基于 O'Caml 语言的通用编程技术①

李 阳, 赵建平, 张德华, 程小林

(中国卫星海上测控部, 江阴 214431)

摘 要: O'Caml 语言是一门优秀的函数式程序语言, 具有计算模型简单、语法语义描述清晰等特点, 而通用编程技术方法通过高度抽象算法、数据结构及其他软件组件可以避免功能相似代码的重复编写. 针对如何将通用编程技术方法和O'Caml 语言相结合来提高O'Caml 程序复用程度的问题, 通过对O'Caml 语言语法进行扩展引入类型标记1函数, 然后运用类型结构化转换和类型映射机制, 实现通用函数, 从而达到在O'Caml语言中引入通用编程技术的目的. 实例结果表明, 在 O'Caml 语言中实现通用编程技术, 有效提高了编程效率和程序的通用性.

关键词: 通用编程技术; 软件复用; O'Caml 语言; 语法扩展

Generic Programming Based on O'Caml Language

LI Yang, ZHAO Jian-Ping, ZHANG De-Hua, CHENG Xiao-Lin

(Satelite Maritime Tracking & Controlling Department of China, Jiangyin 214431, China)

Abstract: O'Caml is a popular functional programming language with a lot of advanced features, such as: simple module, clear grammar and semantci, etc. Generic programming which programs through the abstracting of algorithm, data structure and all other of the software components can avoid the repeated work of coding with similar functions. In order to combine the generic programming and O'Caml language which can improve the reuse of soft, we extend the grammar of O'Caml language to implement type-indexed function, the structure transition and isomorphism of type and generic function. The experimental result shows that the implementation of generic programming in O'Caml language has improved the efficiency and universality of programming.

Key words: generic programming; soft reuse; O'Caml programming language; grammar extension

1 概述

众所周知, 目前主流的程序设计方法包括面向过程程序设计方法和面向对象程序设计方法, 其中面向过程程序设计方法把数据和处理数据看作为两个分离的独立实体, 当数据结构改变时, 所有相关的处理过程都要进行相应修改, 程序复用性差. 而面向对象程序设计方法因为有了多态的特性可以使得具有继承关系的对象之间表现出不同数据类型或行为[1], 但程序复用程度依然不够. 为了进一步提高软件的复用程度, 研究人员提出了通用编程思想, 即通过抽象将算法、数据结构及其它一些软件组件在具体应用中的共性提取出来[2], 从而提高程序通用性, 增大程序适用范围.

本文以优秀的函数式程序设计语言 O'Caml 为平台对通用编程思想进行研究与验证, 具体包括首先在O'Caml 语言中引入通用程序的基础: 类型标记函数, 然后根据类型系统的结构化转换与映射特性, 以及通用函数的转换算法, 实现了函数针对不同类型定义的通用生成, 从而最后完成了 O'Caml 语言中通用编程技术的实现.

2 O'Caml语言与类型标记函数

2.1 O'Caml 语言简介

O'Caml 语言[3]是一种比较新颖而特殊的编程语言,它以λ演算(Lambda Calculus)作为理论基础,将计算

① 收稿时间:2014-08-21;收到修改稿时间:2014-10-08

机计算视为函数的计算, 具有更强的数学表达性, O'Caml 语言的语法描述简练、清晰、易理解、易维护, 程序编写灵活而应用广泛.

2.2 类型标记函数

为了在 O'Caml 语言上实现通用编程技术, 我们对其语法进行扩展来引入一种新的函数, 这种新函数最明显的特征就是表达式中带有显式类型参数. 下面举例说明新函数的形式:

$$\begin{aligned}\text{add}\langle\text{bool}\rangle &= \text{fun}\ (x,y)\to x \vee y\\ \text{add}\langle\text{int}\rangle &= \text{fun}\ (x,y)\to x+y\\ \text{add}\langle\text{string}\rangle &= \text{fun}\ (x,y)\to x\ \hat{}\ y\end{aligned},$$

由上可见, add 函数是定义在 bool, int, string 三个类型上的加函数, 可以看出等号左边的表示形式与普通函数不太相同, 它由函数名 add、“〈 〉”和一个类型名组成, 这种形式的函数称为类型标记函数, 括号中的类型名为类型标记函数的类型参数.

