ISSN 1512-1232

The Automatic Synthesis of Haskell Functions

Natela Archvadze[1], Otari Ioseliani[2], Lia Shetsiruli[3], Merab Pkhovelishvili[4],

[1] Department of Computer Sciences, Ivane Javakhishvili Tbilisi State University, 13 University str., 0186 Tbilisi, Georgia

[2] Georgian-American University, 8 Merab Aleksidze str., 0160 Tbilisi, Georgia

[3] Batumi Shota Rustaveli State University, 35 Ninoshvili str, 6010 Batumi, Georgia

[4] Niko Muskhelishvili Institute of Computational Mathematics, 8 akuri str., 0171 Tbilisi, Georgia

Abstract

According to the growing difficultness and reliability as well as rising effectiveness of the development requirements for applied programs, problems of automation of program synthesis process is the main concern of research. This problem is interesting for research in area of artificial intelligence and software engineering as well.

The analysis of the results of automated synthesis system research of programs using template Forms of Haskell functions is being conducted already. The system is designed for synthesis of correct and executable programs, recursive functions, which are represented in the tail recursion and recursion on the top of the list and parameter with accumulators.

The advantage of program synthesis consist from that at the same time with building the program it's approving the accordance of the program for given specification, that's why it's not required to perform verification of synthesized program in future.

Keywords: *functional languages, recursive functions, the templates of the functions*

1. Introduction

One of the main reasons of appearance the direction of program synthesis, is the growing requirements for reliability of software. Because it's not requiring verification process for the programs created with synthesizer, such programs must be forever "correct" (without errors) as syntactically and semantically as well.

After sixties, the problem of program synthesis represents the interests for research as in Artificial Intelligence and Software Engineering as well. The most influential researches are related to the programs created on Lisp. The appearance of program synthesis was caused because of necessity of exempting programmers from routine works of creation simple and elementary programs.

Besides it was using for initial education of users in functional programming (especially LISP), at first such kind of educator gives a short information about language components and verifies solutions of some problems and after gives user an opportunity to assign x and y and as a result it delivers *f* function. The synthesizer relieves programmer during creation large programs on LISP from many small auxiliary functions. The synthesizer of the program, builds this functions automatically, programmer indicates only x and y.

2. Overview

For the present moment, were composed certain set of approaches for program developing process automation by synthesis [1, 2, 3]:

- Deductive synthesis (logical Output) – Program is retrieved from the proving of theorems;
- Inductive Synthesis (The synthesis of programs by examples) – retrieving of Program from examples by summarizing methods, identifying sequences, progressions etc.
- Transformational synthesis (Transformation) – step-by-step transformation of specifications, specified on high-level programming language into low-level programming language code.

From the listed directions the most developed is deductive synthesis. For using this approach it's necessary to indicate particular statement of problems, this method isn't performing research of the relations between statements. According from this, the functional flexibility of the systems which are built using deductive synthesis is limited by the known spectrum of problem statements.

Modelling of data domains aren't performing in area of transformational synthesis it's performing the direct translation of one of descriptions into another.

In case of wide diversities of problem statements, the inductive synthesis doesn't restrict as the possibilities of system adaptation in embedding processes and possibilities of their future modernizations as well. This merit of inductive synthesis is the serious advantage of it in compare with other listed methods. At present time, there is a short list of solutions in area of inductive synthesis, for example the method of multipoint expressions, which gives an opportunity to perform the synthesis of the programs in rather narrow class. The fact is that there is one actual problem of development of subject domains using inductive synthesis of solutions depending on several examples, whereby composition of known algorithms with objective of using obtained components is performing decomposition in the process of next synthesis.

3. Our approach

The problem of synthesis which is one of the most difficult programs in area of programming can be represented as the development of automatic program generation by the computer for which foresaid problem wasn't solved and for which the computer has no solution.

We are representing generalization of synthesize problem in several directions, specifically the function is not only mathematics, it can represent the area of computer games, image processing etc. From the other side during development of computer equipment it became very important the problems as of processing lists structures and processing the problems of environment transformation as well. The environment means a file with content of sound data, image data etc. Therefore at task of examples, it's possible together with functions set up files, phrases, drawings or figures.

