O'Caml 泛型编程中"泛型抽象"的研究

王 朋,徐 健,于尚超
(解放军理工大学 指挥信息系统学院,江苏 南京 210007)

摘 要:泛型编程旨在编写一般化并可重复使用的算法,主要目的是提高程序的复用性,其效率与针对某特定数据类型而设计的算法相同。泛型编程可以使算法与数据结构完全分离,极大提高了程序的灵活性。在O'Caml语言中已经实现了类型标记函数和泛型函数,但泛型函数的类型参数只能是基本类型或实例化类型,不能使用抽象类型。"泛型抽象"是指使用抽象类型作为类型参数的函数定义方法,实现了"泛型编程"类型参数的多样化。文中在O'Caml语言上进行"泛型抽象"的研究,根据规则对O'Caml语言语法进行扩展,并在O'Caml语言中实现了"泛型抽象"函数定义。

关键词:泛型编程;函数式程序设计;语法扩展;泛型抽象
中图分类号:TP31 **文献标识码**:A **文章编号**:1673-629X(2013)07-0092-04
doi:10.3969/j.issn.1673-629X.2013.07.023

Research on Generic Abstract in O'Caml Generic Programming

WANG Peng, XU Jian, YU Shang-chao
(College of Command Information System, PLA Univ of Tech & Sci, Nanjing 210007, China)

Abstract: The main idea of generic programming is to design an algorithm that is generic and can be reused. The purpose is to reuse of shuttering. The efficiency of the generic function is equal to the function which is designed for some special data type. Generic programming can completely separate the algorithm and the data type, and make the function more flexible. O'Caml has been achieved type-indexed function and generic programming. Nevertheless, the type argument must be one concrete type. Generic abstraction allows people to abstract from the type argument in the definition of functions. It shows the extension of the grammar of O'Caml based on some rules, which also introduces the idea of generic abstract programming, and gives some methods to define generic abstract functions. Finally, the function of generic abstraction would be completed in O'Caml.

Key words: generic programming; functional programming; grammar extension; generic abstraction

0 引 言

"泛型编程"[1]实现一个通用的标准容器库,从而可以编写完全一般化并可重复使用的算法,其效率与针对某特定数据类型而设计的算法相同。O'Caml[2]与Haskell都是比较流行的函数式语言。"泛型编程"与"泛型抽象"(Generic Abstract)已由Andres博士与Hinze博士等在Haskell语言中实现[3~5],但是O'Caml语言中的"泛型编程"研究还处于起步阶段。李阳在O'Caml语言中实现了"类型标记函数"和"泛型函数"[6],但没有在O'Caml中实现"泛型抽象"。文中在原有O'Caml泛型编程的基础上增加了"泛型抽象"的语法扩展。

文中的主要工作:

(1)在Haskell"泛型抽象"的转化规则中使用kind来描述类型的类型。O'Caml语言中没有kind概念,因此不能直接使用Haskell中"泛型抽象"的转化规则。文中通过修改Haskell中"泛型抽象"转化规则,使其适用于O'Caml语言的"泛型抽象"转化规则,并结合实例对规则作了详细说明。

(2)对Camlp5[7]源代码进行全局分析,找出了对O'Caml进行语法扩展需要修改的文件。根据修改后的转化规则,提出在Camlp5中对O'Caml语言进行"泛型抽象"语言扩展方案。

(3)重新安装修改后的Camlp5工具,在新的编译环境下对使用"泛型抽象"的函数进行测试。实验结果表明使用"泛型抽象"可以实现类型参数多样化,提高了"泛型编程"的灵活性。

收稿日期:2012-09-07 **修回日期**:2012-12-24 **网络出版时间**:2013-04-08
基金项目:国家"863"高技术发展计划项目(2008AA01A309)
作者简介:王 朋(1988-),男,山东东营人,硕士研究生,CCF会员,研究方向为形式化分析。
网络出版地址:http://www.cnki.net/kcms/detail/61.1450.TP.20130408.1631.042.html

1 "泛型抽象"相关描述

本节以size函数为例说明"泛型抽象"的基本含义。式(1)中的泛型函数可以计算"列表"(list)的长度,式(2)中的泛型函数可以计算二叉树的节点数。

(let size ⟨ α :: * ⟩ = const 1 in size⟨list α⟩) [1; 2; 3; 4; 5]　　(1)

