ONTOLOGY-BASED MODEL OF REPRESENTATION OF KNOWLEDGE ABOUT LANGUAGE MAPPINGS

Margarita Knyazeva, Vadim Timchenko

Abstract: The paper presents a short review of some systems for program transformations performed on the basis of the internal intermediate representations of these programs. Many systems try to support several languages of representation of the source texts of programs and solve the task of their translation into the internal representation. This task is still a challenge as it is effort-consuming. To reduce the effort, different systems of translator construction, ready compilers with ready grammars of outside designers are used. Though this approach saves the effort, it has its drawbacks and constraints. The paper presents the general idea of using the mapping approach to solve the task within the framework of program transformations and overcome the disadvantages of the existing systems. The paper demonstrates a fragment of the ontology model of high-level languages mappings onto the single representation and gives the example of how the description of (a fragment) a particular mapping is represented in accordance with the ontology model.

Keywords: Ontology; Language mappings base; Programming language mappings; Language mappings editor; single representation of program.

ACM Classification Keywords: I.2.5 Artificial intelligence: programming languages and software

Conference: The paper is selected from XIVth International Conference "Knowledge-Dialogue-Solution" KDS 2008, Varna, Bulgaria, June-July 2008

Introduction

Traditionally, program analysis, parallelizing and optimizations are applied to programs represented not in high-level languages but by means of different schemata and models that are suitable and convenient representations to work with source programs (e.g. Martynyuk schemata, Lavrov schemata, program representations in the form of various graphs) and are described in detail for example in the works [Voevodin, 2002] [Voevodin, 1992] [Kasyanov, 1988]. Therefore, systems designed for optimizations and parallelizing, especially those that work with several languages of source program texts, have their own single internal representations into which the analyzed program can be transformed and which is used for its further processing.

As a rule, internal representations in systems of program optimization and parallelizing are implemented in such structures as lists, trees, graphs. In some systems, e.g. Polaris [Blume, 1992], SUIF/SUIF2 [Wilson, 1994], Open Parallelizing System (OPS) [Shteinberg, 2004], the internal program representation is realized in the form of class hierarchy in an object-oriented language. The major advantages of this representation are simplicity of designing, modifiability and extensibility.

The internal program representation in OPS is a universal program information data structure and the foundation for design and analysis of information dependencies in the program, program transformations and algorithms to facilitate their execution. The system is supposed to be open to potential changes in the language of source texts due to adding or replacing relatively small programs.

Many systems for designing internal program representation, including OPS, SUIF/SUIF2, use external programs or libraries (Translator Construction Systems (TCS) ANTLR, SableCC, Sage++, Bison, YACC, compilers of Portland Group Inc.) with ready language grammars that makes them dependent on outside designers. Besides, while there is a tendency to improvement of such characteristics of TCS as (i) usability and simplicity of the translator interface that make it possible to integrate it into a software tool; (ii) usability of the description of programming language grammar; (iii) readability of the generated code, yet one may experience difficulty in using them. For example, when one describes own grammars, such tools normally impose constraints on their class and form of specifying that are conditioned by the used parsing method. Moreover, one has to make the effort to

integrate program representations with the help of foreground compilers based on such tools and own internal representations.

Thus, in many systems support of new languages is constrained by the absence of own parser (SUIF/SUIF2, Cetus) which causes regular difficulties in debugging, modifying or integrating into the system of the translator generated by the external TCS or set of classes for its implementation; or by their internal program representation being oriented to the particular language (Polaris, Parafrase).

System of construction of optimizing and parallelizing compilers (SCOPC) [Tapkinov, 2006], a structural predicate system, is an attempt to avoid those constraints. It is a program complex based on the implementation of structural predicate grammars and tools of the structural graph that is an internal program representation used in this system. The main purpose of this system is tools for developing optimizing and parallelizing compilers. Besides, the system can be used for teaching translation methods and carrying research and experiments on developing algorithms and translation methods.

The internal representation is developed for subsets of languages (C, Pascal, Fortran) and is here the result of syntactic and context analysis of the program with the help of structural predicate grammars (SP Grammar).

