Function Templates for the Synthesis of Functional Programs

Natela Archvadze
Department of Computer Sciences
Ivane Javakhishvili Tbilisi State University
Tbilisi, GEORGIA

Merab Pkhovelishvili
Niko Muskhelishvili Computing Mathematic Institute
of Georgian Technical University
Tbilisi GEORGIA

Otari Ioseliani
Georgian – American University
Natural Sciences and Engineering School
Email: Otari.ioseliani {at} gmail.com

Lia Shetsiruli
Shota Rustaveli State University, Batumi, Georgia

Abstract—**One of the main reason for emergence of the direction, called synthesis of programs is to increase requirements for reliability of the software, as constructed programs by a synthesizer do not demand the verification process. The problem of synthesis is of interest to researches of artificial intelligence, and also to software engineering.**

The problem of automatic synthesis of programs for the given structures of data is generally bound to a problem of construction and after-processing of dynamic structures of data. These problems are solved generally by means of the functional programming, because in other paradigms it is very difficult to construct "the main recursive" part of the program (a set of functions) for processing of the designed structures of data. In this work templates of the functional languages - Lisp and Haskell - are considered for a recursive function which are presented from a tail recursion, recursions on the head of the list and with parameter - with accumulators.

***Index Terms*— the Functional languages, recursive functions, function templates**

I. Introduction

Problem of synthesis, which is one of the most composite problem in the field of programming, is possible to present as development of a method of automatic generation of the program by the computer for a task, which was not solved earlier by it and for which it has no algorithm of solution [1,2].

The use of techniques of constructing of data structures in a paradigm of the functional programming allows also to create in parallel the standard templates (frameworks) of functions for processing of these structures. Such frameworks can be considered as templates for filling by necessary functionality. The general form of such templates remains invariable, only the content is changing, bound to requirements to functions, which are defined according to purposes of the developer.

In this work are considered the examples of the functional languages Lisp and Haskell.

II. Properties of Function in the Functional Languages

To the common properties of functions in the functional programming languages belongs purity (lack of the ghost effects, determinancy), lazy evaluationand opportunity to make partial calculations.

Function has property of purity if it can operate only the memory marked out for it, without modifying memory out of its area. The function which output value depends only on values of input data is called determined.

If at identical input datavalues in various calls function can return various values, in the functional paradigm are determined. Lazy strategy of calculations is that the NOT function makes calculations until their result will not be necessary in program work. So values of data-ins are never calculated if they are not required in a function body. It allows to create potentially the infinite structures of data (lists, trees etc.) which are limited to only the physical extent of computer memory. Such infinite structures can quite be processed in the lazy way as only those elements which are necessary for work are calculated in them. Transfer on an entrance of any function of the infinite list does not attract program cycling as it does not calculate all this list entirely (that would be impossible).

In the functional programming languages is accepted that function have a type and the type of function is carried, that is has such appearance: *A1 -> (A2 ->... (An ->B) ...),*where *A1, A2, ...An*— types of input data, and B—typeof result. The carried functions means that such functions accept input data on the single, and as a result of such individual application new function turns out.

III. Ways of the Description of Dynamic Structures of Data

Traditionally the functional programming was engaged in studying and processing of such structure of data processing, as a list. The first functional language Lisp is called so as"list processing".

For representation and processing such structures of data, as treesvarious nature (binary, balanced and other), vectors andmatrixes, various difficult displays, the method of syntactic oriented constructioningis applied (was offered by Ch. Hoare) [3] . This method consists in constructing of types of data from other types (including recursively from itself) by means of application of two prime operations — the Cartesian product (*) and tagged union (+). For example, the list of elements of type A can be presented as follows:

List(A)=NIL+(A x List(A))

The use of technique of constructing data structures in a paradigm of the functional programming allows also to create in parallel the standard templates (frameworks) of functions for processing of these structures. Such frameworks can be considered as templates for filling by necessary functionality. The general form of such templates remains invariable, only the content is changing, bound to requirements to functions, which are defined according to purposes of the developer.

IV. Recursive function templates in Lisp

Let's consider Lisp's templates, which are called as "the generalized forms of representation" and are used for Lisp's recursive functions. These forms are consideredin details in [4].

Thay are used for such functions where the tail recursion is applied:

```
(DEFINE FUN(F F0.L)
   (COND((MEMBER NIL L)a)
          (T(g(f(M F L))
             (APPLY 'FUN(CONS 'F(CONS 'F0(M F0 L)))))
))
```

For such functions, where the recursion comes to the list head, the following form was thought up on Lisp:

```
(DEFINE LIST21(a g f F F0.L)
      (COND((MEMBER NIL L)a)
            (T(LISTN21(APPLY* g(APPLY f(M F L))a)
                       gfFF0.(MF0 L)))))
where
(DEFINE M(F L)
      (AND L(CONS(APPLY* (CAR F)(CAR L))(M(CDR
F)(CDR L)))
```

V. Representations of recursive functions in Haskell

Type"List of elements of typeA"

Further it is meant, that data structure as the list of elements of some type Ais presented within a method of syntactic oriented constructioning as follows [3]:

*List(A) = NIL + (A * List(A)).*
prefix=constructor List(A)
head,tail=selector List(A)
isNil, isNonNil=predicate List(A)
nil, nonNil=parts List(A)

Each function for processing of *List (A)* values has to support at least two patterns. The first processes *NIL*, the second — *nonNIL*. To these two *List (A)* parts in the Haskell language usually there correspond exemplars *[]* and *(x:xs)*. Two patterns can be united in one with use of technology of protection. In a body of the second pattern (or the second expression of protection) xs element processing (or tail) is carried out by the same function.

Templates for functions with a tail recursion and the recursion on the list head

It is possible to copy the Lisp-and forms defined by us and for Haskell-I and to offer as a standard template for the function processing lists. The example considers the function receiving on an entrance one argument list [5,6]:

1 . tail recursion

```
f [ ] = g1 [ ]
f ( x : xs ) = g2 ( g3 x ) ( g4 ( f ( g5 xs ) ) )
```

2 . The recursion comes to the list head

```
f [ ] = g1 [ ]
f ( x : xs ) = g2 ( f ( g3 x ) ) ( g4 (g5  xs ) )
```

The *g1, g2, g3, g4* and *g5* functions depend on the purposes of developers.

g1-function for processing of the empty list;

g2-function for a hitch of results of processing of the head and the rest of the nonempty list;

g3-function for head processing nonempty list;

g4-function for processing of result of a recursive call for the rest of the nonempty list;

g5-function for a pretreatment of the rest of the nonempty list before recursive a call.

Example 1: For example, for the length function calculating length of the given list, whose definition looks so:

```
length :: [a] -> Int
length  []  = 0
length L = 1 + length (tail  L)
```

g1 is the constant returning value 0. The *g2* function is addition operation +. The *g3* function is the constant returning value 1. And the *g4* and *g5* functions are identifiers, that is returning the arguments transferred to an entrance. Thus, it is possible to write length function definition so:

```
gl _ = 0
g2 a b = a + b
g3 _ = 1
g4 x = x
g5 x = x

length [] = gl []
length (x:xs) = g2 (g3 x) (g4 (length (g5 xs))
```

Example 2: function returns a posledny element of argument list, e.g. calculation of last [1,2,3,4] yields result 4.

```
            last  :: [a] -> a
    last []         =  error « empty list»
            last [x]        =  x
        last (_:xs)     =  last xs
```

```
gl _ = error
g2 a b = b
g3 x = x
g4 x = x
g5 x = x
```

g1 Is the error function. The *g2* function returns the second argument. The functions *g4, g5* and g6 are identifiers, which are returning the arguments transferred on input. Therefore, it's possible to write last function definition so:

```
f [] = gl []
f (x:xs) = g2 (g3 x) (g4 (f (g5 xs)))
```

Example 3: the function last, argument of this function is the list from lists, values are the last members of each element, e.g. last' [[1,2,3],[2,4],[3,5,7,9]] gives the list [3,4,9].

```
        last'  :: [[a]] ->[ a]
      last' [x]        = (last x):[]
    last'(x:xs)     =  (last x) :last' xs
```

gl is the function-constructor. The *g2* function returns the last element of the head. And the *g4, g5* and *g6* functions are identifiers, therefore returning the arguments transferred to an input.