引入类型标记函数需要扩展 O'Caml 语言的语法, 类型标记函数的语法如图 1 所示.

Value declarations

$d ::= x\langle a\rangle = \text{typecase}\ a\ \text{of}\ \{P_i \to e_i\}^{i\in 1..n}$ 类型标记函数声明

Expressions

$e ::= x\langle A\rangle$ 类型标记函数表达式

Type patterns

$P ::= T\{\alpha_i\}^{i\in 1..n}$ 带参的数据类型

Type arguments

$A ::= T$ 数据类型

$\mid \alpha,\beta,\gamma,...$ 附属变量

$\mid (A_1\ A_2)$ 类型应用

图 1 类型标记函数对应语法

类型标记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 add 函数针对类型'a list 的定义为 add<`a list> = fun(x,y) →if (list.length x) = (list.length y) then list.map(add<`a>) (zip(x,y)) else raise “args must have same length!”, 通过定义可以实现针对列表的加操作. 根据定义可知, 函数 add<`a list>调用了函数 add<`a>来具体完成两个列表对应元素的加操作, 而后者是 add 针对变量`a 的类型标记函数, 类型标记函数之间的这种调用关系称为附属[4], 被调用的函数称为原函数的附属函数.

类型标记函数的附属信息直观反映在函数的类型中, 例如函数 add<`a list>的类型为 add<`a list>::(add<`a>::'a→'a→'a)=> 'a list→'a list→'a list, 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 list>的类型才能为'a list→'a list→'a list. 双箭头左边的小括号里面的内容称为附属约束[5], 它是附属函数针对原函数类型变量的类型签名, 类型参数中的类型变量称为附属变量.

2.3 类型标记函数转换

我们的目的是将类型标记函数[6]引入到 O'Caml 语言中, 但是单纯根据前面介绍的类型标记函数语法写出的程序语句不能通过 O'Caml 语言编译器的编译, 因此我们需要一些转换规则将类型标记函数的程序语句转换成符合 O'Caml 语法的程序语句.

$$\frac{\begin{array}{c}[\![d_{\text{tif}}]\!]_i^{\text{tif}} \equiv \{d\}_i^{i\in 1..n} \sim \Sigma_2\\ \left\{d_i \equiv \text{cp\_x\_}T = [\![e_i]\!]^{\text{tf}}\right\}^{i\in 1..n}\end{array}}{[\![x\langle a\rangle = \text{typecase}\ a\ \text{of}\ \{T_i\to e_i\}_i^{i\in 1..n}]\!]^{\text{ttf}} \equiv \{d_i\}_i^{i\in 1..n} \sim \{x\langle T_i\rangle\}^{i\in 1..n}}\ (\text{tr-tif})$$

$$\frac{[\![e_{\text{tif}}]\!]_\Sigma^{\text{tf}} \equiv e}{}$$

$$\frac{x\langle T\rangle \in \Sigma}{[\![x\langle T\rangle]\!]_\Sigma^{\text{tif}} \equiv \text{cp\_x\_}T}\ (\text{tr}-\text{named})\qquad \frac{}{[\![x\langle\alpha\rangle]\!]^{\text{tif}} \equiv \text{cp\_x\_}\alpha}\ (\text{tr}-\text{dep var})$$

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

图 2 类型标记函数声明和表达式转换规则

首先是声明转换规则, 如图 2 所示, 规则对每个类型标记函数声明进行转换使其通过 O'Caml 语言编译器的编译. 声明中包含有函数的类型签名, 它提供了函数的附属信息, 这些信息在转换中被用到.

然后是表达式转换规则, 例如类型标记函数形式为 X<A>, 如果类型参数是一个已定义的基本类型或者是一个附属变量, 那么调用规则(tr-named)和(tr-depvar)可以直观的得出合适的组件. 如果类型参数 A 是一个类型应用, 例如函数 X< A1 A2>, 其中 A1 是 A2 的参数, 这种情况下将调用规则(tr-app)对其进行转换, 转换结果由函数 X 针对 A_2 生成的组件和 X 附属函数 Y 针对 A_1 生成的组件共同构成.

在扩展后的 O'Caml 语言中, 类型标记函数的类型都是限制类型, 因此也需要将类型标记函数的限制类型转换成普通类型, 例如函数 add<`a list>的限制类

型是(add<`a>::'a→'a→'a)=> 'a list→'a list→'a list, 转换之后将变成类型'a→'a→'a→'a list→'a list→'a list, 转换后的类型是一个普通高阶函数的类型, 里面没有附属函数和附属变量的信息.