SP Grammar is a logic grammar in which rules are described in terms of terms and logical formulae. This language can turn out to be difficult for users of this system who are not specialists in mathematical logic. Moreover, the user has to know the internal program representation structure.

The Program Transformation System (PTS) is being developed in the Intellectual Systems Department, the Institute for Automation & Control Processes, the Far Eastern Branch of the Russian Academy of Sciences, to conduct research into program transformations. The support of an extensible set of programming languages of program source texts which must then be translated into the single intermediate representation [Artemieva, 2002] [Artemieva, 2003] is one of the requirements to this system. In order to comply with it the concept of a subsystem of generation of the single internal program representation [Kleshchev, 2007] [Knyazeva, 2008] is introduced as a solution of the problem of multilingualism in program transformations. This approach seeks to avoid the above drawbacks and restrictions in program transformations and parallelizing.

Designing a multilingual system open to new source languages requires especially flexible ways of formalizing information about them. So, to achieve this goal it was decided to use the mapping approach.

The mapping approach to translation started to be developed in early 1970s in the Research Computing Center, Lomonosov Moscow State University. The basic concepts of the approach and expressive means of appropriate description were developed by V. Sh. Kaufman [Kaufman, 1978] [Bunimova, 1978]. By mapping of the language L1 onto the language L2, Kaufman understands a mapping (in its common mathematical sense) $p : I1 \rightarrow I2$, where I1 and I2 are sets of texts acceptable in these languages. The value of the mapping is that it records the way of interpretation of L1 constructions by L2 means and is the most important part of the translator construction task and gives the freedom to choose translation algorithms as it depends on nothing but L1 and L2. Three relations must be fixed to assign the mapping from L1 onto L2: (i) relation that describes the structure of L1 texts; (ii) relation that describes the structure of L2; (iii) relation that describes the connection between those two relations.

Each relation is a system of elementary relations that are expressed in a declarative way by V-language means [Kaufman, 1977]. The significant part of this formalism is based on the apparatus of mathematical logic (predicates) and expressive means of Backus-Naur form.

The practical value of the mapping approach as a technology of constructing translators or interpreters is the design of automated TCS that must deal with a problem of how to construct a translator that implements this mapping according to a formally assigned mapping (translator specifications). The translator implements the mapping if it produces the mapped text according to the projection as a result for each text in the source language. This is the so-called problem of the solvability of mappings. Kaufman's approach did not solve this problem.

This paper presents a way of describing language mappings that can be easily used by specialists who lack mathematical training. On the other hand, it is oriented to the implementability of the above mappings, i.e. possibility in principle of constructing an interpreter that would generate this program in the single representation on the basis of the description of the mapping of a programming language onto the single internal representation

and of the program in this programming language. Thus, in this case there is no problem of mapping solvability as all the described mappings are known-solvable due to the way of representing their description.

The paper was written with financial support from the Far Eastern Branch of the Russian Academy of Sciences (initiative-based research project "Internet system for controlling information about program transformations" – 06-III-A-01-007) and by the Russian Foundation for Basic Research (project 'Research into collective control in the semantic web of information resources of different levels of commonality' – 06-07-89071-a).

Implementation-oriented approach to description of language mappings. The main ideas.

The first idea of the implementation-oriented approach to the description of language mappings is to develop and describe the ontology model of a programming language for which its mapping onto the single internal representation is written.

The programming language ontology is a set of information that describes a set of concepts of this language and interrelations between these concepts, i.e. the way of uniting them into language constructions. It can be said that the language ontology defines the abstract syntax of this language. The model of the language ontology is represented by the semantic network of concepts connected with each other by directed arcs and provided with a special markup.

The second idea is to describe the connection between the language ontology and concrete syntax of this language that defines the content of the correct language constructions from the point of view of the syntax of this language. The connection between the language ontology and the concrete syntax of this language contains such language elements as the punctuation content, defines the lexeme order and so on. This kind of information restricts the way of expressing the sense which is in the abstract program representation in the form of a text.

These models are necessary for the parsing of program texts and for representations of these programs in the abstract syntax. The program in the abstract syntax is a program represented in terms of the programming language ontology that does not contain elements of the concrete syntax.

