Mathematical Problems of Computer Science 31, 130–141, 2008.

# On One Approach to Optimization of Recursive Function Computations

Artashes K. Ghazaryan

Institute for Informatics and Automation Problems of NAS of RA.
E-mail: artashesg@gmail.com

**Abstract**

The goal of this work is the theoretical justification and the development of a new optimizer that synthesizes programs calculating multivariate recursive functions and systems of functions.

The current version of the optimizer processes a wide category of multivariate systems of recursive functions using two algorithms – stack recursion optimization and combined total replacement optimization.

The results of this work can be used in development of packages, calculating the systems of recursive functions, modeling discrete multivariate systems with complex interconnections, solving boundary-value and field-value problems, etc.

## 1 Introduction

The recursive description for a wide class of algorithms has a major convenience for developers, since the recursive approach allows a very compact description. In addition, the recursive description approach is more conformed to human abstract way of thinking, which builds construction of different levels and sets up relationships between them, unlike the iterative approach when it becomes necessary to describe all steps of algorithm consecutively, to solve the task.

However, the majority of the existing programming languages do not support convenient facilities for recursion organization. As a result, the programs, directly corresponding to recursive description, turn out to be slow, spending a lot of memory and are difficult to debug.

There is also a whole class of recursive descriptions, the direct implementation of which simply will not work on most of high-level programming languages. For instance, consider the following recursive description:

$$F(x_1, x_2) = \begin{cases} 0 & \text{if } x_1 = 0; \\ F(0, F(x_1, x_2)), & \text{otherwise;} \end{cases}$$

Depending on the way the grammatical parsing is implemented for a particular compiler, on argument $x_1 \neq 0$ and arbitrary $x_2$ the function $F(x_1, x_2)$ will either always return 0, or the program will "hang up" until the program stack overflow and emergency stop. For example C/C++ compilers on this example will "hang up".

There are systems, which use the ideology of the so-called "lazy calculations" which can deal with such tasks. This approach is used in functional programming languages (for example: Mathematica by Wolfram Research, Haskell), but in essence it cannot be fast. Moreover, the functional languages do not guarantee the order of operations execution. This means that there are difficulties with the input/output operations; with calling native OS functions (as they often require the right calling order because of the side effects). So the interaction with the "outside" world is quite complicated.

The approach proposed in this paper allows not only to combine the comfort of recursive description with the efficiency of iterative approach, but also to expand the class of solvable tasks, using the theory of recursive functions, in particular, some of the results of the "fixed point" theory.

Using a combination of automatic program generation techniques, the stack recursion, the "fixed point" theory, the package FARECO (stands for *FA*st *REC*ursive *O*ptimization) were developed.

Currently the FARECO package is at the stage of testing and approbation at the Institute for Informatics and Automation Problems of the National Academy of Sciences of the Republic of Armenia.

## 2 Theoretical Justification

Let us define the class of recursive programs, which we deal with. To do this, let us first give a few definitions.

Let's introduce some notation. Let $f$ be an n-place function with the definitional domain $D^n = D \otimes D \otimes \dots D$ and the range domain $D$. It is convenient to introduce some *undefined* element and denote it by $\omega$. Let us also denote $D^+ \equiv D \cup \omega$, and suppose that $\omega \notin D$. Then any n-place partially defined function $f$, which maps $D^n$ into $D^+$, can be regarded as a completely defined function. If $f$ is not defined on some n-tuple $\langle a_1, a_2, \dots a_n \rangle \in D$ then $f(a_1, a_2, \dots a_n) = \omega$.

Let's call the $n$-place function $f : [(D^+)^n \to D^+]$ *efficiently computable* (or just computable), if there exists an algorithm which computes it, i.e. there exists such an algorithm $A$ which accepts the vector $\vec{x} \equiv \langle x_1, x_2, \dots x_n \rangle \in D^+$ as an input, and the calculations according to that algorithm should finish after a finite number of steps and produce the result $f(\vec{x})$.

