POLYNOMIAL-TIME EFFECTIVENESS OF PASCAL, TURBO PROLOG, VISUAL PROLOG AND REFAL-5 PROGRAMS

Nikolay Kosovskiy, Tatiana Kosovskaya

***Abstract**: An analysis of distinctions between a mathematical notion of an algorithm and a program is presented in the paper. The notions of the number of steps and the run used memory size for a Pascal, Turbo Prolog, Visual Prolog or Refal-5 program run are introduced. For every of these programming languages a theorem setting conditions upon a function implementation for polynomial time effectiveness is presented.*

*For a Turbo or Visual Prolog program It is proved that a polynomial number of steps is sufficient for its belonging to the class **FP**. But for a Pascal or Refal-5 program it is necessary that it additionally has a polynomially bounded run memory size.*

***Keywords**: complexity theory, class **FP**, programming languages Pascal, Turbo Prolog, Visual Prolog and Refal-5.*

***ACM Classification Keywords**: F.2.m ANALYSIS OF ALGORITHMS AND PROBLEM COMPLEXITY Miscellaneous.*

Introduction

What is an effectively in time computable function? In the frameworks of computational complexity it is accepted that this is a function which belongs **FP**. The definition of this class is done in the terms of a Turing machine. The class **FP** is defined as the class of all algorithms which may be calculated by a Turing machine in a polynomial under the input word length number of steps [Du 2000]. Theoretically the Turing machine is a very good instrument because all the steps fulfilled by it while a function computation are identically simple and may be regarded as using the same amount of time. But who writes a Turing machine program? We prefer a high level programming language.

Can we estimate a run time of a program according to the number of the fulfilled operators? For example, is it true that the fulfilling of the next tree operators needs three units of time?

x:=z^2; y:=5; z:=log(sin(x * y));

In dependance of a translator implementation the first operator either may be represented in the form *if* <exponent> = 2 *then x:=z*z* or the value of *x* is calculated be the formula e^{2*ln(z)}. The values of logarithm

and exponent are calculated with some (enough good) precision. The first alternative is used in the most of translators for calculation of a square, cube, ... And what is for 25^{th} power?

The second operator, of course, uses one unit of time.

The calculation of the third operator as a rule uses an expansion in series of functions log and sin.

And how many steps of calculation are here?

It is well known that the solving of a system of linear equations with integer coefficients by means of Gauss method may be implemented in a polynomial under the dimensionality of the system number of operations (addition and multiplication). But even if we solve a system with 5 variables with great enough absolute values of coefficients then the overflow occurs. The "fighting" with such an overflow demands the necessity either to introduce a structure simulating a number of great capacity or to use an approximate method. But can we guarantee that even after that the number of the used operations is adequate to the time of a program run?

Below there are presented theorems setting conditions under which a program in Pascal, Turbo Prolog, Visual Prolog or Refal-5 [Babaev 1992] performing a polynomial number of steps calculate a function belonging to the class **FP**. Such a program may be called a polynomially effective one.

Another approach to the simulation of the class **FP** is presented in [Beltiukov 2006]. This approach is based on the selection of such a subset of Pascal which provides to receive such a class of functions which equals to **FP**. It uses a variant of bounded recursion introduced by A. Grzegorczyk.

A program as an algorithm

Some researchers assume that the notions of a program and an algorithm coincide. Somebody suppose that a program is an implementation of an algorithm. Let's try to clear up the question. Here under the notion "algorithm" we will understand a mathematical notion of an algorithm such as a Turing machine, Markov normal algorithm, Markov-Post algorithm [Kosovskaya 2010]. Here under the notion "program" we will understand a program written in such a language as one from the Pascal languages family, or such programming languages destined for the solving of traditional Artificial Intelligence problems as Turbo Prolog, Visual Prolog and Refal-5 [Babaev 1992].

A mathematical notion of an algorithm implies a potentially infinite set of both the input data and the results of an algorithm run.

That is why only file may be used as an input data for a language from the Pascal languages family regarded as a mathematical notion of an algorithm. But the notion of the value type *integer* may be extended in the following way. We allow to use integers with unbounded number of digits. Such an unbounded number may be implemented with the use of dynamic linear arrays with the elements of the initial value type *integer*. A function of such a new type integers will be called a pascal-like function. The set of all such functions for every language from the Pascal languages family may be regarded as a mathematical notion of an algorithm.

Analogous notions may be introduced for the other programming languages from the Pascal languages family.

For the dialects of Prolog it may be, for example, lists of elements of the value type *integer* or files or constant terms without variables in some (not interpreted) signature.

Here a mathematical notion of transformation object for Turbo or Visual Prolog language is regarded as any list of elements of the value type *integer* (including an empty list). As a rule such an element is a remainder modulo 2^{16}, more precisely any integer between -2^{15} and $2^{15} - 1$ inclusively. In such a case Turbo or Visual Prolog may be regarded as a mathematical notion of an algorithm.

For the programming language Refal-5 analogous notion of transformation object may be a sequence of macro-digits which are separated in the program text by blanks. A macro-digit is a digit 0 or a sequence of decimal digits not beginning with 0 and defining a decimal notation of a positive integer less then 2^{32}.

Number of steps and run memory size for some algorithm notions

It is accepted in the theory of algorithm complexity that every run complexity characteristic is regarded as a function of the input data length. This function equals maximum of the complexity characteristic under consideration upon all input data with the same length. The value of such a characteristic will be estimated up to a multiplicative constant. The notion of the input data for every programming language under consideration was done in the previous section.

The **number of steps** and the **run memory size** are the most naturally defined **for a Turing** machine. They are respectively the number of fulfilled commands and the number of cells visited by a Turing machine head.

The **number of steps** and the **run memory size for the normal Markov algorithms** may be defined respectively as the number of fulfilled substitutions and the maximal word length in the sequence of transforming words beginning with the input word and ending with the result of algorithm run.

The **number of steps for a pascal-like function** is defined as the number of fulfilled statements and computed expressions and sub-expressions (boolean and arithmetic) during the run of a program receiving a result according to the rules of Pascal semantics.

The **run memory size for a pascal-like function** is defined as maximum upon all computational steps (beginning with the input data and ending with the result) of the sums of the record lengths of all calculated variables values (including the values of all elements of all arrays) . The length of the stack of variable values for fulfilling embedded procedures and pascal-like functions (including recursive ones) is taken in account. While running a call of such a procedure or function a separator followed by actual parameters and all local variable values of the call is put into this stack. Besides that the length of the stack of definition descriptions of the called procedures and functions sufficient for all program run is taken in account. The notation length of a variable or an array element which has no value up to that moment equals 1.

As you can see the full definition of the number of steps and the run memory size for a pascal-like function contains full definition of Pascal operational semantics for its used version.

The **number of steps for a Turbo or Visual Prolog query** is defined as the number of fulfilled calls of a predicate appeared during the run of this query (including all calls of built in predicates). It is clear that this definition is much shorter than the definition of number of steps of pascal-like function.

The **number of steps for a Refal-5 function** is the number of fulfilled Refal-5 rules.

The **run memory size for a Refal-5 function** is the sum of the length of expression and the length of a built-in Refal-5 stack, maximal upon all steps of calculation. I.e. the maximum of the lengths of all expression records (beginning from the initial and ending the final) appearing according to the semantics of refal-5 functional expressions computation.

The number of steps and the run memory size for the regarded algorithm notions will be used in the theorems of the next section.

Polynomial-time computations

FP is the usual denotation of the class of all functions computable by a Turing machine with the number of steps not more than a polynomial under the length of the input word [Du 2000].

The words "Turing machine" may be replaced by the words "normal Markov algorithm" (as it was proved by G.S. Tseitin in the Leningrad seminar on mathematical logic approximately 40 years ago). He proved the following theorem.

Theorem 1. The class of all functions computable by a normal Markov algorithm with the number of steps not more than a polynomial under the length of the input word equals to the class **FP**.

The proof of this theorem is based on the simulation of a normal Markov algorithm by a Turing machine with a suitable number of steps.

This analogous theorem is not valid for pascal-like functions. For example, the function 2^n may be calculated by a pascal-like function in not more than $log(n)$ (which is approximately equal to the length of n) number of steps. But this function does not belong to **FP** as the length of its result is n and hence the number of a Turing machine steps can't be less than $n \approx 2^{||n||}$, where $||n||$ denotes the value length of n. This function is exponential of input data length.

Let the definition of a pascal-like function does not contain pointers, sets, records and files as well as procedures and functions as parameters.

A pascal-like function is called **double polynomial** if both the number of steps and the run memory size are less than a polynomial under the record length of input data.

Theorem 2. [Kosovskaya 2010] The class of all double polynomial pascal-like functions equals to the class **FP**.

It means that for an effective pascal-like function it is not sufficient only a polynomial number of steps.

The above formulated theorem has the longest proof among the theorems formulated in this paper.

Let a Turbo or Visual Prolog program has no calls of the built-in predicate *concat* or of such a predicate which processes a file or a term with an uninterpreted function. Usually such a term is used as a record of a tree.

Theorem 3. The class of all functions computable by queries situated in the goal section of a Turbo or Visual Prolog program in a polynomial under summary length of input data number of steps equals to the class **FP**.

The proof of this theorem is sufficiently shorter than any proof of other theorems from this paper except theorem 1.

For functions computable by a Refal-5 program the situation is the same as for pascal-like functions.

Let any condition for application of a rule in the definition of a Refal-5 function does not contain a recursive call of the defining function. And let the definition of a Refal-5 function does not contain commands processing files.

A Refal-5 function is called **double polynomial** if both the number of steps and the run memory size are less than a polynomial under the length of its input expression.

Theorem 4. [Kosovskiy 2011] The class of all double polynomial Refal-5 functions equals to the class **FP**.

The proof of this theorem is much shorter than that of theorem 2 but essentially longer than the one of theorem 3.

Conclusion

It is well known the question whether an NP-complete problem may be computed by a Turing machine in a polynomial number of steps [Garey 1979]. The presented theorems allow to replace in such a question the notion of Turing machine by more practically used notions of programming languages.

For every regarded programming language it is possible to introduce such a complexity measure that a polynomial under the input data length of which provides the belonging of a function computable by a program to the class **FP**.

The shortest conditions were received for Turbo or Visual Prolog programs. Polynomial in time computation by a Turbo or Visual Prolog program equals to the same type computation by a Turing machine.

For more widespread programming languages such as Pascal and Refal-5 checking out the polynomial in time effectiveness needs to take in account not only the number of steps but also the run memory size.

Bibliography

[Babaev 1992] I.O.Babaev, M.A.Gerasimov, N.K.Kosovskiy, I.P.Soloviev. Intellectual programming. Turbo Prolog and Refal-5 for personal computers. St.Petersburg State University Press. 1992. (In Russian)

[Beltiukov 2006] A.P. Beltiukov , A Polynomial Programming Language. In: "Mathematical Problems of Computer Science", Transactions of the Institute for Informatics and Automation Problems of the National Academy of Sciences of Armenia, , vol.27, 2006.

[Du 2000] D.Z.Du, K.I.Ko Theory of Computational Complexity. A Willey-Interscience Publication. John Wilet & Sons, Inc. 2000.

[Garey 1979] M.R.Garey, D.S.Johnson. Computers and Intractability. A Guide to the theory of NP-Completeness. Bell Laboratories. Murray Hill, New Jersey. W.H.Freeman and Company, San Francisco, 1979.

[Kosovskaya 2010] T.M.Kosovskaya, N.K.Kosovskiy. Bases of Polynomial-Time Proof for the Simplest Mathematical Algorithms. In: Computer Instruments in Education. No 2, 2010. (In Russian)

[Kosovskiy 2011] N.K.Kosovskiy. Church Thesis for Polynomial in Time Recursive Algorithms in Words and their Lengths. In: Computer Instruments in Education. No 1, 2011. (In Russian)

Authors' Information

Nikolay Kosovskiy *– Dr., Professor, Head of Computer Science Chair of St.Petersburg State University, University av., 28, Stary Petergof, St.Petersburg, 198504, Russia, e-mail: kosov@NK1022.spb.edu*

Major Fields of Scientific Research: Mathematical Logic, Theory of Complexity of Algorithms.

Tatiana Kosovskaya *– Dr., Senior researcher, St.Petersburg Institute of Informatics and Automation of Russian Academy of Science, 14 line, 39, St.Petersburg, 199178, Russia; Professor of St.Petersburg State Marine Technical University, Lotsmanskaya ul., 3, St.Petersburg, 190008, Russia; Professor of St.Petersburg State University, University av., 28, Stary Petergof, St.Petersburg, 198504, Russia, e-mail: kosov@NK1022.spb.edu*

Major Fields of Scientific Research: Logical Approach to Artificial Intelligence Problems, Theory of Complexity of Algorithms.

The paper is published with financial support of the project ITHEA XXI of the Institute of Information Theories and Applications FOI ITHEA (www.ithea.org) and the Association of Developers and Users of Intelligent Systems ADUIS Ukraine (www.aduis.com.ua).