The third idea is to develop and describe the model of ontology of programming language mappings onto the single representation in accordance with which each mapping is described.

The above formalization makes it possible to develop the conception of subsystem of generation of the single internal program representation that consists of easily customizable to the correspondent model of the ontology of program components and ontologies controlled by the correspondent models that can be changed if necessary [Kleshchev, 2007] [Knyazeva, 2008].

Model of the ontology of language mappings onto the single representation

By the projection, the following mapping is understood:

$$\mathbf{P}: C \rightarrow E, \text{ where}$$

C = {concepts} – a set of concepts of the programming language in terms of which the program is represented;

E = {elements} – a set of elements of the single representation language.

Below is the fragment of the ontology model of programming language mappings onto the single representation. The model is a specification of the abstract syntax of the language of the mapping description. The description of the operational language semantics which is oriented to an interpreter is not represented here, but its informal description for the user is given.

To describe we used the following symbols of the language of specifications which is used in the work [Yershov, 1977]: **unit** – uniting; [a] – a is not necessary (can be absent); = – concept definition; **:** – position opening, * – possibly indefinite attribute; **ser** – serial component.

Mapping description language = (**ser** programming language **:** Mapping description)

Description: Mapping description language specifies the rules of description of mapping of the assigned programming language onto the single program representation.

Semantics:

Mapping description = (**ser** correspondence **:** Correspondence description)

Description: Mapping description specifies a set of correspondences of the concepts of the assigned programming language with the constructions of the single program representation language.

Semantics:

Correspondence description = (programming language concept **:** STRING, construction of the single representation language **:** Elements of the single representation language)

Description: In the correspondence the structure of the construction of the single representation language that must be correspondent to the concept of the source programming language, i.e. the concept of the programming language is correspondent to a set of elements of the single representation language structured in a specific way.

Semantics: To interpret the description of the structure of the construction of the single representation language in this correspondence for the specified concept of the programming language.

Elements of the single representation language = ([fragments **:** Set of fragments], [control arcs **:** Set of control arcs], [attributes **:** Set of attributes])

Description: The construction of the single representation language consists of sets of fragments, control arcs and attributes that can be empty empty.

Semantics: To develop the construction of the single representation language with the specified structure.

Set of fragments = (**ser** fragment **:** Fragment class)

Fragment class = **unit** (*Expression*, *Program block*, *Conditional statement*, *Loop_with_step, Loop_ with_precondition, Loop_with_postcondition, Procedure_call, Dynamic_variable_elimination*, *Assignment, Input, Output*, *Description_of_one_variable*,

Description_ of_one_function,

Description_of_one_parameter, *Block_of_descriptions_of_variables, Block_of_descriptions_of_functions, Block_of_descriptions_of_parameters*, Name of other fragment class)

Name of other fragment class = (name **:** STRING)

Description: *Set of fragments* is a set of names of fragment classes (they are represented by a component *Fragment class*) specified in the single representation language. If during the mapping description there are not enough names of fragment classes specified in the core of the language of names of fragment classes, one can define a new fragment class (by specifying its name).

Thus defined fragment classes are added to the specialized (for the particular programming language) extension of the single representation language which, if necessary, is specified during the mapping description of the programming language onto the single representation.

Semantics: To gradually create fragments a set of names of which is defined by a component *Set of fragments*.

Set of control arcs = (**ser** control arc : Control arc)

Description: *Set of control arcs* is a set of control arcs.