Templates for functions with accumulators

There are defined several ways of representation of functions In Haskell. We will consider functions with collecting parameters (accumulators).

Sometimes is happening that during execution of function, seriously becomes actual the problem of memory.

The recursion is very recourse-intensive way of the organization of computing processes which demands large expenses of memory than simple iterations therefore within a paradigm of the functional programming there is very often an exclusive problem of an expense of memory. This problem can be explained on the example of the function, given number factorial calculation:

Example 4:

```
factirial 0=1
factorial n=n*factorial (n-1)
```

If review an example of calculation of this function with argument 3, it will be possible to see the following sequence:

```
factorial 3
==>3*factorial 2
==>3*2*factorial 1
==>3*2*1*factorial 0
==>3*2*1*1
==>3*2*1
==>3*2
==>6
```

On the example of these calculations visually seen that during recursive calls of functions the memory is using hard as for storage of intermediate results of calculations and on storage of addresses of return from the enclosed recursive calls as well. In this case the quantity of memory is proportional to value of argument, but arguments can be in a larger number, and also calculations can be much more difficult.

In such cases it is possible can be used the accumulator or the store accumulating parameters.

Example 5: For this purpose it is possible to review an example of function a factorial calculation with the accumulator:

```
    factorial_acc n = fun n 1
         fun  0a = a
      fun n a=fun(n-1)(n*a)
```

The second parameter of function carries out a role of accumulating parameter. It contains the result which is being returned after the end of recursion. It contains the result which is being returned after the end of recursion. In this case recursion becomes tail recursion, the memory is spending for storage addresses an returns function values.

The other example – function definition using accumulate parameter, which is calculating average arrhythmic value of list members:

Example 6:

```
funl = fun' l 0 0
fun' []  s n =s/n
fun' (x:xs)  s n =fun' xs (x+s) (n+1)
```

The "fun" function has two accumulators: first is to summarize the elements of list which has been set and the second one is the list – elements counting. The result of function is the division of these values. The tail recursion is the special case of recursion which contains only one call of the recursive function, thus this call is being performed after all of the calculations.

During performing calculations of tail recursion, can be performed by the iteration in constant in memory "size". On practice it means that the "good "translator of functional language should be able to recognize the recursion and implement it as cycle. The method of the accumulating parameter not always leads to a tile recursion, however it unambiguously helps to reduce the total amount of memory.

VI. Main Principls of Creation of Functions with the Accumulating Parameter

It's possible to determine main principals of construction the definitions with accumulated parameter [7]

1. It's being entered new function with additional argument (accumulator) in which are collected the results of calculations.
2. The initial value of accumulated arguments is being set in equality which connects old and new functions.
3. These equalities of an assumed function which correspond exiting are being changed by accumulator returning.
4. The equalities corresponding recursive definition appearing like a call to a new function in which the accumulator gets value which is returned by main function.

It's impossible to generate each function for calculating with accumulator. Construction of function by accumulated parameter, this method is not universal, and doesn't guarantee receiving a tail recursion; on the other hand, creation of definition with accumulating parameters is a matter of creativity.

In [6] we presented the generalized forms for functions with tail recursion for Haskell. Let's represent similar generalized forms for functions with padding argument (accumulator). funn =fun' na - - call of this function parameters n and a have a specific values.

fun'n a= g1 a
fun' (x : xs) = g2 (g3 x)(g4 (fun' (g5 xs) g6 a))

g1 is a function which accepts that value which is returned by the initial function;

g2 is a function which connects two of results of list's head an tile processing;

g3 is a function which is processing the head of the list;

g4 is a function which is processing recursive call for list's tail;

g5 is the function which tentatively is processing recursive call for list's tail ;

g6 is a function which is processing the accumulator.