All computable functions can be constructed from a limited set of base functions, namely a constant zero function $0(\vec{x}) \equiv 0$, one-place increment function $S(x) = x + 1$, the function-selector $C(i, \vec{x}) = x_i$, and operations of superposition, primitive recursion and minimization.

All functions, which can be constructed from the basic functions and the above mentioned three operations on finite number of steps, are called *partially recursive*. If such a function is everywhere defined, then it is called a *general-recursive* function. If the function is constructed without the operation of minimization, then it is called a *primitively-recursive* function.

For further discourse, we need a notion of *partial ranking* [3], that will be denoted by $\sqsubseteq$. To partial ranking corresponds "*less definite than or equal*" relation. We also suppose that for all $d \in D^+$ it is correct that $\omega \sqsubseteq d$ and $d \sqsubseteq d$. Now it is possible to introduce the concept of *monotonic* functions. We would like to say that the n-place function $f$ with the definitional domain $D_1^n$ and the values domain $D_2^+$ is monotonic if from $x \sqsubseteq y$ follows

$f(x) \sqsubseteq f(y)$ for all $x, y \in D_1^n$. Hence, the partial ranking for monotonic functions $f$ and $g$ with the definitional domain $D_1^n$ and the values domain $D_2^+$ can be easily defined:

1. $f \sqsubseteq g$ if $f(\bar{x}) \sqsubseteq g(\bar{x})$ for all $\bar{x} \in D_1^n$.

2. $f \equiv g$, if $f(\bar{x}) \equiv g(\bar{x})$ for all $\bar{x} \in D_1^n$.

Next, let us introduce the concept of *functional*. Let us call functional the operations on functions. In other words, it is a function that takes functions as its argument or input and returns a function. The functional $\tau$ on the set $[D_1^n \to D_2^+]$ maps the set of functions from $[D_1^n \to D_2^+]$ to itself, i.e. $\tau$ takes an arbitrary monotonic function $f$ that maps $D_1^n$ into $D_2^+$ as its argument and produces some other monotonic function $\tau(f)$ that maps $D_1^n$ into $D_2^+$.

Now let us introduce the concept of *"fixed point"* and *"the least fixed point"* for continuous functional. Let $\tau$ be a functional over the set $[D_1^n \to D_2^+]$. We say that the function $f \in [D_1^n \to D_2^+]$ is the fixed point of $\tau$ if $\tau(f) \equiv f$, i.e. $\tau$ maps $f$ to itself. If $f$ is the fixed point of $\tau$ and $f \sqsubseteq g$ for any other fixed point $g$ of $\tau$ then $f$ is called the least fixed point of $\tau$.

Finally let us now formulate the S. C. Kleene First Recursion Theorem [10], which is the basis of FARECO Optimizer.

**Theorem 1 (The S. C. Kleene First Recursion Theorem, [10])** *Let $\tau$ be a continuous functional. Then there exist a computable function $f$ that is the least fixed point of $\tau$.*

1. $\tau(f) = f$
2. *if* $\tau(h) = h$, *then* $f \sqsubseteq h$

The strength of this theorem[1] consists in the following: the recursive definition of a very general type can be represented by an equation of the type

$$F(\bar{x}) = \tau[F](\bar{x})$$

where $\tau$ - is some functional. The fixed point theorem asserts that there **always** exists a computable function that satisfies that equation.

Now we can go directly to the description of the ideas which underlie on Optimizer. We will call a recursive definition or a recursive program such a program over a set $D$ if it has the form, that:

$$F(\bar{x}) \Leftarrow \tau[F](\bar{x})$$

where $\tau[F](\bar{x})$ is functional on set $[(D^+)^n \to D^+]$ constructed with the help of superposition of *base functions* and *predicates* and the functional variable $F$.

Let us describe the process of function calculation, which is defined by recursive program as follows:

[1] As a matter of fact, the First Recursion Theorem was proved for computable functionals. However, it is not difficult to obtain the above given formulation by means of combining the B. Knaster's "fixelement" theorem [8] with the fact that the computable functionals are continuous. This make a bridge between the notions of continuous functional and its particular case - computable functional.