In earlier researches of LISP program synthesis [4,5] it was impossible to determine the dynamic structures, it's not determined the operation of list creation automatically, which required a lot of examples. The generation mechanism in Haskell gives an opportunity to generalize the example and that became the reason of possibility of simplification of examples.

The problem can be generalized for the functional programming languages the program can be transformed itself. This means that for examples will be taken programs and not lists. For example we have program named *reverse,* which returns a list: *reverse [[1,2],[3,4,5] = [[3,4,5],[1,2]]*, it's necessary to perform a synthesis for a kind of function which returns sub-list as well, For example, *reverse_all [[1,2], [5,4,3],[9,8]=[[8,9], [3,4,5],[2,1]]*

3.1. Function isn't only mathematics

The function shows one multiplicity in another. *y=f(x)* for *f* function in *x* point must correspond single value which is defined as *y*. This description isn't only for definitions the syntax of function don't confuse it with the semantics of function in functional programming languages. The functions can be different and they are using for different areas like: computer games, the problems of word processing, image processing compiler construction. Functional programming emphasizes a function that produces results which are depending only on their inputs and not on the program state – i.e. pure mathematical functions.

Let's consider examples from different areas:

- Mathematical functions: *Double: N->N* (the calculations of the factorial, square, sums etc.)

- The area of computer games: during realization of some computer games (For example tic tac toe). The function of the next step must be formulated like: It must be taken the condition of the board on input and must be generated a new condition on output: *NextMove: BoardState-> BoardState*. How this arguments are represented this is another matter. It's possible to perform the representation using lists, matrix, functions, which returns cross or zero putted on intersection of coordinates.

- Word processing, for example: translator from English to another language. The function of translation is the function which takes the text on output and puts the same but translated text: *Translate:text->text*.

- Image processing, for example the function of image processing takes image on input and puts the same image. For example: function which makes color picture black and white, *ToBlackWrite: image->image*. Function of image mirroring: *Mirror: image->image*, it's been described function which is performing some pixel transformation, for example color changing; it will accept another function which will reflect pixel to pixel: *Transform: (pixel->pixel) x image->image.*

- Problem of compiler construction: compiler is the function which accepts the source code of program and returns some byte-code: *Compile: sourceCode->byteCode.*

The compiler will be constructed as composition of lexical set, transformation, and optimization: *Compile = Parse ○ Transform ○ Optimize. (*○ – is the signifier of program composition). Each of them is a function with own input and output data.

3.2 The definition of program class

We are using the utility synthesis which means that specific approach in special cases. It's being considered the structural synthesis, which gives an opportunity of synthesizing the limited class of programs in a short subject area. Specifically it was determined a class of problems which contains lists and the problems of transformation of list structures. These types are determined syntactically oriented to construction method of Tony Hoare [6], particularly the list *A* of type can be defined as:

List(A) = NIL + (Ax List(A));

prefix = constructor List(A);

head, tail = selectors List(A);

isNil, isNonNil = predicates List(A);

nil,nonNil = parts List(A).

This definition of *List (A)* type in point of fact is the inductive set of some complex values which are created on based type *A – atom*.

Like a list structure, the syntactically-oriented constructions have following appearance:

ListStructure(A) = A + List(ListStructure(A);

prefix = constructor List Structure (A);

head, tail = selectors ListStructure(A);

is Atom, isN on Atom = predicates ListStructure(A);

atom,non Atom = parts ListStructure(A).

During synthesis, the examples will be determined by lists and list structures.

Such kind of formal definitions gives an opportunity to develop a template function for processing lists *(type List (A))* and list structures *(List structures (A)),* it can be used as specific functions for processing data, based on general template function.