(let size ⟨α:: * ⟩ = const 1 in size⟨tree α⟩)
(Node leaf 'x' leaf)　　(2)

list α与tree α分别为size⟨list α⟩和size⟨tree α⟩的类型参数,这种带有类型参数的函数称之为"类型标记函数"。size ⟨ α :: * ⟩中的类型参数为α,α :: * 表示α的kind为*。kind表示类型的类型[8],构造符→可以构造复杂类型的类型。kind为* → *的类型称之为"抽象类型",kind为*的类型称之为"实例化类型"。例如tree为"抽象类型",其kind为* → *;而tree α为"实例化类型",其kind为*。根据"泛型编程"的转化规则,size⟨list α⟩和size⟨tree α⟩在编译时都需要调用size ⟨ α ⟩,文献[5]对此有详细描述。因此,可以使用"局部重定义"方式在let...in语句中给出size ⟨ α ⟩的定义,从而计算相应数据的大小。

如果在程序书写过程中对列表和树求大小的函数使用频率较高,那么式(1)与式(2)可以定义为式(3)的形式。

sizeList = let size ⟨α:: * ⟩ = const 1 in size⟨list α⟩
sizeTree = let size ⟨α:: * ⟩ = const 1 in size⟨tree α⟩　　(3)

由式(3)知,两个式子等号右边唯一不同的是泛型函数的类型参数。一个为tree,一个为list。如果把这两个类型参数抽象出来作为一个类型变量,那么式(3)中的两个函数也可以统一写作式(4)中的函数形式。

fsize⟨ γ :: * → * ⟩ =let size⟨ α ⟩ =const 1 in size⟨ γ α ⟩　　(4)

式(4)中γ为函数fsize⟨ γ ⟩的类型参数,其kind为* → *。因此,fsize⟨ γ ⟩的类型参数γ为"抽象类型"。这种定义泛型函数的方法称为"泛型抽象"。式(3)中的sizeList和sizeTree函数可以用fsize⟨list⟩和fsize⟨tree⟩表示。"泛型抽象"函数声明把类型参数中的一部分抽象出来,以"抽象类型"作为类型参数,在更高层次上对函数进行抽象,极大提高了"泛型编程"的灵活性。

因为O'Caml中"泛型编程"只支持类型参数kind为*的函数定义,而式(4)中fsize函数类型参数的kind为* → *,所以O'Caml"泛型编程"不支持式(4)中fsize函数的定义方式。因此fsize函数需要转化成O'Caml可以处理的常规语句。

为了使O'Caml可以处理fsize⟨ γ ⟩的函数定义,在O'Caml编译器中增加了对fsize函数进行处理的分支。具体的做法是:首先在O'Caml编译器内部将fsize⟨ γ ⟩表示为fsize⟨Abs γ ⟩,其中Abs是kind为κ → *的类型。对于任意的类型α,类型Abs α的kind都为*。然后把fsize⟨Abs γ ⟩按照"泛型编程"的转化规则[6]转化为cp表示的组件形式cp(fsize Abs) cp(size γ)。转化过程如式(5)所示。

$$\begin{aligned}&[\text{fsize}\langle\gamma::*\to*\rangle = \text{let size}\langle\alpha\rangle = \text{const 1 in size}\langle\gamma\,\alpha\rangle]\\ \Rightarrow&[\text{fsize}\langle\text{Abs}\,\gamma\rangle = \text{let size}\langle\alpha\rangle = \text{const 1 in size}\langle\gamma\,\alpha\rangle]\\ \Rightarrow&\left[\begin{array}{l}\text{cp(fsize,Abs) cp(size,}\gamma\text{) = let cp(size,}\alpha\text{) = const 1}\\ \qquad\qquad\text{in cp(size,}\gamma\text{) cp(size,}\alpha\text{)}\end{array}\right]\end{aligned}　　(5)$$

式(5)中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。"组件"在O'Caml中可以当作一个函数使用。等号左边的cp(size, γ)作为cp(fsize,Abs)的参数,等号右边cp(size, α)作为cp(size, γ)的参数,并在let...in语句中给出其定义。

以式(4)为例,fsize⟨list⟩表示式(3)中sizeList函数。函数fsize⟨list⟩被调用时,可以转化为cp(fsize, Abs) cp(size, γ)的形式。根据式(5)中转化结果的定义即可得到fsize⟨list⟩的定义。

2 "泛型抽象"转化规则

为了使O'Caml泛型编程可以处理式(4)中fsize函数定义,加入式(6)与式(8)描述的规则。使用这两个规则,可以把式(4)中函数转化成O'Caml可以处理的代码。

$$\begin{aligned}&\{d_{\text{FCR}+gf+\text{gabs}}\mapsto\Gamma_2;\Sigma_2\}^{\text{gabs}}_{\Gamma_1;\Sigma_1} \equiv \{d_{\text{FCR}}\}\\ &\Sigma' \equiv x\langle\text{Abs}\rangle;\Gamma' \equiv x\langle\pi\rangle::\sigma\\ &\text{gapp}(x\langle\text{Abs}\,A\rangle) \equiv q\\ &\frac{\{e::q\}^{\text{gabs}}_{\Gamma_1;\Sigma} \equiv e''[e']}{\{x\langle A::\kappa\rangle = e\mapsto\varepsilon;\Sigma'\}^{\text{gabs}}_{\Gamma_1;\Sigma_1}}(d/\text{tr}-\text{gabs})\\ &\equiv \{\text{cp}(x,\text{Abs}) = e''[e']\}\end{aligned}　　(6)$$

"泛型抽象"函数转化为普通O'Caml代码规则如式(6)所示。式(6)中双横线以上的为规则基本形式,双横线以下的为规则的具体内容。规则(d/tr-gabs)输入为x⟨A::κ⟩ = e,输出为{cp(x, Abs)= e''[e']}形式的代码。其中,Σ表示"签名环境",用于存储所有的组件。Γ表示"类型签名环境",用于存储函数名、类型参数、依赖函数和基本类型组成的四元组列表。以式(4)为例,按照式(6)中规则具体计算过程如式(7)所示。

$$
\begin{aligned}
&[\mathrm{fsize}\langle\gamma::*\to*\rangle = \text{let size}\langle\alpha\rangle = \text{const 1 in size}\langle\gamma\,\alpha\rangle]\\
\Rightarrow&\begin{bmatrix} x = \mathrm{fsize};\Sigma = [\,];\Gamma = [(\mathrm{fsize};f;[\mathrm{size}\langle f\rangle];f\alpha\to\mathrm{Int})]\\ A = \gamma;\ e = \{\text{let size}\langle\alpha\rangle = \text{const 1 in size}\langle\gamma\,\alpha\rangle\}\end{bmatrix}\\
\Rightarrow&\begin{bmatrix} q \equiv \mathrm{gapp}_\Gamma(\mathrm{fsize}\langle\mathrm{Abs}\rangle) = \mathrm{size}\langle\gamma\rangle\ ;\\ \Sigma = (\mathrm{fsize}\langle\mathrm{Abs}\rangle)\ ;\Sigma' = [(\mathrm{fsize}\langle\mathrm{Abs}\rangle)]\end{bmatrix}\\
\Rightarrow&\begin{bmatrix} \{e::q\}^{\mathrm{gabs}}_{\Gamma;\Sigma} \equiv \text{fun } q'\to e';q' = \mathrm{cp}(\mathrm{size},\gamma)\\ e' = \text{let cp(size},\alpha) = \text{const 1 in cp(size},\gamma)\ \mathrm{cp}(\mathrm{size},\alpha)\end{bmatrix}\\
\Rightarrow&\begin{bmatrix} \mathrm{cp}(\mathrm{fsize},\mathrm{Abs})\ \mathrm{cp}(\mathrm{size},\gamma) = \text{let cp(size},\alpha) = \text{const 1}\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{in cp(size},\gamma)\ \mathrm{cp}(\mathrm{size},\alpha)\end{bmatrix}
\end{aligned}
\tag{7}
$$

规则 $\Gamma \equiv x\langle\pi\rangle::\sigma$ 用来找出 x 的“类型签名”(type signature),fsize 的“类型签名”为 $\mathrm{fsize}\langle f::*\to*\rangle::(\mathrm{size}\langle f\rangle)\Rightarrow f\alpha\to\mathrm{Int}$。符号⇒之后的 $f\alpha\to\mathrm{Int}$ 表示 fsize 的基本类型,⇒之前::之后的 $\mathrm{size}\langle f\rangle$ 表示 fsize 函数依赖于 $\mathrm{size}\langle f\rangle$ 函数。