1. The first term $t_0$ is $F(\vec{x})$.
2. The term $t_{i+1}$ should be constructed from $t_i$ for each $i \geq 0$ with the help of the following operations:
   - Substitution: replace some occurrences (we shall see it below) of $F$ in term $t$ simultaneously by $\tau[F]$.
   - Reduction: replace the basic functions and predicates by their values every time, *whenever it is possible*, until the further simplification will be impossible.

The above described computational sequence terminates, and the term $t_k$ is the last in this sequence if and only if $t_k$ contains no occurrences of $F$ i.e. $t_k$ is an element of the set $D^+$.

The term sequence $t_0, t_1, t_2 \ldots$ defined above is called *computational sequence* or *computation* for $F(\vec{d})$.

We will say that the rules for calculating $C$ are rules for calculating the fixed point, if for each recursive program over $P$ on the set $D$

$$C_P(\vec{d}) \equiv f_P(\vec{d}) \text{ for all } \vec{d} \in (D^+)^n$$

In chapter 5 of Z. Manna book[3] one can find 6 substitution and reduction rules, as well as the advantages and disadvantages of each of them, but here we are going to consider only one rule - the full substitution rule. It consists in replacing all occurrences of $F$ simultaneously.

Let us illustrate the full substitution rule by an example. Let us consider the term consistng of a fuctional variable and arguments:

$$F(\vec{x}, F(\vec{x}, \vec{y})) + F(F(\vec{x}, \vec{y}), F(\vec{x}, \vec{y}))$$

According to the full substitution rule all occurences of $F$ are to be replaced: $\underline{F}(\vec{x}, \underline{F}(\vec{x}, \vec{y})) + \underline{F}(\underline{F}(\vec{x}, \vec{y}), \underline{F}(\vec{x}, \vec{y}))$

In Manna's book one can find the proof that the full substitution rule is the rule for calculating the fixed point (see the theorem of J. Vuillemin, on the "safe" rules).

The full substitution rule is used in FARECO Optimizer as one of optimization major steps. The output of that optimization algorithm is a fixed point of the source recursive definition. This very step can significantly enhance the class of problems which can be processed by the Optimizer.

Another, not less important optimization phase increases drastically the speed of calculations. That phase is *the stack recursion optimization*. The general idea of the stack recursion optimization method and its mathematical justification have been developed by Prof. Hrant Marandjian and described in [1]. That idea is the following: all recursive computations require values, calculated on the "previous" arguments, and often the values of computations on some arguments are calculated more than once, although there is no need in such repeating computations. Correct computation organization, which eliminates such unnecessary computations can save a lot of time and memory. Let's explain this on an example of a well-known Fibonacci function:

$$F(x) = \begin{cases} F(x-1) + F(x-2); & \text{if x ¿ 1;} \\ 1; & \text{otherwise;} \end{cases}$$

Let us examine the calls tree of Fibonacci function on argument 4 (look Fig. 1).

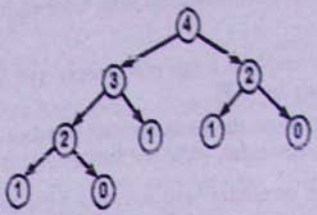


Fig. 1

The tree has 9 vertices with a figure in each corresponding to the number of recursive procedure calls. Easy to see, that the computation procedure is repeatedly called on those values, which were already calculated earlier. The table with the numbers of calls on each argument for the initial argument equal to 4 is the following:

| Point | Calling |
|---|---|
| 0 | 2 |
| 1 | 3 |
| 2 | 2 |
| 3 | 1 |
| 4 | 1 |

If consider it natural that for each argument the program should be called only once, then from 9 calls of Fibonacci function on argument 4, four calls are unnecessary. Thus, if organize the calculations in such a way that the extra calls do not occur, it is possible to obtain very serious gains in computation speed and considerable memory savings. The memory savings are also very significant achievement in the recursive calculations because each recursive call requires retaining the current call context (current state of the processor's registers, etc.) and a new context should be created. For deep recursion some functions are simply not computable due to stack overflow, i.e. because of technical rather than mathematical complexities. The essence of the method is to organize a special calculation process, according to which the calculated values are stored with their arguments. Before any call the algorithm verifies the existence of the result of function computations on the proposed argument and only in case of absence of the last, the real calculation starts. Otherwise, the previously calculated value is immediately returned as a result. The Optimizer generates special execution blocks for each recursive function call which provide the stack recursion optimization. For this optimization method there exists the proof of correctness.