Semantics:

Control arc = (arc name : **unit** (*If, Then, Else, Condition_of_loop, Starting_boundary_of_loop, End_boundary_of_loop, Step, Body, Description_of_parameters*, *Description_of_variables*, *Description_of_functions*, *Right_expression, Left_expression, Parameter_list, Fragment_ancestor,* Other name of control arc), fragment_arc start point : **unit** *(Program_block, Conditional_statement, Loop_with_step, Loop_with_precondition, Loop_with_postcondition, Procedure_call, Dynamic_variable_elimination, Assignment, Input, Output*, *Description_ of_one_function*,

Block_of_descriptions_of_variables, Block_of_descriptions_of_functions, Block_of_descriptions_of_parameters, Name of other fragment class),

fragment-arc end point : **unit** (*Expression, Program_block*, *Conditional_statement, Loop_with_step, Loop_with_precondition, Loop_with_postcondition, Procedure_call, Dynamic_variable_elimination, Assignment, Input, Output*, *Description_of_one_variable*, *Description_ of_one_function*,

Block_of_descriptions_of_variables, Block_of_descriptions_of_functions, Block_of_descriptions_of_parameters, Name of other fragment class))

Another name of control arc = (name **:** STRING)

Description: *Control arc* is a directed arc that is characterized by its name and connects two fragments. The arc start point is a fragment defined by the selector "fragment_arc start point". The arc end point is is a fragment defined by the selector "fragment_arc end point".

If during the mappping description there are not enough control arcs specified in the core of the language of control arcs, one can define a new control arc by specifying its name – a component *Name of control arc* defined by the selector "arc name", and initial and finite fragments which should be connected by it.

Thus defined control arcs are added to the specialized (for the particular programming language) extension of the single representation language which, if necessary, is specified during the mapping description of the programming language onto the single representation.

The component defined by the selector "fragment_arc start point" can be one of the mentioned fragments. The component defined by the selector "fragment_arc end point" can be one of the mentioned fragments.

Semantics: To create a control arc with the name defined by the selector *arc name*, make a fragment defined by the selector *fragment_arc start point* the initial fragment of the control arc and a fragment defined by the selector *fragment_arc end point* – the finite fragment of the control arc.

If a fragment defined by the selector *fragment-arc end point* does not exist (has not been created) on this interpretation step of correspondence description, it will be created on the basis of description of the next interpreted correspondence and assigned as the finite argument of this control arc.

Set of attributes = (**ser** attribute : Attribute)

Description: *Set of attributes* is a set of attributes.

Semantics:

Attribute = (attribute name : **unit** (*Identifier, Declaration_statements, Loop_counter, Left_part_type, Left_part_of_expression, Right_part_type, Variable_not_parameter, Right_part_of_expression, Is_function, Operation_symbol, By_value, Is_array, Is_pointer, Address_expression, Variable_result_of_function, Original_string_name, Type, Lower_bound_of_array, Upper_bound_of_array,* Another name of attribute),

argument of attribute : **unit** (*Expression, Program_block, Conditional_statement, Loop_with_step, Loop_with_precondition, Loop_with_postcondition, Procedure_call, Dynamic_variable_elimination, Assignment, Input, Output*, *Description_of_one_variable*, *Description_ of_one_function*, *Description_of_one_parameter*

Name of other fragment class),

computable : LOG,

[attribute value : Value])

Another name of attribute = (name : STRING)

Value = **unit** (STRING, INTEGER, REAL, LOG, Fragment class)

Description: *Attribute* is characterized by its name, argument for which it is defined and value it can get. Besides, it can be computable or non-computable.

The argument of the attribute can be one of the mentioned fragments. The attribute value is a value of string, integer, real, logical type or it can be represented by the component *Fragment class.* "Computable" means either that it is necessary to compute the value of an attribute in the process of program scanning in the programming language or it is not necessary because the attribute will assume the value specified directly during the mapping description.

If it is necessary to compute the value of an attribute, it is supposed that this information is always directly accessible (without using any program flow analysis methods) during the lexico-syntactical analysis of the source program.

If during the mapping description there are not enough attributes specified in the core of the language, one can define a new attribute by specifying its name – a component *Name of attribute* defined by the selector "attribute name", argument defined by the selector "attribute argument", meaning of indicator about the necessity of computing the value defined by the selector "computable", and the value defined by the selector "attribute value".

Thus defined attributes are added to the specialized (for the particular programming language) extension of the single representation language which, if necessary, is specified during the mapping description of the programming language onto the single representation.

Semantics: To create an attribute with the name defined by the selector *attribute name* the argument of which is defined by the selector *attribute argument*, to assign the component *Value* defined by the selector *attribute value*.