Example 7: in such a representation of recursive functions *Factorial_A* is build function with accumulator *F* and arguments *N, 1*

(a=1). g5 - is the function of argument by unit. *g6*- is function of multiplication and *g2, g3* an *g4* – are identifiers which are returning the arguments given on input. Therefore the definition of factorial function can be written like:

g1 _=1
g2 x=x
g3 x=x
g4 x=x
g5 x=x-1
*g6 x y=x*y*

The technique of automatic creation of recursive forms or function templates for processing data-structures was created within area of functional programming because depends only on dynamic data structure construction, which is using only in functional programming.

VII. Conclusion

We provided regularity proofs for the generalized forms of recursive functions [8]. The generalized forms for recursive function and for the functions with accumulator-parameter are using in resolving the problems of verification of functional programs [9, 10]. These forms will be used during synthesis of functional programs particularly for synthesis Haskell programs in the future

References

[1] N.Archvadze, M.Pkhovelishvili, L.Shetsiruli. The complexity of program synthesis from examples. Proceedings of the Eleventh International Conference Pattern Recognition and Informaton Processing (PRIP'2011). ISNB 978-985-448-772-7. pp. 275-279. http://lsi.bas-net.by/conferences/prip2011/ . 2011.

[2] Archvadze N.N., Pkhovelishvili M.G., Shetsiruli L.D. Several issues of programs synthesis. Proceedings of the International Conference on System Analysis and Information Technologies. ISSN 2075-4086. pp. 403. http://sait.kpi.ua/books/sait2011.ebook.pdf/view 2011.

[3] Doushkin R.B. Functionalnoe programirovanie na Haskell. 2007.

[4] N. Archvadze,M. Pkhovelishvili, L. Shetsiruli, M. Nizharadze. Program Recursive Forms and Programming Automatization for Functional Languages. WSEAS TRANSACTIONS on COMPUTERS. Volume 8, pp. 1256-1265, ISSN: 1109-2750 http://www.wseas.us/e-library/transactions/computers/2009/29-531.pdf. 2009

[5] N.Archvadze, M.Pkhovelishvili, L.Shetsiruli . Automatically building the "basic recursive" part of the data structures programs descriptions. Proceedings of the System Analysis and Information Technologies 14-th International Conference SAIT 2012. p.323. http://sait.kpi.ua/books/sait2012.ebook.pdf/view

[6] N. Archvadze. M. Nizharadze. Typical Template Verification for List Editing In Haskell Language. Proceedings of the International Conference Management systems and modern information technologies. pp 170–172. ISSN 1512-3979.

[7] Graham Hutton. Programming in Haskell. Cambridge, 2007.

[8] N. Archvadze, M. Pkhovelishvili, L.Shetsiruli. Construction of the Generalized Recursive Forms for Functional Languages and their Application Verification of. Electronic Scientific Journal: "Computer Sciences and Telecommunications". No. 3(26), pp. 133-141. ISSN 1512-1232. http://gesj.internet-academy.org.ge . 2010.

[9] N.Archvadze, M.Pkhovelishvili. PRESENTATION OF THE GEORGIAN LANGUAGE DICTIONARY WITH FUNCTIONAL PROGRAMMING LANGUAGES, AND SEARCH BY THE METHOD "WAVE". Electronic Scientific Journal: "Computer Sciences and Telecommunications". ISSN 1512-1232. 2012|No.2(34)[2012.06.30], pp. 59-70. Impact Factor: 0.8125

[10] N.Archvadze, M.Pkhovelishvili. POSSIBILITY OF FUNCTIONAL PROGRAMS VERIFICATION THROUGH APPLICATION OF MODEL CHECKING. Electronic Scientific Journal: "Computer Sciences and Telecommunications". ISSN 1512-1232. 2013|No.4(40) [2013.12.31]. pp. 51-58. Impact Factor: 0.8125