The current version of the FARECO Optimizer takes advantage of these two principal ideas and also uses some simpler, but, nonetheless, quite useful techniques. One of these techniques is the realization of arbitrary types of arguments, based on "tuple" container from BOOST library. This possibility is relied on the following confirmation: while most theoretical assumptions are based on dealing with integer arguments, it is easy to show that more complex arguments can be used - those that are constructed of combination of a simple, built-in types, and these simple ones - by coding of integer numbers (see Overflow algorithm [7]).

## 3 Optimizer Description

The Optimizer is an executable module which is written in C++. It uses standard libraries BOOST and STL, as well as GNU Bison and GNU FLEX utilities. Without any change of source code the Optimizer can be compiled as for OS Windows XP / Vista using Visual C/C++ compiler as well as for Linux using GCC compiler.

The general Optimizer structure is presented on Fig. 3

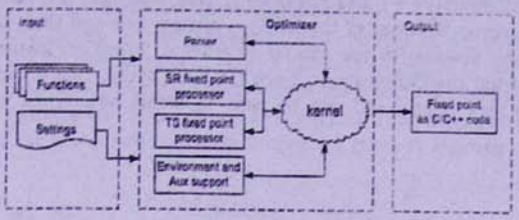


Fig. 3

As an input the Optimizer accepts a settings file. It contains the names of the files with recursive functions descriptions written on the subset of C/C++, some settings associated with the optimization level for each function, as well as some auxiliary settings necessary to instruct the Optimizer to generate the caller functions, the calls counting subsystem, the timing subsystem. The example of such file is given below:

```
# possible options:
# timing = 1 - need timing          |counter = 1 - need counter of calls
# no = fname - no optimize          |sr = fname - stack_recursion only
# ts = fname - full substitution    |mf = fname - master function
# input = file - input file name
input = fib.txt
sr = F
mf = F
timing = 1
counter = 1
```

The current version can perform the stack recursion optimization and the total replacement optimization combined with the stack recursion optimization. The number of functions is not limited by the system. The depth of recursion is limited only by the program stack, which can be configured with the help of appropriate compiler options. Referring to the subset of C/C++ let us explain what is meant by an example: let us consider the Ackermann function $Ack(m, n)$.

$$\text{Ack( m, n )} = \begin{cases} \text{n + 1} & \text{if } m \leq 0 \\ \text{Ack( m - 1, n )} & \text{if } n \equiv 0 \\ \text{Ack( m - 1, Ack( m, n - 1))} & \end{cases} \qquad \text{Fig. 2}$$

Let us write it as follows:

```
int Ack(int m, int n)
{
```

```
    if (m <= 0) { return n+1; }
    if (n==0) { return Ack(m-1,1); }
    else { return Ack(m-1,Ack(m,n-1)); }
}
```

This listing is fully consistent with the syntax of C/C++. The only requirement for such a description is that the body of a function should consist only of constructions like this: if(cond){c++}...if(cond){c++} and else{c++}. The body of "if's" and "else" blocks can encapsulate any correct in terms of the C/C++ syntax code and that code should correctly compile. It is also possible to use special comments, which are transparent for C/C++ compiler but instruct the Optimizer to place the code within that comments verbatim into output. This is very handy and can be used to define new types within the functions definition files.

```
//$
class Int
{...}
//!$
Int Ack(Int m, Int n)
...
```

Easy to see that we introduced a new type *Int* instead of the existing basic type *int* and both optimized and not optimized source codes can be compiled and executed. There are minor peculiarities about some mandatory functions which should expose such classes. At the end of next session this peculiarities will be shown up.

