基于 Haskell 语言的泛型扩展研究

李 阳,于尚超,王 朋
(解放军理工大学 指挥自动化学院 计算机系,江苏 南京 210007)

摘 要:泛型即通过参数化类型来实现在同一份代码上操作多种数据类型。泛型编程是一种编程范式,它利用"参数化类型"将类型抽象化,实现灵活的软件复用。泛型编程思想已经在多种语言中得到运用,并已取得了不小的成果。文中旨在Haskell语言上进行泛型的研究与应用,Haskell语言是一门广为流行的函数式语言,它的计算模型简单,程序语法清晰,易于编写,易于维护。文中利用一些规则对Haskell语言的语法进行扩展,同时引入泛型编程的思想来研究新的函数定义方法,最后在Haskell语言上实现泛型功能。

关键词:泛型编程;类型抽象;函数式程序语言;语法扩展
中图分类号:TP312 **文献标识码**:A **文章编号**:1673-629X(2012)06-0089-04

Generic Extension Research Based on Haskell Language

LI Yang, YU Shang-chao, WANG Peng
(Dept. of Computer, Ins. of Command Automation, PLA Univ. of Techn. & Sci., Nanjing 210007, China)

Abstract: Generic can treat the types as arguments. Then many kinds of data-types can be processed by the same source code. Generic programming is a normal form of programming. It can make types abstract by a method called type parameterization. So realize the goal of software reuse agilely. For the moment, the idea of generic programming has been realized in many languages, which have got great achievement. It is intended to make some research in the field of Haskell language, and give an application as an example. Haskell is a popular language, and it is functional too. It has a lot of advantages, such as simple module, clarity syntax, etc. So the programmer can write and modify Haskell sentence easily. It shows the extension of the grammar of Haskell according to some rules, which also introduces the idea of generic programming, and gives some methods to define functions. Finally, the function of generic would be completed in Haskell.

Key words: generic programming; type abstract; functional programming language; grammar extension

0 引 言

Haskell语言[1]有许多有用的功能,它允许使用显式类型声明来约束函数的类型,它拥有类型类系统可以用于重载函数,重载后的函数对应不同参数具有不同的定义,它还拥有为类型定义类型的种系统用于类型检查,还拥有以其他类型作为参数的列表和元组的构造符等。但Haskell语言缺陷是函数功能的定义比较具体化、单一化,缺乏可扩展性和高度复用性。

在Haskell语言上可以引入泛型编程思想[2,3]解决上述问题,解决方法主要体现在泛型函数的定义上,泛型函数不同于以往的函数,当泛型函数遇到某种未定义的类型参数时,它依靠泛型算法分析参数类型的结构,进行相关转换,可以自动生成函数定义,这种方法可以提高程序的复用程度,优化函数功能的定义。

收稿日期:2011-10-19;**修回日期**:2012-02-01
作者简介:李 阳(1988-),男,江苏淮安人,硕士研究生,研究方向为形式化分析;导师:张兴元,教授,博士生导师,研究方向为软件验证。

1 基于 Haskell 核心语言的语法扩展

1.1 核心语言(core language) FC 与 FCR

为便于研究,先介绍一核心语言(core language) FC,该语言是Haskell的子语言,拥有Haskell的语法体系和规则,Haskell语言经过一定的精简修改即可得到核心语言。泛型功能的扩展可以基于FC语言实现。

首先介绍核心语言FC的语法,如下所示:

Programs:类型声明和主要表达式 P ::= $\{ D_i; \}^{i \in 1 \cdots n}$ main = e

Type declarations:数据类型声明 D ::= data T = $\{ \Lambda\, a_i :: k_i \}^{i \in 1 \cdots l} \{ C_j \{ t_{j,k} \}_j^{k \in 1 \cdots n} \}^{j \in 1 \cdots m}$

Value declarations:值声明 d ::= x = e

Kinds:种类型 k ::= $\star$ | $k_1 \rightarrow k_2$

Types:

t, u ::= a, b, c, f, … 类型变量

| T 数据类型

| $(t_1 t_2)$ 类型应用

| $\forall$ a :: k. t 全称化量词

Expressions:

e ::= x, y, z, … 变量 | C 构造符

| ($e_1 e_2$) 表达式应用 | λ x → e λ 函数

| case e_0 of $\{ p_i \rightarrow e_i \}^{i \in 1 \cdots n}$ case 表达式

| let d in e let 表达式 | fix e 不动点

Patterns:

p ::= ($C\{ p_i \}^{i \in 1 \cdots n}$) 构造符模式

| x, y, z,… 变量模式

由该语法可知,程序由数据类型声明和 main 表达式组成,数据类型声明由一系列构造符构成,每个构造符都有一定的参数。Kinds 是类型的类型,称为"种",表达式中所有类型都可以被赋予种类型,构造符(→)可以构造复杂的种类型。语法中的类型[5]部分包括变量和已定义的数据类型,类型之间可以相互应用,还可以引入全称类型,函数类型并没有包含在类型语法中,但是函数构造符(→)可包含于数据类型的定义中。表达式语法显示变量、构造符、λ 函数、case 语句、let 语句以及不动点函数都可以作为表达式存在于程序中,case 语句包含多种情况讨论,每个讨论分支都是由一个模式到一个表达式的形式,模式是表达式的一种约束形式,它包含构造符和变量。let 语句可以用于定义函数,经过一定转换,let 还有一种用于递归的形式 letrec,fix 语句可用于定义递归。FC 语言引入 letrec 语句即可实现递归功能,称为核心语言 FCR。

1.2 扩展语言 FCR+tif

下面在核心语言的基础上进行语法扩展,首先介绍一种函数,这种函数最明显的特征就是表达式中带有显式类型参数,这种函数称为 Type-indexed 函数[4]。举例说明 Type-indexed 函数的形式:

add⟨Bool⟩ = (∨) add⟨Int⟩ = (+)

add⟨Char⟩ = λ x y→

chr(add⟨Int⟩(ord x)(ord y))

上述函数 add 是定义在 Bool,Int,Char 三个类型上的加函数,这三个函数名都由两部分组成,一部分即 add,存在于三个函数名中,另一部分是包含于尖括号内的类型名称。add 函数是 Type-indexed 函数,它的函数类型表示为:add⟨a::⋆⟩ = a→a→a,该类型表示参数 a 的种类型是⋆,所有 Type-indexed 函数的类型参数的种类型都被约束为⋆。Type-indexed 函数在定义时可以调用其他的 Type-indexed 函数,被调用的函数称为原函数的附属函数[6]。

对核心语言 FC 进行扩展引入 Type-indexed 函数及其相关特性,扩展后的语言即 FCR+tif,扩展后的语法如下所示:

Values declarations:

d ::= x = e 函数声明,来自 FC 语法

| x⟨a⟩ = typecase a of $\{ p_i \rightarrow e_i \}^{i \in 1 \cdots n}$ type-indexed 函数声明

Expression:

e ::= … 来自 FC 语法

| x⟨A⟩ type-indexed 函数

Type patterns:

P ::= $T\{ \alpha_i \}^{i \in 1 \cdots n}$ 带参数据类型

Type arguments:

A ::= T 数据类型

|α, β, γ, … 附属变量

| ($A_1 A_2$) 类型应用

其中构造符 typecase 被用于 Type-indexed 函数的声明,且不同的分支语句拥有不同的类型模式[7]匹配相应的表达式。Type-indexed 函数的类型参数的形式比较复杂,可以是数据类型、可以是附属变量,也可以是更复杂的类型应用。例如函数 add⟨[a]⟩,它的参数是一种类型应用,它的类型为:add⟨[a]⟩:: ∀ a:: ⋆. (add⟨a⟩:: a→a→a) ⇒ [a] →[a] →[a],该类型表示 add 函数对自己进行了调用,双箭头左边的小括号里面的内容称为附属约束[8],这意味着只有当函数 add⟨a⟩被定义且它的类型为 a→a→a 时,add⟨[a]⟩的类型才能为[a] →[a] →[a],附属约束类似于 Haskell 语言中的类型类约束[9]。Type-indexed 函数的这种复杂参数形式称为多态类型参数[10],多态类型参数中可以存在一些变量,这些变量称为附属变量,每个附属变量都有对应的附属约束存在,附属变量仅允许出现在 Type-indexed 函数的参数和其附属约束以及所调用的 Type-indexed 函数的参数中。add 函数的附属函数是其本身,而对于某些函数它的附属函数可能是其他函数或者没有附属函数,而对应的附属约束可以根据函数参数中的附属变量的位置来判断是否需要:如果一个函数的参数的形式为 $A_0\{ A_i \}^{i \in 1 \cdots n}$,且 A_0 中不包含任何进一步的运算,那么 A_0 称为参数的头(head),可以用 head(A)表示。如果一个附属变量 α 是参数 A 的 head,那么函数 x⟨A⟩将产生一个附属约束 x⟨α⟩,如果 α 存在于 A 中但是不是 A 的 head,那么函数 x⟨A⟩关于变量 α 的附属约束为每个附属函数 y_k 应用到 α 的运算即 $y_k \langle \alpha \rangle$[11]。

1.3 语言 FCR+tif 到语言 FCR 的转换

下面介绍如何将 FCR+tif 语言转换成 FCR 语言,首先介绍图 1 中的两个转换规则:声明转换规则和表达式转换规则。规则中引入了环境的定义:

Environments: E::= ε 空环境

| E,x←v 非空环境

x←v 表示元素 x 被赋予值 v,根据定义可将环境看作为一个列表。Type-indexed 函数中定义的类型称为类型签名[12],用符号 Σ 表示类型签名环境,前面定

义的 add 函数的类型签名就是{Bool, Int, Char}。环境 Σ 的元素通常表示为 x⟨T⟩这样的形式,该形式表示类型 T 就存在于 Type-indexed 函数 x 的类型签名中。在上述规则中,环境 Σ_1 称为输入环境,转换发生需要以 Σ_1 为基础,Σ_2 称为输出环境,包含着声明中分析出来的类型签名,这些环境仅在表达式转换的时候需要进行处理。

$$[\![d_{\mathrm{FCR+tif}}]\!]^{\mathrm{tif}}_{\Sigma_1} \equiv \{d_{\mathrm{FCR}}\}_{;}^{i\in 1..n} \sim \Sigma_2$$

$$\frac{\left\{d_i \equiv \mathrm{cp}(x,T_i) = [\![e_i]\!]^{rif}\right\}^{i\in 1..n}}{[\![x\langle a\rangle = \mathrm{typecase}\ a\ \mathrm{of}\ \{T_i \to e_i\}_{;}^{i\in 1..n}]\!]^{tif} \equiv \{d_i\}_{;}^{i\in 1..n} \sim \{x\langle T_i\rangle\}^{i\in 1..n}} (\text{tr-tif})$$

$$\frac{}{[\![x=e]\!]^{\mathrm{tif}} \equiv x = [\![e]\!]^{\mathrm{tif}} \sim \varepsilon} (\text{tr-fdecl})$$

$$[\![e_{\mathrm{FCR+tif}}]\!]^{\mathrm{tif}}_{\Sigma} \equiv e_{\mathrm{FCR}}$$

$$\Sigma' \equiv \Sigma\{,\Sigma_i\}^{i\in 1..n}$$

$$\frac{\left\{[\![d_i]\!]^{\mathrm{tif}}_{\Sigma'} \equiv \{d_{i,j}\}^{j\in 1..m_i} \sim \Sigma_i\right\}^{i\in 1..n}}{[\![\mathrm{let}\{d_i\}_{;}^{i\in 1..n}\ \mathrm{in}\ e]\!]^{\mathrm{tif}}_{\Sigma} \equiv \mathrm{let}\left\{\{d_{i,j}\}_{;}^{j\in 1..m_i}\right\}_{;}^{i\in 1..n}\ \mathrm{in}\ [\![e]\!]^{\mathrm{tif}}_{\Sigma}} (\text{tr-let})$$

$$\frac{x\langle T\rangle \in \Sigma}{[\![x\langle T\rangle]\!]^{\mathrm{tif}}_{\Sigma} \equiv \mathrm{cp}(x,T)} (\text{tr-named})$$

$$\frac{}{[\![x\langle\alpha\rangle]\!]^{\mathrm{tif}} \equiv \mathrm{cp}(x,\alpha)} (\text{tr-dep var})$$

$$\frac{\mathrm{dependencies}_{\Gamma}(x) \equiv \{y_k\}_{;}^{k\in 1..l}}{[\![x\langle A_1A_2\rangle]\!]^{\mathrm{tif}}_{\Sigma} \equiv [\![x\langle A_1\rangle]\!]^{\mathrm{tif}}_{\Sigma}\ \left\{[\![y_k\langle A_2\rangle]\!]^{\mathrm{tif}}_{\Sigma}\right\}^{k\in 1..l}} (\text{tr-app})$$

图1　声明和表达式的转换规则

声明转换规则表示一个 Type-indexed 函数的声明可以转换成一组具有不同名称的且符合核心语言语法的函数声明。规则中的 cp 表示一个组件,它包含一个函数名以及函数的一个类型参数,它由 Type-indexed 函数的某个单一 case 语句转换而成,与此同时,Type-indexed 函数定义的类型签名存储在输出环境中,而一般形式声明的函数的输出环境为空。

表达式转换规则显示了不同形式的表达式的转换过程,规则(tr-let)表示了由 let 语句构成的表达式的转换过程,环境 Σ' 包含了所有 let 语句中的声明转换时产生的组件。规则(tr-named)和(tr-depvar)表示函数类型参数为数据类型或者附属变量时可直接转换成合适的组件。规则(tr-app)表示类型参数为类型应用时,附属函数的转换组件将作为显式参数出现在表达式转换结果中,如果附属函数也有自己的附属函数,那么转换将会传递下去。例如函数 x⟨$T_1(T_2\alpha)$⟩,其中函数 x 的定义调用了函数 y 和函数 z,而函数 y 的定义也调用了函数 z,因此函数将被转换为

cp(x, T_1)(cp(x, T_2) cp(x, α) cp(y, α) cp(z, α))
(cp(y, T_2) cp(y, α) cp(z, α))
(cp(z, T_2) cp(z, α))

转换过程还需要考虑其他一些因素,例如环境 Σ 是扩展后的语言 FCR+tif 所特有的,语言 FCR 并不存在这样的环境,FCR 中所定义的类型环境 Γ 只存储变量类型和 FCR 的函数类型,它并不包含 Type-indexed 函数的类型。因此在语言转换的同时要考虑 Σ 和 Γ 的转换,它们适用于图2中的规则,规则$[\![\Gamma_{\mathrm{FCR+tif}}]\!]^{\mathrm{tif}} = \Gamma_{\mathrm{FCR}}$表示转换时 $\Gamma_{\mathrm{FCR+tif}}$ 中的 Type-indexed 函数的类型被去掉后即可成为 Γ_{FCR},规则$[\![\Gamma_{\mathrm{FCR+tif}}]\!]^{\mathrm{tif}} = \Gamma_{\mathrm{FCR}}$表示转换时环境中的表达式 x⟨T⟩需要转换成相应的 cp 形式。

$$[\![\Gamma_{\mathrm{FCR+tif}}]\!]^{\mathrm{tif}} \equiv \Gamma_{\mathrm{FCR}}$$

$$\frac{}{[\![\varepsilon]\!]^{\mathrm{tif}} \equiv \varepsilon} \qquad \frac{}{[\![\Gamma, x :: t]\!]^{\mathrm{tif}} \equiv [\![\Gamma]\!]^{\mathrm{tif}}, x :: t}$$

$$\frac{}{[\![\Gamma, x\langle a :: *\rangle :: t]\!]^{\mathrm{tif}} \equiv [\![\Gamma]\!]^{\mathrm{tif}}}$$

$$[\![\Sigma_{\mathrm{FCR+tif}}]\!]^{\mathrm{tif}} \equiv \Gamma_{\mathrm{FCR}}$$

$$\frac{}{[\![\varepsilon]\!]^{\mathrm{tif}} \equiv \varepsilon}$$

$$\frac{x\langle a :: *\rangle :: t \in \Gamma}{[\![\Sigma,\ x\langle T\rangle]\!]^{\mathrm{tif}} \equiv [\![\Sigma]\!]^{\mathrm{tif}}_{\Gamma}, \mathrm{cp}(x,T) :: t\left[T/a\right]}$$

图2　环境的转换规则

2　Haskell 语言泛型功能实现

2.1　数据类型的结构化描述

众所周知,数据类型是由构造符构成的,每个构造符都有一些参数。基于这些结构特点,通过某些机制可以将数据类型转换成一种特殊的形式,下面引入三个特殊的数据类型:

```
data Unit = Unit
data Sum (a::*)(b::*) = Inl a | Inl b
data Prod(a::*)(b::*) = a * b
```

这三个类型在泛型函数的定义与使用中具有特殊的意义,通过它们可以对数据类型进行转换,使用 Sum 来表示多个构造符的情形,使用 Prod 来表示字段序列,使用 Unit 表示构造符为空的情况,构造符的参数不需要转换,转换的结果称为数据类型的结构化描述[13]。经过这样的转换可以将一个很大的类型类转换成只包含这三个数据类型的类。

下面通过引入实例来介绍这样的泛型机制,一个

二叉树的类型可以定义为：

data Tree (a::*) = Leaf | Node (Tree a) a (Tree a)，它可以转换为 Str(Tree)：type Str(Tree) (a::*) = Sum Unit (Prod (Tree a) (Prod a (Tree a)))，Str(Tree)称为数据类型 Tree 的结构化描述。所有类型与它们的结构化描述都是同构[14]的，可以使用映射组合[15]来表示它们的同构特性，假设一个数据类型声明：data EP (a::*)(b::*) = EP(a→b)(b→a)，如果类型 T 和它的结构化描述 Str(T)是同构的，且 T 的种类型为 $\{k_i \rightarrow\}^{i\in 1\cdots n}$，那么存在映射类型为：

$ep(T) :: \{\forall a_i :: k_i .\}^{i\in 1\cdots n}$

$EP(T\{a_i\}^{i\in 1\cdots n})\ (Str(T)\ \{a_i\}^{i\in 1\cdots n})$

而 ep(T) = EP from to，其中

$from \cdot to \equiv id :: \{\forall a_i :: k_i .\}^{i\in 1\cdots n}$

$Str(T)\ \{a_i\}^{i\in 1\cdots n} \rightarrow Str(T)\ \{a_i\}^{i\in 1\cdots n}$

$to \cdot from \equiv id :: \{\forall a_i :: k_i .\}^{i\in 1\cdots n}$

$T\{a_i\}^{i\in 1\cdots n} \rightarrow T\{a_i\}^{i\in 1\cdots n}$

对于类型 Tree，它的映射组合可以定义为：

```
ep(Tree):: ∀ a::*. EP (Tree a) (Str (Tree) a)
ep(Tree)=
let fromTree Leaf= Inl Unit
fromTree(Node l x r) = Inr (l*(x*r))
toTree (Inl Unit) = Leaf
toTree (Inr(l*(x*r))) = Node l x r
in EP fromTree toTree
```

2.2 定义泛型函数

由前文可知，Type-indexed 函数 x〈T〉可以转换成 cp(x, T)，但是有一个前提即 T 必须存在于函数 x 的类型参数环境中，但是利用结构化描述这一特性可以为那些不存在于类型参数环境中的数据类型生成组件 cp。例如 add 函数在基本类型上可以定义为：

```
add〈Bool〉=(∨) add〈Int〉=(+)
add〈Char〉=λ x y→
chr(add〈Int〉(ord x)(ord y))
add〈Unit〉Unit Unit = Unit
add〈Prod α β〉(x1*x2) (y1*y2)=
(add〈α〉x1 y1)*(add〈β〉x2 y2)
add〈Sum α β〉(Inl x)(Inl y)= Inl (add〈α〉x y)
add〈Sum α β〉(Inr x)(Inr y)= Inr (add〈β〉x y)
add〈Sum α β〉_ _=
error"args must have same shape"
```

函数 add 的类型参数只包含一些基本类型，并不包含类型 Tree，但是根据公式：

add〈Tree α〉x y = add〈Str(Tree) α〉(from ep (Tree) x) (from ep(Tree) y)

可以自动生成函数 add 在 Tree 上的定义，转换成组件形式可以表示成：

cp (add, Tree) = λcp(add, α)→λx→λy→ to ep (Tree) (cp(add, Str(Tree)) cp(add, α) (from ep (Tree) x) (from ep(Tree) x))

可知欲定义 cp (add, Tree)，就必须先定义 cp (add, Str(Tree))，根据前文已知：

type Str(Tree) (a::*)= Sum Unit (Prod (Tree a) (Prod a (Tree a)))，add 函数的附属函数是其本身，因此可得：

cp(add, Str(Tree)) = λcp(add, α)→cp(add, Sum) cp(add, Unit) (cp(add, Prod) (cp(add, Tree) cp(add, α)) (cp(add, Prod) cp(add, α)(cp(add, Tree) cp(add, α))))

函数 cp(add, Str(Tree))和 cp(add, Tree)是相互递归的，继而可得到 cp(add, Tree)的定义。

由上可知，只要函数为类型 Sum、Unit、Prod 赋予了定义，就可以通过类型转换的方法实现众多未知的类型的函数定义，当然如果函数拥有很多附属函数或者类型参数较为复杂，那么转换过程也将更加复杂。

语言 FCR+tif 在类型参数环境中添加 Unit，Sum 和 Prod 三个数据类型的定义并修改相关转换可以使得函数对类型参数环境以外的类型自动生成定义，继而实现泛型的功能。能够在 Haskell 语言中实现泛型函数的定义即基本完成了 Haskell 语言的泛型扩展，下一步的工作即以此为基础做出相关完善。

3 结束语

泛型编程思想经过几十年的发展，已经形成了比较成熟的理论体系，并已经在 C++[16]、Java[17]、C#、ML[18]等多种语言上得到实现与应用，取得了很好的成果。文中将泛型编程思想引用到了函数式语言 Haskell 上，在 Haskell 语言上实现了泛型功能，泛型 Haskell 提高了程序的复用性、优化了函数的定义方法，使得 Haskell 语言在实际工作中更加实用。当然，在目前的工作基础上还可以进行进一步的扩展，使得程序的表达形式更加清晰、容易理解，使得语言的语法系统更加成熟、适用。

参考文献：

[1] Jones S P, Shields M. Practical type inference for arbitrary-rank types[R]. Microsoft Research, 2003.

[2] 陈叶旺，余金山. 泛型编程与设计模式[J]. 计算机科学，2006，33(4)：253-257.

[3] 吴拥民. 泛型设计的理论研究[J]. 闽江学院学报，2006，94

(下转第96页)

使用图5程序安全性判定算法遍历树形结构：

(1)访问 bb0。stat. value=0,k. value=8,变量 a 的属性:a. alloc=2,a. len=0。转到(2)。

(2)访问 bb1。若分支条件 k^>0 满足转到(3),否则转到(5)。

(3)遍历 bb1 的左子树 bb2。一般表达式 a[stat^|*4|]:=k^,使得 a. len++;函数调用表达式 call func(k^),访问子函数 func,使得全局变量 stat. value 加1;一般表达式 k:=k^-1,使得 k. value 减1。转到(4)。

(4)访问 bb3。转到(2)。

(5)访问 bb4。转到(6)。

(6)访问 bb5,结束。

遍历结果为:在第三次执行步骤(3)后,a. len=3,即 a. len >a. alloc。属于不安全操作,程序中断,给出错误提示。

6 结束语

针对 CoSy C 语言编译器的输入程序安全性问题,提出了重建 CoSy 中间表示 CCMIR 的方法,为所关心变量附加安全属性,通过程序安全性判定算法来分析程序安全性。

该方法具有可扩展性,可用于分析具有树形结构中间表示的其它编译器平台的输入程序的安全性。

参考文献:

[1] ACE Associated Compiler Experts bv. CCMIR Primer[EB/DK]. Amsterdam,The Netherlands:[s. n.],2008.

[2] ACE Associated Compiler Experts bv. CCMIR Definition[EB/DK]. Amsterdam,The Netherlands:[s. n.],2008.

[3] ACE Associated Compiler Experts bv. CoSy Tutorial[EB/DK]. Amsterdam,The Netherlands:[s. n.],2008.

[4] 高 攀. C 语言安全编译器研究[D]. 成都:电子科技大学,2004.

[5] 魏 强,金 然,王清贤. 基于中间汇编的缓冲区溢出检测模型[J]. 计算机工程,2009(3):169-172.

[6] 匡春光,王春雷,刘 强,等. C 库中易受缓冲区溢出攻击的脆弱函数分析[J]. 微电子学与计算机,2011(2):189-192.

[7] 杨小龙,刘 坚. C/C++源程序缓冲区溢出漏洞的静态检测[J]. 计算机工程与应用,2004(20):108-110.

[8] 黄玉文,刘春英,李肖坚. 基于可执行文件的缓冲区溢出检测模型[J]. 计算机工程,2010(2):130-131.

[9] 赵奇永,郑燕飞,郑 东. 基于可执行代码的缓冲区溢出检测模型[J]. 计算机工程, 2008(12):120-122.

[10] 丁永尚,何福男. 关于缓冲区溢出漏洞的解决方法[J]. 计算机系统应用,2010(2):192-194.

[11] 王业君,倪惜珍,文伟平,等. 缓冲区溢出攻击原理与防范的研究[J]. 计算机应用研究, 2005(10):101-104.

[12] 严蔚敏,吴伟民. 数据结构[M]. 北京:清华大学出版社,2006:118-170.

(上接第92页)

(2):72-76.

[4] Löh A. Exploring Generic Haskell[D]. Netherlands:Utrecht University,2004.

[5] Pierce B C. Types and Programming Languages[R]. [s. l.]:The MIT Press,2002.

[6] Löh A, Clarke D, Jeuring J. Dependency-style Generic Haskell[C]//Proceedings of the eighth ACM SIGPLAN international conference on functional programming. [s. l.]:ACM Press,2003:141-152.

[7] Jay B. The pattern calculus[J]. ACM Transactions on Programming Languages and Systems,2004,26(6):911-937.

[8] 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.

[9] Wadler P,Blott S. How to make ad-hoc polymorphism less ad-hoc[C]//Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. [s. l.]:ACM Press,1989:60-76.

[10] Pitts A. Parametric polymorphism and operational equivalence [J]. Mathematical Structures in Computer Science, 2000 (10):231-259.

[11] Altenkirch T,McBride C. Generic programming within dependently typed programming[C]//IFIP TC2/WG2. 1 Working Conference on Generic Programming. Germany:[s. n.], 2002:1-20.

[12] Hinze R,Jeuring J,Löh A. Type-indexed Data Types:Mathematics of Program Construction[C]//Sixth International Conference,Volume 2386 of Lecture Notes in Computer Science. [s. l.]:[s. n.],2002:148-174.

[13] Lammel R,Visser J,Kort J. Dealing with Large Bananas[C]//Proc. of WGP'2000. Utrecht:Universiteit Utrecht,2000:46-59.

[14] Ehrig H. Applied and computational category theory[J]. European Association for Theoretical Computer Science,2006(6):134-135.

[15] Fegaras L,Sheard T. Revisiting catamorphisms over datatypes with embedded functions[C]//Conference Record of POPL'96:the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. [s. l.]:[s. n.],1996:284-294.

[16] 韩玉坤,王冬星. 浅谈 C++中泛型编程方法的运用[J]. 电脑学习,2007(2):47-48.

[17] 许文胜,薛锦云. 泛型编程扩展及其 JAVA 实现[J]. 计算机工程与科学,2007,29(10):89-94.

[18] 缪伟宇,邵志清. 使递归算法泛型化[J]. 计算机技术与发展,2008,18(7):96-99.