在表达式中调用“泛型抽象”函数与一般的类型标记函数不同,因为直接调用组件 cp(fsize,Abs)不具有任何实际意义。因此,对于“泛型抽象”函数表达式的调用转化规则如式(8)所示。

$$
\frac{\{e_{\mathrm{FCR}+gf+\mathrm{gabs}}\}^{\mathrm{gabs}}_{\Gamma;\Sigma} \equiv \{e_{\mathrm{FCR}}\}}{\dfrac{\begin{array}{c}x\langle\mathrm{Abs}\rangle \notin \Sigma\\ \mathrm{gapp}_\Gamma(x\langle A\rangle) \equiv q;\{x\langle A\rangle\}^{\mathrm{gabs}}_{\Gamma;\Sigma} \equiv e\end{array}}{\{x\langle A\rangle::t\}^{\mathrm{gabs}}_{\Gamma;\Sigma} \equiv e}(e/\mathrm{tr}-\mathrm{genapp})}\\
\frac{\begin{array}{c}x\langle\mathrm{Abs}\rangle \in \Sigma\\ \mathrm{gapp}_\Gamma(x\langle\mathrm{Abs}\ A\rangle) \equiv q;\{x\langle\mathrm{Abs}\ A\rangle\}^{\mathrm{gabs}}_{\Gamma;\Sigma} \equiv e\end{array}}{\{x\langle A\rangle::t\}^{\mathrm{gabs}}_{\Gamma;\Sigma} \equiv e}(e/\mathrm{tr}-\mathrm{gabs})
\tag{8}
$$

泛型函数表达式的转化规则由(e/tr - genapp)和(e/tr - gabs)组成。$x\langle\mathrm{Abs}\rangle \notin \Sigma$ 表示 $x\langle\mathrm{Abs}\rangle$ 的签名不在签名环境 Σ 中,即 x 是普通泛型函数,使用规则(e/tr - genapp)进行转化。$x\langle\mathrm{Abs}\rangle \in \Sigma$ 表示 x 是“泛型抽象”函数,使用规则(e/tr - gabs)进行转化。(e/tr - gabs)规则用来转化“泛型抽象”函数的表达式。在处理表达式 $x\langle A\rangle$ 时,利用 $\mathrm{gapp}_\Gamma(x\langle A\rangle) \equiv q$ 计算 x 的“依赖函数”q。gtrans 算法[6] 根据以上信息以及 $x\langle\mathrm{Abs}\ A\rangle$ 计算出符合 O'Caml 基本语法的表达式 e。以式(4)为例。当调用 $\mathrm{fsize}\langle\mathrm{tree}\rangle$ 时,由于 $\mathrm{fsize}\langle\mathrm{Abs}\rangle \in \Sigma$,所以使用规则(e/tr - gabs)进行转化,最终转化为 cp(fsize,Abs) cp(size,tree)形式。

3 根据规则搭建环境

为了在 O'Caml 中实现“泛型抽象”的语法扩展,利用 Camlp5 工具对 O'Caml 语言进行语法扩展。Camlp5 是函数式语言的预处理器和漂亮格式打印工具,是 O'Caml 语言编译器可扩展的前端。作为一个预处理器,它可以扩展函数式语言的语法,或者重新定义语言的整个语法。

经过对 Camlp5 工具源代码进行分析,“泛型抽象”的实现只需要修改 pa_o. ml 文件的源代码。文中主要工作为分析并按照规则修改代码。表 1 为工作量统计。

表 1 工作量统计

内容	分析源代码	语法扩展	函数声明转化	表达式转化
代码量	约 32400 行	82 行	205 行	135 行

要在 O'Caml 中加入“泛型抽象”函数,就需要在 O'Caml 中加入“泛型抽象”的语法规则,如式(9)所示。

```
Value declarations
d ::= ... ( * everything from O'Caml declaration * )
| let_abs x⟨a_i::κ⟩ = e ( * generic abstraction * )
```

(9)

在 Camlp5 工具中修改 sig_item 条目,增加用于分析“泛型抽象”的语句的分支语句。在该分支语句中三个核心字段:“let_abs”用于表示该函数使用“泛型抽象”形式定义;标识符 rec 表示该函数是否递归;let_binding_abs 条目用来分析函数名和主要“泛型抽象”的语法。let_binding_abs 条目又调用了 name_and_sig 条目、expr 条目。name_and_sig 条目是用来分析函数名和所带有的类型参数的。由于需要对签名环境做出改变,把函数名和类型参数放在一个条目中便于签名环境的更新。name_and_sig 条目中调用了 fun_name 条目、type_var 条目、type_sig_new 条目。分别用来分析函数名、参数类型、函数类型签名。具体条目之间调用关系如图 1 所示。

```
sig_item ::= "let_abs"; "rec"; let_binding_abs
    →<:sig_item< $_flag:r$ $_list:l$ >>
let_binding_abs ::= name_and_sig; expr
    →[ (<:patt< ··· >>,<:expr< ··· >>); ··· ]
name_and_sig ::= fun_name_abs;"<";type_var;">";
                 "{"; type_sig_new; "}"
    → (x1, x2, l2, signature_e)
    fun_name_abs ::= LIDENT; → (x1, l1)
```

图 1 Camlp5 中各条目的调用关系以及返回结果形式

let_binding_abs 条目对已有的函数信息进行形式化转化。首先把类型参数用 base_type 类型结构化表示。然后通过 type_arg 函数对类型参数进行转换。把原来 base_type 类型通过 Str 结构添加类型的根 Abs。如类型参数 f 变成(Abs f)形式。计算附属约束的 gapp 算法根据经过函数 type_arg 转化的类型参数和

函数的类型签名计算该函数的附属约束。然后把附属约束通过 expr_fun_def 函数以参数的形式写入 patt_Quotation 中。最后与 expr 条目返回的 expr_Quotation 组合成元组返回给 sig_item 条目的分支语句。

根据规则(e /tr-genapp)与(e /tr-gabs) 对 expr 进行修改。expr 条目中的分支语句如下所示:

```
x_1=LIDENT; '<'; l_1=type_argument; '>' →… ;
```

x_1 返回函数名,l_1 返回类型参数,该语句中调用了 type_argument 条目。该条目返回 argu_type 类型。首先检查(x1,"Abs")是否在签名环境中,如果在签名环境中,说明x_1 函数是"泛型抽象"函数。在l_1 前加上一个类型的根 Abs 即(Abs l_1)。然后按照普通的泛型函数表达式进行计算。

4　实验结果

通过修改和增加 Camlp5 中部分代码,在 O'Caml 中实现"泛型抽象"。由于加入的代码量比较大,文中就不一一列举。修改后的 Camlp5 源代码经过编译并重新安装。在新的编译环境下对部分"泛型抽象"函数进行测试。fequal 函数在新编译环境下测试代码如图2所示,编译结果如图3所示。

```
let_tif equal <a> = {equal<a> ::(equal<a>)=>a->a->a}
  typecase a of int --> fun a b-> a=b ;;
let_tif equal <a> = {equal<a> ::(equal<a>)=>a->a->a}
  typecase a of string --> fun a b-> a=b;;
let_tif equal <a> = {equal<a> ::(equal<a>)=>a->a->a}
  typecase a of unit -->
    ( fun a b -> match (a,b) with (Unit,Unit)->Unit) ;;
let_tif rec equal <a> = {equal<a> :: (equal<a>)=>a->a->
a}
  typecase a of ('a,'b) sum-->
( fun a b -> match (a,b) with
(Inl x,Inl y) -> Inl (equal <'a> x y )
| (Inr x,Inr y)-> Inr (equal <'b> x y));;
let_tif equal <a> = {equal<a> :: (equal<a>)=>a->a->a}
  typecase a of ('a,'b) prod -->
(fun (a,b) (c,d) ->((equal <'a> a c),((equal<'b>) b d
)));;
let_gf rec equal <a> = {equal<a> :: (equal<a>)=>a->a->
a}
  typecase a of (type 'a list = Non | Cons of 'a * ('a list))
--> (fun a b);;
let_abs fequal<r> = {fequal<l> :: (equal<l>)=>a->a->a}
  let equal <a> = (fun a b -> a=b) in  equal <(a) r>;;
let _ = equal<int list> ;;
let _ = fequal<list> (Cons (1,Non)) (Cons (1,Non))
```