Once the configuration file and the files with the descriptions of functions are prepared, the Optimizer can start. As a result it produces a header file which should be included into a user's program. Then the optimized functions can be called either directly or through special caller functions, which include a call counter and a timing subsystem. In majority of cases, it is convenient to use it, but if it is necessary to call directly the optimized function, one should refer to the function with the same name and signature as the function in the input file.

## 4 Results of Experiments

In order to examine the optimization abilities of FARECO Optimizer a set of experiments were performed. In this section the results of comparisons of execution time and the number of calls for optimized and not optimized primitively-recursive function (Fibonacci function); general-recursive function (Ackermann function) and "hanging" function will be provided.

### 4.1 Fibonacci function

Fibonacci function is specified by the following recursive description:

$$F(x) = \begin{cases} F(x-1) + F(x-2); & \text{if x ¿ 1;} \\ 1; & \text{otherwise;} \end{cases}$$

The C/C++ description which plays at the same time the role of input for Optimizer is the following:

```
int F(int x)
{
    if ( x < 2 ) { return 1; }
    else { int ret = F(x-1) + F(x-2); return ret; }
}
```

The source function as well as optimized by stack recursion algorithm were compiled into executable modules and executed on the same arguments.

Two indicators were chosen for consideration the number of recursive calls and the computation time, spent on each argument. Fig. 4 shows dependence of the number of calls upon the argument. Here x-coordinate is the argument value and y-coordinate is the number of calls given in logarithmic scale. Fig. 5 shows dependence of the computation time (in ms) upon the arguments. For this figure both coordinates are given in logarithmic scale.

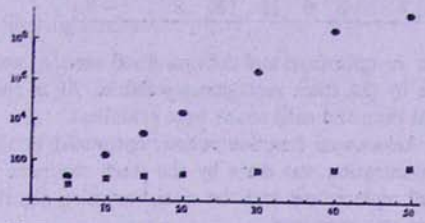

Fig. 4. Calls numbers of not optimized and optimized functions (circles - not opt., squares - opt.
)

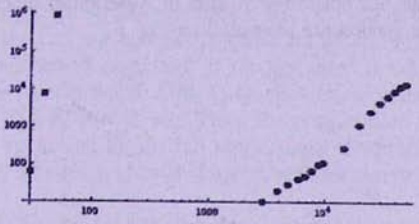

Fig. 5. Computation time of not optimized and optimized functions (squares - not opt., circles - opt.)

Using "Mathematica" package the fitting formulas were constructed. The rough outline of that fitting is presented below:

| Not opt. calls | Opt. calls | Not opt. time | Opt. time |
|---|---|---|---|
| $0.13x^6$ | $x+1$ | $0.12x^4$ | $0.22x$ |

Figures and formulas clearly show a substantial acceleration of computation time. and the difference in the calls number speaks about significant memory savings.

## 4.2 Ackermann function

The Ackermann function is a simple example of computable functions, which is not a primitive recursive. It takes two non-negative integer numbers as arguments and returns a positive integer. This function is growing very quickly. For example, the number of digits in decimal representation of $Ack(4,4)$ is so big that it repeatedly exceeds the number of atoms in the observable universe. The precise analytical values of Ackermann functions for the arguments within the limits of $[0-4]$ are presented in the following table:

| n/m | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 5 | 13 |
| 1 | 2 | 3 | 5 | 13 | 65533 |
| 2 | 3 | 4 | 7 | 29 | $2^{65536}-3$ |
| 3 | 4 | 5 | 9 | 61 | $2^{2^{65536}}-3$ |
| 4 | 5 | 6 | 11 | 125 | $2^{2^{2^{65536}}}-3$ |

For Ackermann function the optimized and not optimized versions were also constructed. The optimization was done by the stack recursion algorithm. As in the previous case the differences in computational time and calls count were examined.

Two additional sets of Ackermann function values, optimized by different algorithms, were constructed. One optimization was done by the stack recursion algorithm and the other - by the combined full replacement and the stack recursion algorithm. For this case the execution time difference was examined.