Each function for processing value *List (A)* of type must contain at least two clauses, patterns. First is processing *NIL*, the second is processing *nonNIL,* for this two parts of *List(A)* type in Haskell usually corresponds following samples: *[ ] and (x:xs).*

Functions, which are processing list *structure (A)* data, must contain at least following clauses:

```
fl [ ] = ...
f1 a = if (isAtom a) then ...
                 else f2 a
f2 (x:xs) = if (isAtom x) then ...
                 else ...
```

3.3. The templates for representation of language functions

For developing lists and list structures is being used several templates [7, 8] during synthesis will be concretized the recursive functions which are parts of foresaid templates.

The template of recursive functions which gets one list as argument – a *list* [9, 10].

1. Form: tail recursion;

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

2. Form: the recursion comes on the head of the list;

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

The functions *g1, g2, g3, g4* and *g5* are depending on the goals of developers.

g1–function for processing empty list;

g2–function for combining the results of processing the head and the rest of not empty list;

g3–function of processing the head of not empty list;

g4–function for processing the result of recursive call for the rest of not empty list;

g5–function for pre-processing the rest of not empty list.

3. Form: for the functions with additional argument (accumulator).

```
fun n =fun' n a
           - - a call of function, the 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) )
```

Where: *g1, g5* – functions have the same values as in top and *g6* – is the function which is being processed by accumulator.

3.4. Creation of language functions collections

It was defined the functions from which must be formed the synthesis of the program. These functions are part of the Haskell language module ***Prelude.***

It was defined the specific type of the data *"Funtype",* which describes the functions according to the numbers arguments and type.

This is necessary during synthesis process when must be chosen function which argument's type corresponds the type of example and the target list, the result corresponds the type of elements.

During definition of type is considered that the functions are curried, polymorphic of types and have restriction for the type. The type of function is determining by the arrow and round brackets, the list is being determined by the [ ] square brackets.

data Funtype = Fun1 String String Listp Atom | Fun2 String Char Bool | Fun3 String String Atop Atom Atom deriving (Eq,Show)

The type "*Funtype*" is the descendant of standard Haskell *Eq, Show* types. We had represented only the part of definition, by only 3 constructors.

For example, first Fun1 constructor determines (combines) such a functions which can transform any type of list in Atom: *[a] ->a*. In the definition of constructor for first type *string*, corresponds the name of function and the second type *string*, determines the restriction. Functions *head*, *last, product*, *sum*, *and maximum*, *minimum* correspond Fun1 – constructor. For example the functions *head* and sum can be represented as:

Fun1 head " " [a] a

Fun1 sum "Num a" [a] a

The constructor Fun2 describes the functions which are influencing on symbols and as the result returns us logical values, these functions are: *isAlpha*, *isDigit*, *isLower*, *isSpace*, *isUpper.* For example the function *isDigit* can be represented as *Fun2 is Digit "" Char Bool.*

The constructor Fun2 describes the functions with arguments, like "*max*", "*min*", and "*div*", "*mod*". For example the function *div* can be represented like: *Fun3 div "Integral a" a a a.*

3.5. Inductive conclusion for inductive synthesis

It's known that the inductive inference is the inference from given data, which determines their general rule. For determining the general rule of inductive conclusion must be specified a set of the rules, the object of conclusion is an output object. The methods of representation of rules, the possibility of show the examples, output method, and the wrong output criterion.

As the example of *f* – function can be used a sequence of *(x, f(x))* couples with input and output values. For the "Turing" machine determining of *(x, f(x))* input and output couples for *f* – function, corresponds input *x* and output *f(x)* – values during program automatic synthesis and it's getting from calculation of *x* and *y*. According to this, the automatic synthesis of the program can be considered as the inductive output of function.

3.6. The example of inductive synthesis

If the initial argument is the following list *[A, B, C, D]* the result will be *[[A],[B],[C],[D]],* this can be written *[A,B,C,D] --> [[A],[B],[C],[D]]* for the both of sides is list structures. Let's review the problem of output of "?"

(A B C D) --> ?

At first, the system gives us two examples, it must be calculated the difference as for the left side of example and for the right side as well.