图2　equal 函数的函数声明以及 fequal 函数声明

```
# val cp_equal_int : 'a -> 'a -> bool = <fun>
# val cp_equal_string : 'a -> 'a -> bool = <fun>
# val cp_equal_unit : unit -> unit -> unit = <fun>
# val cp_equal_sum : ('a -> 'b -> 'c) -> ('d -> 'e -> 'f) -
>('a, 'd) sum -> ('b, 'e) sum -> ('c, 'f) sum = <fun>
# val cp_equal_prod : ('a -> 'b -> 'c) -> ('d -> 'e -> 'f) -
>'a * 'd -> 'b * 'e -> 'c * 'f = <fun>
# val cp_equal_list :
('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list = <fun>
# - : '_a list -> '_a list -> bool list = <fun>
# val cp_fequal_abs : (('a -> 'a -> bool) -> 'b) -> 'b = <
fun>
# - : bool = true
```

图3　equal 函数以及 fequal 函数的编译结果

fequal 函数以一些较简单的类型为类型参数进行"泛型抽象"定义,实验结果应该使用更加复杂的函数和类型进行测试。因此,实验过程中对一些具有代表性的函数和复杂抽象类型进行编译测试。结果如表2所示。

表2　各个函数针对不同类型参数编译实验结果

	'a list	'a tree	('a,'b) tree	('a,'b,'c) tree
fsize	√	√	√	√
fequal	√	√	√	√
fadd	√	√	√	√
fmap	√	√	√	√

经过对编译器的测试,"泛型抽象"可以在 O'Caml 语言中实现。由图1中实例得,普通泛型函数 equal 的类型参数必须是确定的类型,如(int list)。而"泛型抽象"函数 fequal 的类型参数可以是抽象类型,如 list。由此看出,"泛型抽象"极大提高了泛型编程的灵活性。

5　结束语

泛型编程思想已经经过几十年的发展形成了比较成熟的理论体系,并且在 C++、Java、C#中得到实现与应用[9~12]。在 O'Caml 语言中泛型编程的研究尚处于起步阶段。文中对函数式语言 O'Caml 中的泛型编程作进一步探讨,在该语言的泛型编程中引入"泛型抽象"的概念,并用工具 Camlp5 对语法进行了相应扩展。"泛型抽象"使泛型函数定义可以以抽象类型作为类型参数,不仅提高了泛型编程的灵活性,而且使得 O'Caml 语言在实际应用中更加实用。当然目前工作只是一个初步探讨和基本实现。在此基础上还可以进行进一步的优化,使得泛型函数的语法更加简洁清晰,容易理解。

(下转第100页)

该软件产品经过两个月的实际测试后投入运行，在连续半年的工作中未出现任何软件失效现象，使相关可靠性研究在实际系统中进一步得到验证。

4 结束语

软件测试的目的是为了提高软件的可靠性，软件可靠性评价的结果又可以为软件测试服务。通过对软件测试及软件可靠性评价技术的研究，选取了适用的可靠性统计模型，并通过该模型计算出了某软件模块的可靠性以及软件的可靠性。该研究成果不仅能够充分利用这些测试结果，满足在不同阶段对软件可靠性进行评价的需要，而且有利于衡量软件测试工作的充分性。该模型具有简单、实用、易于工程化等特点，但该模型的适用范围及模型的一致性等问题有待进一步研究。

参考文献：

[1] 张 玲，袁 娜，马永刚，等. 基于测试用例和时间域软件可靠性模型[J]. 计算机技术与发展，2009，19(11)：167-170.

[2] 刘志方，钟德明，曾福萍，等. 软件可靠性测试的理论分析[J]. 测控技术，2008，27(10)：62-64.

[3] 程维虎，杨振海. 软件可靠性模型和估计[J]. 数理统计与管理，2010，29(1)：52-61.

[4] 艾 骏，陆民燕，阮 镰. 面向软件可靠性测试数据生成的剖面构造技术[J]. 计算机工程，2006，32(22)：7-9.