In order to eliminate the problem of the function fast growth the value of the function can be taken by modulo of some prime number. For figures below prime number 91 was chosen.

The calculations times for not optimized version of Ackermann function and optimized by stack recursion algorithm version are presented on Fig. 6.

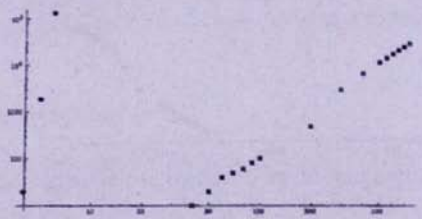

Fig. 6. Computation time of not optimized and optimized functions (diamonds - not opt., squares - opt.)

It is obvious that this figure is very similar to the corresponding one for Fibonacci function. The figure for the number of calls is also virtually the same as the corresponding one for Fibonacci function, so it is not presented here for space saving purposes.

Fig. 7 presents the calculation times for combined algorithm and stack recursion algorithm. This experiment was performed to show that the combined algorithm does not worsen the calculation time, while, as it was shown in Section 2, the class of solvable problems has been widened.

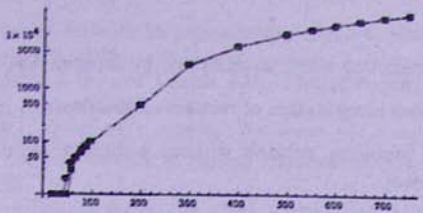

Fig. 7. Computation times of combined algorithm and stack recursion algorithm (circles - combined, squares - stack rec.)

Both figures use logarithmic scales.

## 4.3 "Hanging" function

Let us examine the following function:

$$F(m,n) = \begin{cases} 0 & \text{if } (\ m \equiv 0\ ) \\ \text{F(0,F(m,n))} & \text{othervise;} \end{cases}$$

On C/C++ this function will be written as follows:

```
int F(int m, int n)
{
    if ( m == 0 ) { int ret = 0; return ret; }
    else { int ret = F(0,F(m,n)); return ret;}
}
```

Execution of this program on any arguments, with exception of $m \neq 0$ leads to "hang". It will be so because the C/C++ compiler requires for all arguments of executing function to be known at the moment of execution. If the argument is a function then it should be calculated. Hence, the calculation of $F(m,n)$ requires to calculate $F(0,F(m,n))$, which in its turn requires $F(0,F(0,F(m,n)))$, etc. Thus, the computation will never end.

If this function is optimized by the full replacement algorithm, then the result will be the fixed point of that function and calls of optimized function on arbitrary arguments will give the expected 0.

The main difference between the current approach and the one, described in [2] and [11] is that this implementation of the Optimizer uses the full replacement optimization strategy in addition to the stack recursion optimization. As it was shown in Theoretical Justification section this approach significantly expands the class of problems that can be solved by the Optimizer.

This implementation allows not only to integer arguments for functions, but also arbitrary simple types such as double, char, predefined STL containers such as strings, vectors and it also allows to define the developer's own types, by defining the corresponding C++ classes. That class can have an arbitrary complexity. The only requirements for the class design are the defined copy constructor, operators less (<), equal (==) and stream inserter (<<). The last two should be defined outside of the class definition scope.

The architecture of data structure of the Optimizer was designed to allow the parallel computations as well

### 4.4 Summary

According to theoretical prediction which is confirmed by experiments, the Optimizer

- considerably accelerates computation of recursive functions;
- expands the class of problems, solvable without additional effort and without much computational overhead;

However, there are some difficulties associated with the introduction of fictitious variables and functions into a system of recursive equations. Currently the Optimizer does not recognize such cases. But this does not diminish its value, as these problems also cannot be solved in usual way.

Using Rice's theorem [9] it can be easily shown that the problem of whether a given computable function contains fictitious variables or not, is recursively unsolvable. In some very simple particular cases the occurence of fictitious variables or functions can be detected via analyzing the given function definition grammatical structure.

Nevertheless, we suppose to develop the Optimizer in this direction. It seems promising to implement some kind of "lazy" computations from the world of functional programming, but without an undetermined computations order.