For example: *[A, B] --> [[A], [B]] and [A, B, C, D] --> [[A], [B], [C], [D]].*

It appears that the difference between arguments depends on the number of elements (*length* – function, in first case returns 2 and in a second case returns 4). We have similar quantity for the result lengths, according from this it can be made a conclusion, that a number of members in result is the same as it's in argument. That's why it will be checked first "Form" and will be tried to determine *g1, g2, g3* and *g4, g5* functions for determining *g1* function is being generated the query for the example which determines the result when is the empty list.

Fun [] - ->

Result: [] conclusion g1 x =x;

For determining *g2* and *g3* functions, is being used the heuristic, and will be performed the comparing between the argument an initial member of the result.

head [A, B]=A

head [[A], [B]]=[A]

According to this *A* – is being compared with *[A]* and after it will being calculated the difference between them, which means that must be found the function which will correspond *A* to *B*. For this reason in the set of functions of language is choosing the pattern which will correspond a list to Atom. Such kind of function is *cons (:), cons x [] =x: []*. Conclusion – *g3 x=x: []* because *x* – argument in not changing g2 *x =x* will be true. For determining *g4* and *g5* functions we are using given in the example processing of tiles of the lists.

tail [A, B]=[B]

tail [[A], [B]]=[[B]]

According to the fact that the number of members is not changing *(length [B] =length [B] =1)* is being created a conclusion that the recursion is performing for tile of the list. This means that *g4* and *g5* functions are identical: *g4 x = x; g5 x = x; the* conclusion of this process is the recursive definition:

fun [] = []

fun(x:xs) = (x:[]): fun xs

After this must be examined the correctness of synthesized function this will be performed by the call on *fun* function by a different number of arguments. At first will be checked *fun [], fun[a], fun [a, b]* and according to this will be created a conclusion that the synthesized function fun is true.

Besides the function *cons (:)* it is possible for other functions to be the function of set of language which are satisfying the conditions. According to this all of the functions are considered as the alternative of *cons (:)* function. It's being used the heuristic in compliance which is being considered a couple of functions (the combination of two functions) after the combination of following three functions etc. for that moment it is necessary that the type of output of first function be the type of the second function argument and be compliant with given examples.

3.7. From examples to Algorithms

During execution of the system is being distributed the three main steps: selection of the templates, construction of the function, examining of the constructed function.

During selection of template is being performed according to a number of arguments of the function which must be constructed. If there are a several templates it will be performed checking of each of them.

The functions which are represented in template will be constructed using several heuristics. During choosing functions will be used all the functions. The conclusion is that it can be several alternatives of synthesized function.

At the third stage is being checked the synthesized alternative functions on the examples. During this process is being performing calculation of the real values of functions.

4. Conclusions

The program synthesis algorithms for construction of recursive functions have been discussed already. The synthesis is being performed using different recursive templates. These templates are using as for program synthesis and for verification as well [11, 12, 13].

We are representing generalization of synthesize problem in several directions, specifically the function is not only mathematics, it can represent the area of computer games, image processing etc. From the other side during development of computer equipment it became very important the problems as of processing lists structures and processing the problems of environment transformation as well. The environment means a file with content of sound data, image data etc. Therefore at task of examples, it's possible together with functions set up files, phrases, drawings or figures.

References

1. Drozhdin V.V., Zhukov M.V. –The deductive synthesis of functional and imperative programs// Izvestia Penz.gos. pedagog. univ. 2009, N.13 (17). Pp.89-94.
2. Sh.Barman, R.Bodic, S.Jain,Y.pu, S.Srivastava, N.Turg. Parallel Programming with Inductive Syntesis.University of California, Bercley.
3. Monaxov O.Evolucionnii cintez na ocnove shablonov(ru). Novosibirsk, 2006.
4. 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.
5. 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.
6. Dushkin R.B.Functionalnoe programirovanie na Haskell (ru). 2007.
7. 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
8. 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.
9. Graham Hutton. Programming in Haskell. Cambridge, 2007.
10. 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. 2010.
11. 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.
12. 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.
13. 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.

Article received: 2014-09-05