3 基于O'caml语言的通用编程实现

3.1 类型结构化转换

图 3 展示了 O'Caml 语言中类型的声明语法, 其中 name 是类型名, 'a 是 name 的类型参数变量, 等号后面是声明类型的表达式. 这里我们定义三个新类型: unit、sum 和 prod, 类型声明如下所示,

```
type unit = Unit
type ('a, 'b) sum = Inl of a | Inr of b
type ('a, 'b) prod = 'a * 'b
```

类型 unit 是单元类型, 类型 sum 是一个选择类型, 类型 prod 是一个元组类型. 然后根据类型声明的结构特点, 我们可以使用类型 sum、prod 和 unit 对类型的表达式进行描述. 将类型表达式转换成由类型 sum、prod 和 unit 组成的表达式的过程称为类型的结构化转换. 例如, 二叉树的类型声明为: type 'a tree = Leaf | Node of ('a tree)*'a*('a tree), 对它进行类型结构化转换得到的结果为 type 'a str_tree = (unit,('a tree,('a.'a tree)prod)prod)sum, str_tree.

```
type 'a name = …….          类型声明
    | Name_i ……
    | Name_j of t_j ……
    | Name_k of t_k *……*t_l.....;;
```

图 3　类型声明语法

3.2 类型映射

所有类型与它们的结构化转换类型都是同构的, 可以使用映射函数组合 ep_T 来表示它们的同构特性[6]. 例如, 类型 T 和它的结构化转换类型 str_T 是同构的, 那么存在映射函数组合 ep_T = EP (from, go), ep_T 的类型为: ep_T:: ({'a_i}$^{i\in 1..n}$ T, {'a_i}$^{i\in 1..n}$ str_T)ep, from 和 go 是对称函数, 类型为: from:: {'a_i}$^{i\in 1..n}$ T→ {'a_i}$^{i\in 1..n}$ str_T 和 go:: {'a_i}$^{i\in 1..n}$ str_T→ {'a_i}$^{i\in 1..n}$ T, 因此二叉树类型 tree 的映射函数组合的定义如下所示.

```
ep_tree::('a tree, 'a str_tree) ep
let ep_tree = let fromtree Leaf = Inl Unit and fromtree (Node (1, x, r))=Inr(1, (x, r)) and gotree (Inl Unit)= Leaf and gotree (Inr(l,(x,r)))= Node (l, x, r) in
EP(fromtree, gotree);
```

3.3 通用函数定义

通用函数可以为那些不存在于函数签名环境的类型生成定义, 这是因为通用函数可以导出类型标记函数针对类型的函数声明. 前面描述类型标记函数的时候声明了 add 函数针对类型 int、bool 和 string 的定义, 下面介绍如何通过推导的手段得到函数 add 针对 tree 类型的定义.

首先声明函数 add 针对类型 unit、prod 和 sum 的定义, 然后我们写出函数 add〈`a tree〉定义的公式: add〈`a tree〉x y = add〈`a str_tree〉(from ep_tree x) (from ep_tree y), 该公式经过转换得到的结果如下所示, 根据转换结果可知欲得组件 cp_add_tree

$$\text{let\_tif rec cp\_add\_tree} = \text{fun cp\_add\_`a} \rightarrow \left(\text{fun } (x, y) \rightarrow \text{go ep\_tree} \begin{pmatrix} \text{cp\_add\_str\_tree} & \text{cp\_add\_`a} \\ (\text{from ep\_tree x}) & (\text{from ep\_tree y}) \end{pmatrix}\right)$$

的定义, 就必须先定义组件 cp_add_str_tree, 而函数 add〈`a str_tree〉经过转换定义如下所示, 根据

$$\text{let\_tif rec cp\_add\_str\_tree} = \text{fun cp\_add\_`a} \rightarrow \text{cp\_add\_sum cp\_add\_unit}\begin{pmatrix} \text{cp\_add\_prod } (\text{cp\_add\_tree cp\_add\_`a}) \\ (\text{cp\_add\_prod cp\_add\_`a } (\text{cp\_add\_tree cp\_add\_`a})) \end{pmatrix}$$

定义可知函数 cp_add_str_tree 和 cp_add_tree 是递归调用的, 又由于 cp_add_unit、cp_add_sum 和 cp_add_prod 的定义已经存在, 因此可以得到 cp_add_tree 的定义.

4 实验验证

4.1 扩展 O'Caml 语言语法

本节我们使用语法修改工具[8]对 O'caml 语言语法进行扩展. 利用语法修改工具修改 O'Caml 的抽象语法树可以定义任何新的语法

4.1.1 类型标记函数语法实现

图 5 展示了部分实现类型标记函数的语法条目, 标识符"let_tif"作为类型标记函数声明语句的关键字, 然后定义了语法条目 let_binding_tif 对类型标记函数的声明语句进行语法分析和处理. 语法条目 expr 负责对表达式的局部声明语句和函数应用进行分析处理. 语法条目 type_sig 负责对类型标记函数中的函数类型签名进行分析处理.

4.1.2 类型结构化转换与类型映射语法实现

图 6 展示了部分实现类型结构化转换和类型映射的语法条目, 标识符“type_str”作为结构化转换类型声明的关键字, 然后定义了语法条目 type_declaration_str 用于分析类型声明语句, 得到的类型表达式由类型 unit、sum 和 prod 构成. 标识符“let_type”作为类型映射函数声明语法条目的关键字, 语法条目 let_binding_type 对类型声明语句进行语法分析, 由于篇幅关系没有展示所有相关的语法条目.

```
str_item:
  [ ["let_tif";r=V(FLAG "rec");l=V (LIST1 let_binding_tif SEP "and")
      -> match l with
          [ <:vala< [(p, e)]>>  ->
            match p with
              [ <:patt< _ >> -> <:str_item< $exp:e$ >>
              | _ -> <:str_item< value $_flag:r$ $_list:l$ >> ]
        | _ -> <:str_item< value $_flag:r$ $_list:l$ >> ]   ] ] ;

let_binding_tif:
  [ [x1 =func_name; "<"; x2 =type_var; ">";"=";"{";x3=type_sig ;"}";
     "typecase";x4=type_var;"of";x5=type_case_type;"-->";x6=expr ->
      ... ... ] ];

expr:
  [ ["let_tif";o=V(FLAG "rec");l=V(LIST1 let_binding_tif SEP "and");
     "in"; x=expr LEVEL "top"->
      <:expr< let $_flag:o$ $_list:l$ in $x$ >>

   | x1 = func_name ; "<";l1= type_argument ;">"-> ... ... ] ];

type_case_type: [[x1 = type_sig_type ->... ... ]];

type_sig:
  [ [x1=type_sig_name;"<";x2=type_var;">";"::";"(";x3= type_sig_dep;")";
     "=>";x4=LIST0 type_sig_type ->
     let l= (type_signature (x1,x2,x3,x4)) in ((x1,x2,x3,x4),l)]];
```

图 5 类型标记函数语法条目

```
str_item:
   [ [ "type_str"; tdl = V (LIST1 type_declaration_str SEP "and") ->
       <:str_item< type $_list:tdl$ >> ] ]

type_declaration_str:
   [ [ tpl = type_parameters; n =type_patt; "="; tk = type_kind_str;
       cl = V (LIST0 constrain) ->
          let n2 = (loc, <:vala< ("str"^"_"^(snd n)) >>) in
        {MLast.tdNam = n2; MLast.tdPrm = <:vala< tpl >>;
         MLast.tdPrv = pf; MLast.tdDef = tk; MLast.tdCon = cl} ] ]

str_item:
   [ ["let_type";r=V(FLAG "rec");l=V(LIST1 let_binding_type SEP "and")
      -> match l with
          [ <:vala< [(p, e)]>>  ->
            match p with
              [ <:patt< _ >> -> <:str_item< $exp:e$ >>
              | _ -> <:str_item< value $_flag:r$ $_list:l$ >> ]
        | _ -> <:str_item< value $_flag:r$ $_list:l$ >> ]   ] ]

let_binding_type:
   [ [r=V(FLAG "rec");"type"; tpl = type_parameters; n = type_patt;"=";
      tk = type_kind_type; cl = V (LIST0 constrain)  -> ... ... ] ]
```

图 6 类型结构化转换与类型映射语法条目

4.2 实验结果验证

4.2.1 通用函数 add

本文在前面章节介绍通用编程技术的时候已经调用了 add 函数作为示例, 下面将展示 O’Caml 语言中通用函数 add 的声明实现过程, 首先是定义通用函数需要的一些声明, 包括类型 prod、sum 和 unit 的声明, add 函数针对基本类型 bool、int 和 string 的声明以及 add 函数针对类型 prod、sum 和 unit 的声明, 相关定义见前文.

声明了类型 unit、sum 和 prod 和 add 函数基本定义之后, 就可以利用转换规则生成通用函数, 图 7 展示了 add 函数针对类型 tree 的声明导出过程. 图 8 展示了通用函数 add 针对类型 tree 的运算.

```
type ('a,'b) tree= Leaf | Node of (('a,'b) tree)*('a*'b)* (('a,'b) tree);;

type_str ('a,'b) tree= Leaf | Node of (('a,'b) tree)*('a*'b)* (('a,'b) tree);;

let_type type ('a,'b) tree= Leaf|Node of(('a,'b) tree)*('a*'b)*(('a,'b) tree);;

let_tif rec add <a> =(add<a>::(add<a>)=>'a->'a->'a) typecase a of(`a,`b)tree-->
fun x y->(go ep_tree)(add<str_tree>add<`a>add<`b>(from ep_tree x)(from ep_tree y))
and add <a> =(add<a> :: (add<a>)=>'a->'a->'a) typecase a of (`a,`b) str_tree-->
   add<(unit,((`a,`b) tree,((`a,`b) prod,(`a,`b) tree) prod) prod) sum)>;;
```

图 7 类型 tree 的结构化转换和映射以及 add〈tree〉的定义

```
let _=add<((int,string)tree)>
        (Node((Node(Leaf,(1,"a"),Leaf)),(2,"b"),(Node(Leaf,(3,"c"),Leaf))))
        (Node((Node(Leaf,(3,"c"),Leaf)),(2,"b"),(Node(Leaf,(1,"a"),Leaf))));;

#   - : (int, string) tree =
        Node (Node (Leaf,(4,"ac"),Leaf),(4,"bb"), Node(Leaf,(4,"ca"),Leaf));;


let _=add<(int cons_list)> (Cons(1,(Cons(2,None)))) (Cons(1,(Cons(2,None))));;

# - : int list = Cons (2, Cons (4, None));;
```

图 8 通用函数 add 的运算

可以看出, 没有利用通用技术编写的 add 函数针对不同数据类型进行加操作的定义需要针对这些数据类型重复编写加操作的定义程序, 而利用通用技术编写的 add 函数只需将不同的类型结构带入 add 函数的通用定义, 即可实现针对这些类型的加操作, 统计得出后者代码量相对前者减少了 55%.

4.2.2 通用多态模式匹配

模式匹配在类型推导和逻辑运算以及类型检查中都有着重要的作用[9], 但是针对不同类型的数据(项), 往往要重新定义模式匹配函数, 从而提高了程序员的工作量和程序复杂度, 为了提高模式匹配函数的通用性, 我们引入通用编程技术来生成通用模式匹配函数.

模式匹配用于比较合一时, 项通常可以定义为变量或者构造子针对其他项的应用, 项的类型可以归纳为 T::=v|c(T_1…T_n), v∈Var, c∈Con, Var 表示变量的集合, Con 表示构造子的集合, 模式匹配中处理项的核心函数主要有 3 个: children、varcheck 和 topeq, children 函数用于提取项中所包含子项, 函数 varcheck 用于检查项是否是变量, 如果是则输出该变量, 函数 topeq 用

于对两个项的最外层进行比较, 以作为匹配子项的条件. 上述三个函数针对不同的类型需要程序员提供不同的定义, 而引入通用编程技术可以将上述函数声明为通用函数.

```
type 'a tree = Vt of var | Node of 'a tree * 'a *('a tree) ;;
type_str 'a tree = Vt of var  | Node of 'a tree * 'a *('a tree) ;;
let_type type 'a tree = Vt of var  | Node of 'a tree * 'a *('a tree)

type ('a ,'b) treef = FVt of var  | FNode of 'b * 'a *('b) ;;
type_str ('a ,'b) treef = FVt of var  | FNode of 'b * 'a *('b) ;;
let_type type ('a ,'b) treef = FVt of var  | FNode of 'b * 'a *('b) ↵

let varcheck_tree  x = varcheck<(`a tree)> x;;

let children_tree  x =
  let_local fflatten<`a>={fflatten<a|b>::(fflatten<a|b>)=>'a->'b}(fun x ->x)in
  let_local map<`a>={map<(a,b)>::(map<(a,b)>)=>'a->'b} (fun x->None) in
  let_local map<`b>={map<(a,b)>::(map<(a,b)>)=>'a->'b} (fun x->Cons(x,None))in
  (fflatten<((`a,`b)treef)>) ((map<((`a,`b)treef)>) (fixtree x) );;

let topeq_tree x y=
  let_local fequal<`a>={fequal<a|bool>::(fequal<a|bool>)=>'a->'a->bool}
  (fun a b->a=b)in
  let_local fequal<`b>={fequal<a|bool>::(fequal<a|bool>)=>'a->'a->bool}
  (fun a b->true)in
  fequal<((`a,`b)treef)> (fixtree x) (fixtree y);;
```

图 9　类型 tree 声明以及通用函数的定义

图 9 声明了类型 tree, 并对它进行了结构化转换和类型映射, 然后根据需要得到了函数 varcheck、children 和 topeq 针对类型 tree 的通用定义, 函数中调用了其他部分类型映射函数, 这些函数的通用定义因为篇幅在这里没有详细展示. 图 10 展示了模式匹配函数针对类型 int tree 的运算结果.

```
let _= unify [] (Vt "x", Node(Vt "x1",1,Vt "x2"));;

# - : (var * int tree) list maybe = Just [("x",Node (Vt "x1", 1, Vt "x2"))];;

let _= unify [("x", Node (Vt "x1", 1, Vt "x2"))]
  (Node(Vt"x",1,Vt "y"),Node(Node(Vt"x3",1,Vt"x4"),1,Node(Vt"y1",3,Vt"y2")))

# - : (var * int tree) list maybe =
  Just[("x3", Vt "x1");("x4", Vt "x2");
       ("x", Node (Vt "x3", 1, Vt "x4"));("y",Node (Vt "y1", 3, Vt "y2"));;]
```

图 10　模式匹配函数针对类型 int tree 的运算结果

上述运算结果证实了利用通用函数可以得到通用的模式匹配函数, 而对本实验结果进行统计得出利用通用编程技术完成模式匹配操作的代码量只有未利用通用编程技术代码量的 40%, 从而表明通用编程技术的应用能够大大提高程序编写效率.

5　结语

上述实验中的函数定义都是使用 O'Caml 语言完成的, 从而实现了在 O'Caml 语言中的通用编程技术的扩展. 实验中可以看出利用通用编程技术定义通用函数可以避免针对不同类型的数据结构重复定义功能相似的代码, 从而大大提高了编程的效率, 提高了程序通用性.

参考文献

1 吴拥民.泛型设计的理论研究.闽江学院学报,2006,27(2): 72–76.

2 陈林,泛型程序重构技术研究[学位论文].南京:东南大学,2009.

3 Wadler P. The essence of functional programming. An invited talk at 19th Annual Symposium on Principles of Programming Languages, Albuquerque, New Mexico. 2009: 107–113.

4 Löh A, Clarke D, Jeuring J. Dependency–style generic Haskell. Proc. of the 8th ACM SIGPLAN International Conference on Functional Programming. ACM Press. 2005. 141–152.

5 Wadler PS. Blott. How to make ad–hoc polymorphism less ad–hoc. Proc. of the 16th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, 2008: 60–76.

6 Ehrig H. Applied and computational category theory. European Association for Theoretical Computer Science, 2006, (6): 134–135.

7 Oliveira BC, Gibbons J. Typecase: A design pattern for type–indexed functions. Proc. of the 2005 ACM SIGPLAN workshop on Haskell, Tallinn, Estonia, 2007, (9): 98–109.

8 Rauglaudre D. Camlp5–Reference Manual version 5.09. http://cristal.inria.fr/ddr/camlp5/doc/pdf/camlp5–5.09.pdf.

9 Knight K. Unification: A multidisciplinary survey. Computing Surveys, 2007, 21(1): 93–124.