If the value if an attribute defined by the selector computable is true, the value defined by the selector *attribute value* must be computed at the stage of program scanning in the programming language.

Fig.1, fig. 2 and fig. 3 demonstrate the examples of correspondence description between Pascal concepts, such as “Expression”, “Assignment statement”, “Variable name”, and constructions of the single representation language corresponding to them.

These descriptions are a fragment of description of the mapping of the Pascal programming language onto the single internal representation and are made in compliance with the above ontology model.

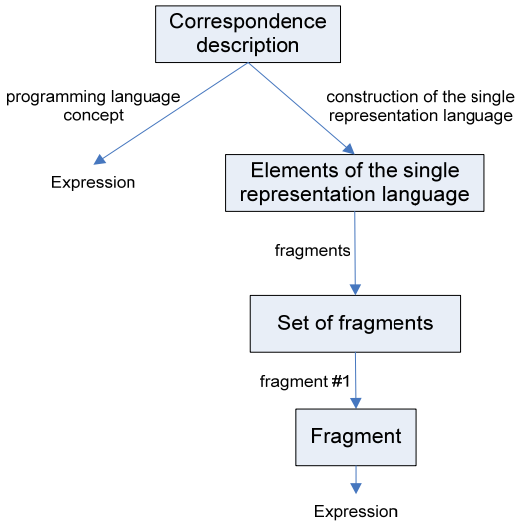

Fig. 1. Description of correspondence between Pascal concept “Expression” and construction of the single representation language.

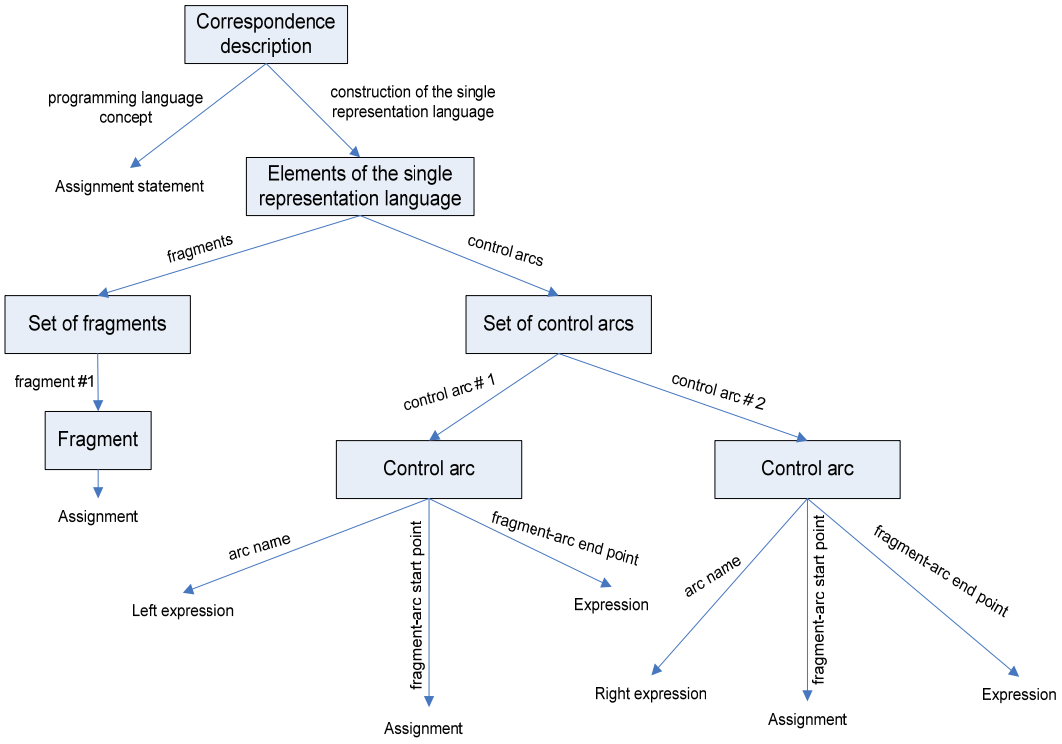

Fig. 2. Description of correspondence between Pascal concept “Assignment statement” and construction of the single representation language.

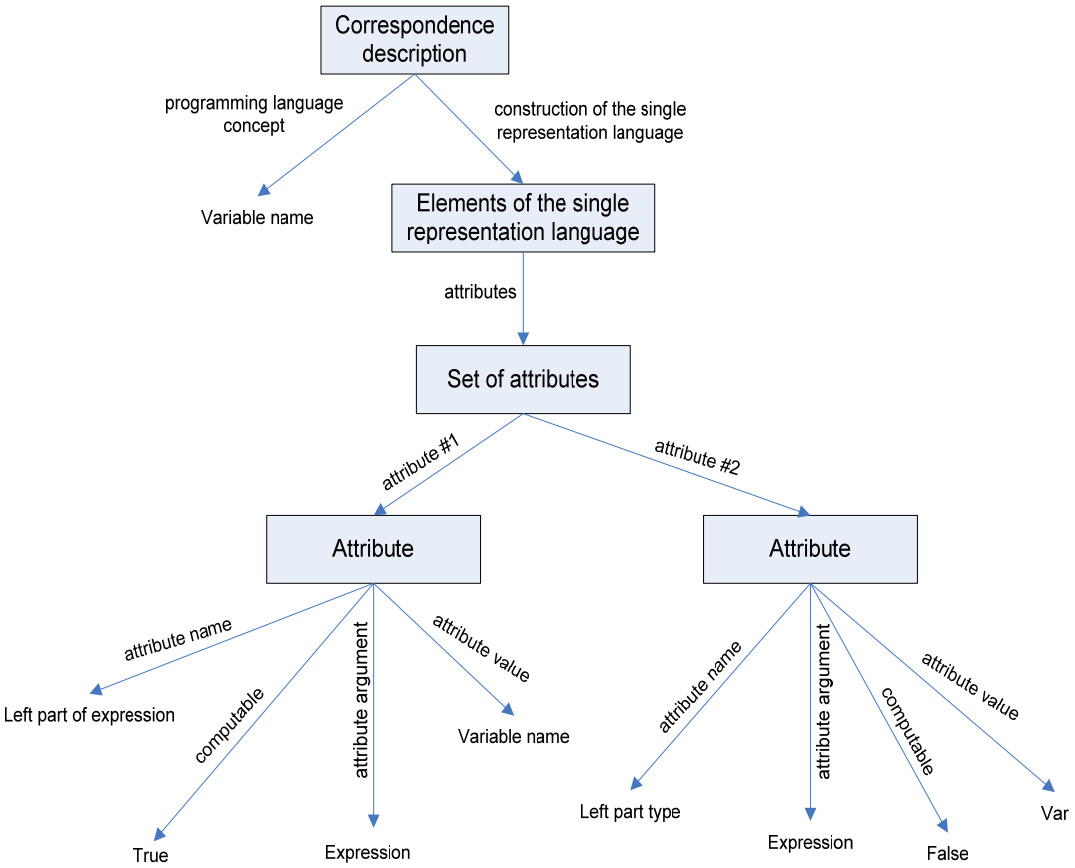

Fig. 3. Description of correspondence between Pascal concept "Variable name" and construction of the single representation language.

Conclusion

The paper presents a short review of some systems in which analysis, optimization and parallelizing are performed on the basis of the internal intermediate representation of these programs. Many such up-to-date systems support several languages of representation of source program texts and, thus, solve the task of their translation into the internal representation using some approaches. Being quite a challenge, this task is complicated by the fact that to make the internal representation as usable and efficient as possible for the flow analysis, optimization and parallelizing becomes more and more important and the translation itself is considered as a subordinate task that can be done with the help of ready tools of outside designers. Thus, solving this problem is still connected with some difficulties and the approaches, being closer and closer to the solution, still have their drawbacks and constraints. The paper also deals with the general idea of using the mapping approach to solve the problem of multilingualism in PTS that can help overcome disadvantages of the existing systems. The paper demonstrates a fragment of the ontology model of high-level languages mappings onto the single representation and gives the example of how the description of (a fragment) a concrete mapping is represented in accordance with the ontology model.

Bibliography

[Voevodin, 2002] Voevodin V.V., Voevodin Vl.V. Parallel computing. Saint Petersburg: BHV-Peterburg, 2002. (In Russian).

[Voevodin, 1992] Voevodin Vl.V. Theory and practice of research on parallelism of sequential programs // Programmirovanie. 1992. #3. P. 38-54. (In Russian).

[Kasyanov, 1988] Kasyanov V. N. Optimizing transformations of the programs. Moskow: Nauka, 1988. (In Russian).

[Blume, 1992] Blume W., Doallo R., Eigenmann R. a. o. Parallel programming with Polaris // Computer. – 1992. – Vol.29, N 12. – P. 78 – 82.

[Wilson, 1994] Wilson R.P., French R.S., Wilson C.S., a.o. SUIF: An infrastructure for research on parallelizing and optimizing compilers // SIGPLAN Not. – 1994. – Vol.29, N 12. – P. 31-37.

[Shteinberg, 2004] Shteinberg B.Ya. Open parallelizing system // Mathematical methods of parallelizing of recurrent loops for upercomputers with parallel memory - Rostov-on-Don: Rostov University, 2004.- P. 166-182. (In Russian).

[Tapkinov, 2006] Tapkinov B.Yu. Internal program representation in System of construction of optimizing and parallelizing compilers // Proceedings of All-Russian Scientific Conference "Scientific service in the Internet: parallel programming technologies", Novorossiysk, 18-23 September 2006. P. 88-91. (In Russian).

[Artemieva, 2002] Artemieva I.L., Knyazeva M.A., Kupnevich O.A. A model of a domain ontology for "Optimization of sequential computer programs". Terms for optimization process description. In 3 parts. // Scientific & Technical Information. Part 1: 2002. №12: 23-28. (In Russian).

[Artemieva, 2003] Artemieva I.L., Knyazeva M.A., Kupnevich O.A. A model of a domain ontology for "Optimization of sequential computer programs". Terms for optimization process description. In 3 parts. // Scientific & Technical Information. Part 2: 2003. №1: 22-29. (In Russian).

[Kleshchev, 2007] Kleshchev A.S., Knyazeva M.A., Timchenko V.A. Smart system of generation of single internal representation in system of program transformations. Second International Conference "System analysis and information technologies" : Conference proceedings. In 2 vol. Vol. 1. – M.: Editorial URSS, 2007. – 288 p. (In Russian).

[Knyazeva, 2008] Knyazeva M.A., Timchenko V.A. Subsystem of generation of single internal representation in system of program transformations. // Software & Systems. - 2008.- №1. (In Russian).

[Kaufman, 1978] Kaufman V.Sh. On technology of translator construction (mapping approach). // "Programmirovanie". 1978. № 5. – P. 36 – 44. (In Russian).

[Bunimova, 1978] Bunimova E.O., Kaufman V.Sh., Levin V.A. On description of language mappings. – "Vestnik MGU. Computational Mathematics and Cybernetics". 1978. №4. – P. 68 – 73. (In Russian).

[Kaufman, 1977] Kaufman V.Sh., Levin V.A. Natural approach to the problem of description of context conditions. – "Vestnik MGU. Computational Mathematics and Cybernetics". 1977. №2. – P. 67 – 77. (In Russian).

[Yershov, 1977] Yershov A.P., Grushetsky V.V. Realization-oriented method of description of algorithmic languages. Preprint, Computational Center of Siberian Branch of Academy of Sciences of the USSR, Novosibirsk, 1977. (In Russian).

Authors' Information

Margarita A. Knyazeva – Ph.D. Senior Researcher of the Intellectual System Department, Institute for Automation & Control Processes, Far Eastern Branch of the Russian Academy of Sciences: 5 Radio Street, Vladivostok, Russia; e-mail: rita_knyazeva@mail.ru

Vadim A. Timchenko - Researcher; Institute for Automation & Control Processes, Far Eastern Branch of the Russian Academy of Sciences: 5 Radio Street, Vladivostok, Russia; e-mail: rakot2k@mail.ru