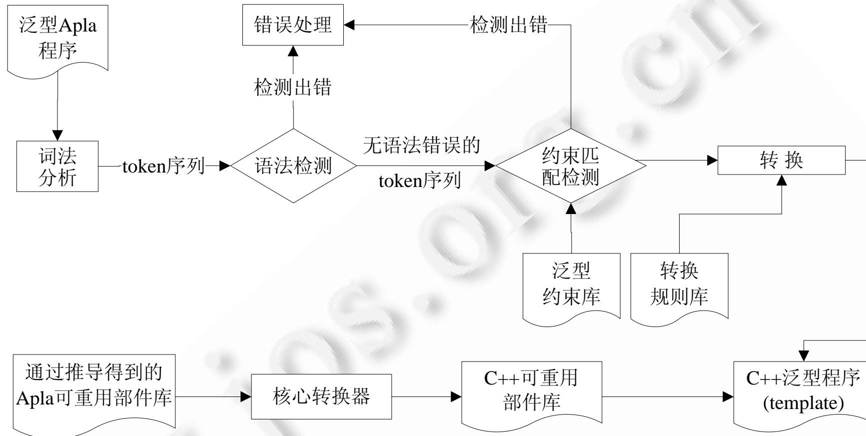

Fig.4 Generation system architecture of Apla to C++ program

图 4 Apla 到 C++程序生成系统的系统架构

约束匹配检测机制可判定形式参数和实例化参数是否满足约束的静态语法需求,此过程是基于本系统完全自动化完成.

依据图 4 所示的系统架构图以及第 3.1 节给出的约束匹配检测算法,下面将展示泛型 Kleene 算法实例的系统实现过程:

Step 1. 使用本文定义的泛型约束机制将 Kleene 泛型算法用 Apla 语言描述,得到一泛型 Apla 程序,如本文第 3.3.1 节所示.

Step 2. 对泛型 Apla 程序进行词法分析,将其转换为一个 token 序列,再采用递归下降法对该 token 序列进行语法检测.

Step 3. 语法检测通过后,调用预定义 Apla 泛型约束库(见第 2.2 节),确定泛型 Apla 程序所调用的约束的精化关系.

Step 4. 依据此精化关系,对无语法错误的 token 序列,启动约束匹配检测:

Step 4.1. 形式参数约束匹配检测:

Step 4.1.1. 依据闭半环约束定义,平台自动生成类型参数 *elem* 的可操作集合为 $S\{\oplus,\otimes\}$;

Step 4.1.2. 扫描 Kleene 泛型过程,生成与形式类型参数 *elem* 相关的依赖性表达式 write ($c[i,j]$, “,”)和 $c[i,j]\oplus(c[i,k]\otimes c[k,j])$;

Step 4.1.3. 生成依赖性表达式中的类型参数相关操作为 $P\{\oplus,\otimes\}$;

Step 4.1.4. $P\subseteq S$ 成立,形式参数约束匹配检测通过;

Step 4.2. 实例化参数约束匹配检测:

Step 4.2.1. 扫描约束例化部分,得到实例化类型 T 为 integer 和 boolean,实例化操作 P 为 MIN, MAX,+,∧,∨;

Step 4.2.2. 依据闭半环约束定义,闭半环约束中的操作参数精化自 Basetype,Basetype 是一类预定义约束,刻画了一组基本类型,因此可自动生成数据域的类型集合 X{integer,real,char, boolean};

© 中国科学院软件研究所 http://www.jos.org.cn

Step 4.2.3. 依据闭半环约束定义,闭半环约束中的操作参数精化自 Basebinaryop,Basebinaryop 是一类预定义约束,刻画了一组基本二元操作,因此可自动生成操作域的操作集合:

Z\{MIN,MAX,+,−,*,/,=,≠,∧,∨,>,<,⩾,⩽,∪,∩,∈,∉,⊂,⊃,⊆,⊇\};

Step 4.2.4. 实例化类型 T 均属于集合 X,实例化操作 P 均属于集合 Z.因此,实例化参数约束匹配检测通过.

Step 5. 通过约束匹配检测的泛型 Apla 程序将依据 Apla 语言到 C++语言的转换规则库将其转换为对应的 C++模板程序,如图 5 所示,界面左侧为泛型 Apla 程序,右侧为自动生成的 Kleene C++模板程序.

Step 6. 对 Kleene C++模板程序编译、执行,可得到程序的执行结果,如图 6 所示.经检验,程序的执行结果与预期相符.

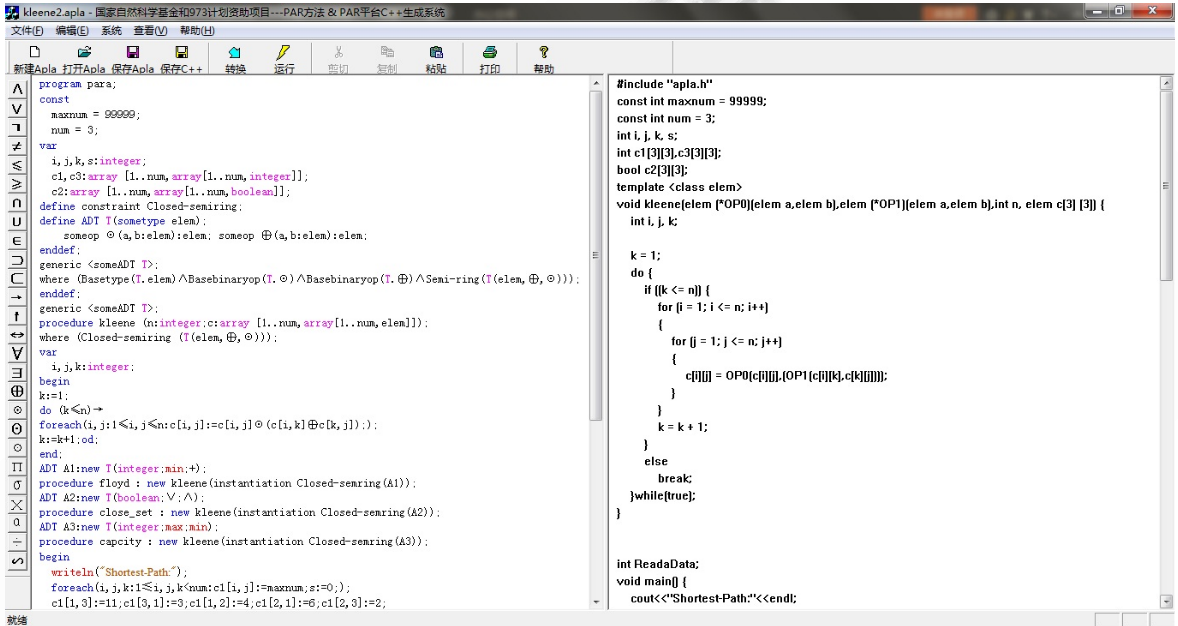

Fig.5 Automatically generating Kleene C++ template program

图 5 自动生成 Kleene C++模板程序

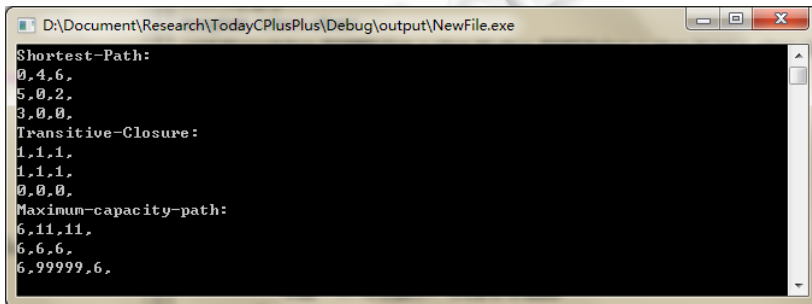

Fig.6 Program execution result

图 6 程序执行结果

泛型 Apla 程序经过约束匹配检测(完全自动化)和约束匹配验证(部分自动化,见第 3.3.2 节),保证了其可靠性和安全性.因而,经过系统自动转换生成的 Kleene C++模板程序的可靠性和安全性得到显著提高.

© 中国科学院软件研究所 http://www.jos.org.cn

5 适用范围

泛型约束 R 包括了两个不同的部分[6]:对满足概念需求的类型进行合法操作的静态语法表示(即,valid expressions)R_{static} 以及类型的动态语义约束(expression semantics)$R_{dynamic}$,即,$R=R_{static}\cup R_{dynamic}$.在第 1 节介绍的语言中,多数语言均为可执行级语言,在语言层面直接支持泛型概念.显然,受语言其他设施的限制,抽象层次较低,对于动态语义约束部分,难以在编译时进行检验[14].Reis,Stroustrup 和 Meredith 在文献[31]中给出了 C++ concepts 语义描述的初步设想,认为用一阶逻辑的形式刻画 concepts 的语义比起用自然语言文本刻画要更加清楚,且无二义性.但由于复杂性、成本等原因,对于编译器(例如 ConceptC++编译器)来说,要描述和检验泛型约束的动态语义已超出其能力范围,建议构造工具对动态语义进行分析,并最终实现程序转换.

本文方法力求解决上述问题,适合解决泛型的动态语义约束分析及验证问题,强调在高抽象层次上描述泛型概念.选取抽象程序设计语言 Apla 为宿主语言,具有功能抽象、数据抽象等程序设计思想,提供了标准数据类型,预定义 ADT 和自定义 ADT 机制,其符号表达方式兼容传统的数学符号和数学表达式,可形式化地验证某个类型是否符合基于动态语义的泛型约束.经过一些算法实例检验发现:若定义在泛型数据集上的多个操作相互具有关联,用本文介绍的方法尤其适用.

现有泛型语言多为可执行级语言,受编译器的限制,语言的抽象层次较低,因此只能描述并检测泛型约束的静态语法需求[18,25].本文提出对静态语法需求实施约束匹配检测、对动态语义需求实施约束匹配验证两类算法,其中,约束匹配检测完全计算机自动化完成(见第 4 节系统实现部分),约束匹配验证需要手工推演出可验证的谓词逻辑公式,并验证其正确.部分逻辑公式借助 Isabelle 定理证明器进行自动验证,验证过程为部分自动化.

基于 Apla 泛型约束设计机制,我们已实现了基于 3 类泛型约束机制的典型算法,在平台的支持下,我们已自动检测和验证了包括:1) 基于基本数据类型约束:泛型求和算法、泛型极值类算法[32];2) 基于自定义抽象数据类型约束:泛型 Bellman-Ford 算法[33]、泛型 Kleene 算法;3) 基于子程序约束:泛型二分搜索算法、泛型中缀表达式求值算法等,并自动转换生成了 C++模板程序.经实际运行检测,程序的运行结果均与预期相符.本文给出了泛型 Kleene 算法实例.限于篇幅,其他算法另文论述.

6 结束语

本文以开发安全可靠的C++模板程序(如C++可重用部件库等)为目标,以抽象程序设计语言 Apla 为宿主语言,提出了基于基本数据类型、自定义抽象数据类型和子程序的 3 类泛型约束机制,设计泛型约束机制和约束匹配检测和验证算法,同时支持静态语法和动态语义层约束,支持完善的模块化约束匹配自动检测及验证;并进一步设计了泛型约束机制在 PAR 平台 C++生成系统的实现方案及其系统原型.相比国内外现有研究,具有以下特点:

(1) 提出了抽象数据类型约束的代数结构描述方法、理论和实现技术,支持动态语义层约束,较现有泛型语言更能精确地描述泛型需求;

(2) 提出约束匹配检测和验证两类算法:对静态语法需求实施约束匹配检测,检测过程完全自动化;对动态语义需求实施约束匹配验证,借助 Isabelle 定理证明器,验证过程部分自动化;

(3) 用 Apla 编写的泛型程序,经过完善的约束匹配检测和验证,保证了其可靠性和安全性.因而,经过系统自动转换生成的 C++模板程序的可靠性和安全性得到显著提高.

可以发现:使用抽象程序语言 Apla 来定义新的约束机制是一个行之有效的途径,并且具备极大的潜力.本文进一步完善了薛锦云等人提出的算法程序形式化开发方法 PAR[8]和为 PAR 定义的抽象程序设计语言 Apla 的理论基础和应用价值.下一步的工作将包括扩充可重用泛型约束库、完善约束设计机制、设计更多实例检验 PAR 平台和本方法的实现能力;借鉴现有研究(尤其是 ConceptC++)在约束类型检测的经验,提升约束匹配检测处理更复杂问题的能力;实现网络化 PAR 平台,吸引更多的程序设计爱好者在线参与开发,检验 PAR 平台 C++生成系统和本方法的应用范围和效果.

© 中国科学院软件研究所 http://www.jos.org.cn

致谢 在此,我们向对本文的工作给予支持和建议的同行,尤其是江西师范大学王昌晶副教授、石海鹤副教授、游珍博士、博士生谢武平等同仁表示感谢.同时,对审稿人提出的有益建议表示感谢.

References:

[1] McIlroy MD. Mass-Produced software components. In: Naur P, Randell B, eds. Proc. of the Software Engineering Concepts and Techniques. Brussels: Scientific Affairs Division, NATO, 1969. 138−155.

[2] Girard JY. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur [PH.D. Thesis]. Paris: Université Paris VII, 1972.

[3] Reynolds JC. Towards a theory of type structure. In: Robinet B, ed. Proc. of the Programming Symp. LNCS 19, Berlin, Heidelberg: Springer-Verlag, 1974. 408−425. [doi: 10.1007/3-540-06859-7_148]

[4] Milner R. A theory of type polymorphism in programming. Journal of Computer and System Sciences, 1978,17(3):348−375. [doi: 10.1016/0022-0000(78)90014-4]

[5] Musser DR, Stepanov AA. Generic programming. In: Proc. of the Symbolic and Algebraic Computation. LNCS 358, Berlin, Heidelberg: Springer-Verlag, 1989. 13−25. [doi: 10.1007/3-540-51084-2_2]

[6] Austern MH. Generic Programming and the STL. Beijing: China Electric Power Press, 2003 (in Chinese).

[7] Chen L, Xu BW. Using static analysis to extract C++0X concepts. Chinese Journal of Computers, 2009,32(9):1792−1803 (in Chinese with English abstract). [doi: 10.3724/SP.J.1016.2009.01792]

[8] Xue JY. PAR method and its supporting platform. Technical Report, 348, Macao: UNU-IIST, 2006.

[9] Oliveira BCDS, Moors A, Odersky M. Type classes as objects and implicits. In: Proc. of the ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA 2010). New York: ACM Press, 2010. 341−360. [doi: 10.1145/1869459.1869489]

[10] Kapur D, Musser DR. Tecton: A framework for specifying and verifying generic system componets. Technical Report, RPI-92-20, Troy, Department of Computer Science, Rensselaer Polytechnic Institute, 1992.

[11] Reis GD, Stroustrup B. Specifying C++ concepts. In: Proc. of the Conf. Record of the 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2006). New York: ACM Press, 2006. 295−308. [doi: 10.1145/1111037.1111064]

[12] Cardelli L, Martini S, Mitchell JC, Scedrov A. An extension of system F with subtyping. In: Proc. of the Theoretical Aspects of Computer Software. LNCS 526, Berlin, Heidelberg: Springer-Verlag, 1991. 750−770. [doi: 10.1007/3-540-54415-1_73]

[13] Garcia R, Jarvi J, Lumsdaine A, Siek JG, Willcock J. An extended comparative study of language support for generic programming. Journal of Functional Programming, 2007,17(2):145−205. [doi: 10.1017/S0956796806006198]

[14] Swen B. Object orientation, generic programming and type constraint checking. Chinese Journal of Computers, 2004,27(11): 1492−1504 (in Chinese with English abstract). [doi: 10.3321/j.issn:0254-4164.2004.11.008]

[15] Siek JG. The C++0x "concepts" effort. In: Gibbons J, ed. Proc. of the Generic and Indexed Programming. LNCS 7470, Berlin, Heidelberg: Springer-Verlag, 2012. 175−216. [doi: 10.1007/978-3-642-32202-0_4]

[16] Siek JG, Lumsdaine A. Essential language support for generic programming. In: Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2005). New York: ACM Press, 2005. 73−84. [doi: 10.1145/1065010. 1065021]

[17] Bernardy JP, Jansson P, Zalewski M, Schupp S. Generic programming with C plus plus concepts and Haskell type classes-a comparison. Journal of Functional Programming, 2010,20(3-4):271−302. [doi: 10.1017/S095679681000016X]

[18] Gregor D, Jarvi J, Siek JG, Stroustrup B, Reis GD, Lumsdaine A. Concepts: Linguistic support for generic programming in C++. In: Proc. of the ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006). New York: ACM Press, 2006. 291−310. [doi: 10.1145/1167473.1167499]

[19] Oliveira BCDS, Gibbons J. Scala for generic programmers: Comparing Haskell and Scala support for generic programming. Journal of Functional Programming, 2010,20(3-4):303−352. [doi: 10.1017/S0956796810000171]

[20] Oliveira BCDS, Schrijvers T, Choi W, Lee W, Yi K. The implicit calculus: a new foundation for generic programming. In: Proc. of the 33rd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2012). New York: ACM Press, 2012. 35−44. [doi: 10.1145/2254064.2254070]

[21] Siek JG, Lumsdaine A. A language for generic programming in the large. Science of Computer Programming, 2011,76(5):423−465. [doi: 10.1016/j.scico.2008.09.009]

[22] David V, Haveraaen M. Concepts as syntactic sugar. In: Proc. of the 9th IEEE Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM 2009). California: IEEE Computer Society Press, 2009. 147−156. [doi: 10.1109/SCAM.2009.26]

[23] Sutton A, Maletic JI. Emulating C++ 0x concepts. Science of Computer Programming, 2013,78(9):1449−1469. [doi: 10.1016/ j.scico.2012.10.009]

© 中国科学院软件研究所 http://www.jos.org.cn

[24] Gibbons J, Paterson R. Parametric datatype-genericity. In: Proc. of the 2009 ACM SIGPLAN Workshop on Generic Programming (WGP 2009). New York: ACM Press, 2009. 85−93. [doi: 10.1145/1596614.1596626]

[25] Gregor D, Jarvi J, Kulkarni M, Lumsdaine A, Musser D, Schupp S. Generic programming and high-performance libraries. Int'l Journal of Parallel Programming, 2005,33(2-3):145−164. [doi: 10.1007/s10766-005-3580-8]

[26] Xue JY, Li YQ, Yang QH. Several new patterns of reusable program parts. Journal of Computer Research and Development, 1993, 30(1):39−44 (in Chinese with English abstract).

[27] Shi HH, Xue JY. Research on automated sorting algorithms generation based on PAR. Ruan Jian Xue Bao/Journal of Software, 2012,23(9):2248−2260 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4164.htm [doi: 10.3724/SP.J.1001.2012.04164]

[28] You Z. The analysis of Isabelle theorem prover and its application in PAR method/PAR platform [MS. Thesis]. Nanchang: Jiangxi Normal University, 2009 (in Chinese with English abstract). [doi: 10.7666/d.y1555777]

[29] Kleene SC. Representation of events in nerve nets and finite automata. In: Shannon CE, McCarthy J, eds. Proc. of the Automata Studies. Princeton: Princeton University Press, 1956. 3−41.

[30] Aho AV, Hopcroft JE, Ullman JD, Wrote; Huang LP, Wang DJ, Zhang S, Trans. The design and analysis of computer algorithms. Beijing: China Machine Press, 2007 (in Chinese).

[31] Reis GD, Stroustrup B, Meredith A. Axioms: Semantics aspects of C++ concepts. Technical Report, N2887=09-0077, ISO/IEC JTC1/SC22/WG21, 2009.

[32] Wang CJ, Xue JY. Formal derivation of a generic algorithmic program for solving a class extremum problems. In: Proc. of the 10th ACIS Int'l Conf. on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing (SNPD 2009). California: IEEE Computer Society Press, 2009. 100−105. [doi: 10.1109/SNPD.2009.46]

[33] Wang CJ, Xue JY. Formal derivation of a high-trustworthy generic algorithmic program for solving a class of path problems. In: Proc. of the Frontiers in Algorithmics. LNCS 5598, Berlin, Heidelberg: Springer-Verlag, 2009. 27−39. [doi: 10.1007/978-3-642-02270-8_6]

附中文参考文献:

[6] Austern MH.泛型编程与 STL.北京:中国电力出版社,2003.

[7] 陈林,徐宝文.基于源代码静态分析的 C++0x 泛型概念抽取.计算机学报,2009,32(9):1792−1803. [doi:10.3724/SP.J.1016.2009.01792]

[14] 孙斌.面向对象、泛型程序设计与类型约束检查.计算机学报,2004,27(11):1492−1504. [doi: 10.3321/j.issn:0254-4164.2004.11.008]

[26] 薛锦云,李云清,杨庆红.若干新的可重用程序部件模式.计算机研究与发展,1993,30(1):39−44.

[27] 石海鹤,薛锦云.基于 PAR 的排序算法自动生成研究.软件学报,2012,23(9):2248−2260. http://www.jos.org.cn/1000-9825/4164.htm [doi: 10.3724/SP.J.1001.2012.04164]

[28] 游珍.Isabelle 定理证明器的剖析及其在 PAR 方法/PAR 平台中的应用[硕士学位论文].南昌:江西师范大学,2009. [doi: 10.7666/d.y1555777]

[30] Aho AV, Hopcroft JE, Ullman JD,著;黄林鹏,王德俊,张仕,译.计算机算法的设计与分析.北京:机械工业出版社,2007.

左正康(1980－),男,江西抚州人,博士,讲师,CCF 学生会员,主要研究领域为软件形式化与自动化,泛型程序设计.

薛锦云(1947－),男,教授,博士生导师,CCF 高级会员,主要研究领域为软件形式化与自动化,面向服务的软件工程.

© 中国科学院软件研究所 http://www.jos.org.cn