[5] 艾 骏，陆民燕，阮 镰. 实时嵌入式软件可靠性测试数据自动生成方法[J]. 测控技术，2007，26(3)：59-61.

[6] 蔡建平. 软件可靠性测试方法新探[J]. 计算机工程与设计，2009，30(20)：4658-4661.

[7] 吴玉美，阮 镰. 软件可靠性测试的加速机理研究[J]. 计算机应用，2006，26(6)：1449-1451.

[8] 吴玉美，陆民燕，阮 镰. 软件可靠性加速测试方法研究[J]. 计算机工程与应用，2006，42(8)：62-64.

[9] 李秋英，李海峰，徐 刚. 基于覆盖率信息的软件可靠性增长测试实践[J]. 计算机应用研究，2010，27(7)：2594-2597.

[10] Hu Q P，Dai Y S，Xie M，et al. Early software reliability prediction with extended ANN model[C]//Proceedings of the 30th Annual International Computer Software and Applications Conference. [s. l.]：IEEE Computer Society Press，2006：234-239.

[11] Hu Q P，Xie M，Ng S H，et al. Robust recurrent neural network modeling for software fault detection and correction prediction[J]. Reliability Engineering and System Safety，2007，92(3)：332-340.

[12] Xie M. Software Reliability Modeling[M]. Singapore：World Scientific Publisher，1991.

[13] Pham H. Software Reliability[M]. Singapore：Springer-Verlag，2000.

[14] Butschy B，Albeanu G，Boros D N，et al. Improving software reliability forecasting[J]. Microelectronics and Reliability，1997，37(6)：901-907.

[15] Guo J，Liu，H，Yang，X，et al. A software reliability time series growth model with Kalman filter[J]. WSEAS Transactions on Computers，2006，5(1)：1-7.

[16] 杨 波，黄洪钟，郭夙昌. 数据驱动的软件可靠性模型研究[J]. 中国科技论文在线，2007(10)：768-774.

[17] 楼俊钢，江建慧，帅春燕，等. 软件可靠性模型研究进展[J]. 计算机科学，2010，37(9)：13-27.

[18] 吴艳征，宋志强. 浅谈黑盒测试用例设计方法[J]. 科技信息(学术研究)，2008(16)：178-179.

[19] 蒋天乐，徐国治. 软件缺陷及软件可靠性技术[J]. 计算机仿真，2004，21(2)：141-144.

(上接第 95 页)

参考文献：

[1] Loh A. Exploring Generic Haskell[D]. Utrecht：Utrecht University，2004.

[2] Frisch A，Garrigue J，Rémy D，et al. The OCaml system release 4.00[M]. [s. l.]：Institut National deRecherche en Informatique et en Automatique，2012.

[3] Johan J，Andres L，Dave C. Dependency-style generic Haskell[J]. SIGPLAN Notices，2003，38(9)：12-13.

[4] Jansson P，Jeuring J T，Cabenda，et al. Testing properties of generic functions[R]. Utrecht：Utrecht University，2006.

[5] Hinze R. Polytypic programming with ease (extended abstract)[M]//Functional and Logic Programming. [s. l.]：[s. n.]，1999.

[6] 李 阳. OCaml 语言中泛型编程工具的研究与实现[D]. 南京：解放军理工大学，2012.

[7] de Rauglaudre D. Camlp5-Reference Manual[M]. [s. l.]：Institut National de Recherche en Informaticque et Automatique，2008.

[8] Holdermans S，Jeuring J，Loh A，et al. Generic views on data types[C]//Proc of International Conference on Mathematics of Program Construction. [s. l.]：[s. n.]，2006.

[9] 陈 林. 泛型程序重构技术研究[D]. 南京：东南大学，2009.

[10] 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)：13-16.

[11] 陈 林. 一种基于类型传播分析的泛型实例重构方法[J]. 软件学报，2009，20(10)：2617-2627.

[12] 陈 林. 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.

O'Caml泛型编程中“泛型抽象”的研究

作者： 王朋， 徐健， 于尚超， WANG Peng， XU Jian， YU Shang-chao
作者单位： 解放军理工大学指挥信息系统学院,江苏南京,210007

刊名： 计算机技术与发展 ISTIC
英文刊名： Computer Technology and Development
年，卷(期)： 2013,23(7)