## 5 Future Optimizer Development

Two ways of Optimizer development are planned. One way is computation distribution and parallelization, which can accelerate computations in many cases. Another way is expanding the class of tasks the Optimizer can deal with. So far, more priority has the first way. However, further research will also be conducted in the direction of extension of problems class.

The next version of the package, which is currently under development, is supposed to be parallel. It is expected that this will also expand the class of solvable problems, because some problems have unacceptable long computational time and hence, currently they are practically unsolvable.

The parallel and distributed computations can give effect in cases where the computational environment setup time and the time on data transfer between the computing units are considerably less than the time of the calculations (Amdahl's law [12]).

Thus, the domain of problems addressed through a parallel version of the package FARECO is clearly defined: it is different kinds of problems from physics, reducible to a boundary-value or field-value problems where the computations in any point depends on values in adjacent points and computably cumbersome. These include problems such as the computations of distribution of neutron and thermal fields in nuclear reactors, modeling of movement of physical body in mediums, etc.

## References

[1] Мараңджян Г.Б, "Об одном методе синтеза программ числовых функций", *Математические вопросы кибернетики и вычислительной техники*, XVI, 1986.
[2] Marandjian H., *General form recursive equations*, CSL, pp. 501-511, 1994.
[3] Manna, Z., *Theory of Computation*. NY, McGrow-Hill 1978.

[4] Barron D., *Recursive methods in programming*, General Editor: Stanley Gill Associate Editor: J. J. Florentin, 1969.
[5] Aho A.V., Hopcroft J. E. and Ulman J.D., *Data Structures and Algoritms*. Addison-Wesley, Reading, Massachusetts. 1983.
[6] Barendregt, H. P., *The lambda calculus. Its syntax and semantics*. North-Holland. 1984.
[7] Ghazaryan A., *On one method of flexible numeration*, Proceedings of the conference. CSIT, p. 15, 1997.
[8] Knaster B. *Une théorème sur les fonctions d'ensembles*. Annales Soc. Polonaise Math., 62, pp. 133 - 134, 1927.
[9] Rice H. G., *Classes of recursively enumerable sets and their decision problems*. Trans. Amer. Math. Soc, pp. 358 – 366, 1974.
[10] Kleene, S. C., *Introduction to Metamathematics*. New York - Toronto, D. Van Nostrand Co., Inc., 1952.
[11] Халатян, И. Г., Пакет прикладных программ - автоматический программный синтез. *Тезисы докладов Третьей Республиканской конференции аспирантов Армянской ССР*, Часть 2, Ереван, сс. 16 -17, 1989.
[12] Amdahl G. M., *Validity of the single-processor approach to achieving large scale computing capabilities*. In AFIPS Conference Proceedings vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston, Va., pp. 483-485, 1967.

## Ռեկուրսիվ ֆունկցիաների հաշվման օպտիմալացման մի եղանակի վերաբերյալ

Ա. Ղազարյան

### Ամփոփում

Այս աշխատանքի նպատակն է նոր օպտիմալացման մշակումն ու տեսականորեն արդարացված լինելը, որը սինթեզում է բազմաչափ ռեկուրսիվ ֆունկցիաներ և ֆունկցիաների համակարգեր հաշվող ծրագրեր: Օպտիմալարին ներկայացվող տարբերակը մշակվում է ռեկուրսիվ ֆունկցիաների բազմաչափ համակարգերի լայն դասի՝ օգտագործելով երկու ալգորիթմ. ստեկային ռեկուրսիայի օպտիմալացում և լրիվ փոխարինման ալգորիթմի միավորված օպտիմալացում: Այս աշխատանքի արդյունքները կարելի է օգտագործել այն ծրագրաշարերի մշակման մեջ, որոնք հաշվում են ռեկուրսիվ ֆունկցիաների համակարգեր, մոդելավորում են բարդ փոխկապակցություններով ընդհատ բազմաչափ համակարգեր, լուծում են եզրային և դաշտային խնդիրներ և այլն: