TOOLS AND TECHNIQUES OF ANNOTATED PROGRAMMING

VICTOR N. KASYANOV

Academy of Sciences of the U.S.S.R
Siberian Branch
Institute of Informatics Systems
Novosibirsk 630090, U.S.S.R.

Annotated programming is a method of program processing which takes into account program application information a priori known and conveyed in annotations. A model for annotated programming is described within whose framework many kinds of practical work with programs (e.g. partial evaluation and optimization) can be performed. A transformation machine concept and some tools for annotated program transformations are considered.

Introduction

Transformation techniques are gaining in importance for both theoretical and technological programming. Systems of equivalent transformations have been conventionally used in the optimizing compilers [1-3] and are currently widely applied in mechanical aids for supporting the program development process [4,5]. The long-range objective of program transformation paradigm is to essentially improve the construction, reliability, maintenance and extendibility of software. The current state-of-the-art of program transformation is still rather far from supporting these ambitious goals, and research continues along a variety of diverse paths [6].

In the paper, we outline transformational approach to program concretization, whereby a given general-purpose program can be correctly transformed into multitude of more qualitative special-purpose programs. A concretization transformation is aimed at improving a given program without disturbing its correctness in a given restricted and stable context of its applications. In addition to the restricted sets of program inputs and outputs some suitable criterion of program quality can be defined by program application context. For example,

memory, time or reliability may be considered as program quality criteria by the context given.

According to the approach presented, any source program is considered as a base for constructions of a number of different specialized programs. Every construction starts with the source program and an application context conveyed in formalized comments (annotations). Some program annotations can be formed in parallel with the development of the source program, others are added by users and describe a specific context of source program applications. Then a series of concretizing transformations is applied to the annotated general-purpose program (either automatically or interactively with the user), which results in a correct and qualitative specialized program.

A well-known example of program specialization is the so-called partial evaluation (or mixed computation) of programs on partially given inputs [7]. Partial evaluation can be applied to compiling, program generation, including compiler generation and generation of a compiler generator, and metaprogramming without order-of-magnitude loss of efficiency [8].

Concretization problem

Investigations of transformation systems and their applications to various kinds of program manipulations show that during performing transformations it is important to take into account information known about application context of program transformed, as well as to employ generalizing and specializing transformations which are nonequivalent.

Unlike the equivalent transformations that preserve the functions calculated by programs transformed, generalizing transformation can convert a source program to such a result one that solves a more general problem than the source program (for example, function calculated by result program can be obtained from source program function by the addition of further parameters or results). Specialization is in some ways the complement of generalization. A well-known example of specializing transformation is the so-called partial evaluation (or mixed computation) of programs on partially given inputs [7,8].

Similar to distinguishing optimizing transformations among all equivalent ones it is possible to distinguish among generalizing and specializing transformations the so-called concretizing transformations aimed at optimization of source program in a restricted and stable context of program applications [9]. Every concretizing transformation is aimed at improving the program given according to a given qualitative criterion (e.g. memory, time or reliability) without disturbing the meaning of the program in a given restricted context of its application.

Let us illustrate the concretization problem with an example of a simple Pascal-procedure E1 which computes in X the solution of linear equation system represented by a triangular matrix A and a vector B

```
PROCEDURE E1 (A:MATRIX; B:VECTOR; VAR X:VECTOR);
  VAR I,K:INTEGER; Z:REAL;
  BEGIN X[1]:=B[1]/A[1,1]; FOR I:=2 TO N DO BEGIN Z:=0;
      FOR K:=1 TO I-1 DO Z:=Z+A[I,K]*X[K]; X[I]:=(B[I]-Z)/A[I,I]
  END END.
```

If E1 deals only with a diagonal matrix A and qualitative criterion is a program length then the procedure can be replaced by the following improved version of E1

```
PROCEDURE E2 (A:MATRIX;B:VECTOR; VAR X:VECTOR);
   VAR I:INTEGER; BEGIN FOR I:=1 TO N DO X[I]:=B[I]/A[I,I]  END.
```

In another context, if the single goal of any application of E1 is to compute the first element of X, the following version of the procedure E1 is more qualitative with respect to all main criteria of program quality

```
PROCEDURE E3 (A:MATRIX; B:VECTOR; VAR X:VECTOR);
   BEGIN X[1]:=B[1]/A[1,1] END.
```

It should be noted that to our time most needs of concretizations are satisfied by using such universal tools of program text construction as macro generators and editors. But the approach to automatization of concretizations is not convenient for programmers because it makes high demands to programmers. Under the approach an end user must programme all specialization processes of its own program.

Concretization and compiling

Assume that a given program, P, is to be run repeatedly on a range of inputs to produce a range of outputs. Similar to partial evaluation [8] we consider two-stage process:

- at the first stage a context of applications of P is given to produce a specialized program which is equivalent to P on the ranges of its inputs and outputs and is better than the original P by the quality measure given by the context ,
- at the second stage the specific data values in the input range is given to produce the results from the output range.

Note that in partial evaluation at the first stage only a part of the inputs is given, and partial evaluation can be used for compiling, program generation (including compiler generation), metaprogramming without order-of-magnitude loss of efficiency. So, applications of program concretization include partial evaluation and its applications as well.

Let us consider some examples of the application.

General context-free parsing algorithms (e.g., Earley's parser) are notoriously slow, whereas parsers for specific grammars are efficient enough to be a standard part of modern compiler technology. Concretization of a general context-free parser in context of a particular context-free grammar can, at least in principle, make dramatic improvements in efficiency.

Suppose an interpreter I for some language L is given. The input to I is a program P and an input data to P. The result of concretizing I to the given program P (i.e. in the context of applications defined by P as input program of I) will be a program that takes the same input data as P and is equivalent to P . So, the concretization will translate the program P from the language L into the language output by the concretizator.

Concretizators are program systems that treat programs as data objects and can be used also to generate a program generator automatically from a given general-purpose program.

Suppose now that the concretization process itself can be programmed, so there is a program S which transforms any input pair < a program P, a context C > into specialized version P_C of the program P for the context C. Let also that S will be used

only to concretize the same fixed program Q as P, regardless of the the value of C. Then the result of concretizing the concretizator S is the program S_Q that transforms a context C for Q into Q_C. For example, the following three cases are possible:

- parser generator S_Q, if Q is a general parser, C is a context-free grammar G and Q_C is a parser for $L(G)$,
- compiler generator S_I, if Q is interpreter for some language L, C is a source program in the language L and I_C is a target program,
- generation of a compiler generator S_S, if Q is the concretizator S, C is an interpreter S for some language L, Q_C is compiler for the language L interpreted by I.

Annotated programs

The main idea of concretization is to take advantage of the known context in which only some program executions are admissible and only some of their results are used to tailor that program to the context, with the objective of realizing a more qualitative (in the meaning defined by the context) computation of the used results. But modern high-level languages do not have enough means of description of contexts of program applications.

So, it is natural to pass from program to program with annotations in which context information can be conveyed [9].

As a basic language let us consider a high-level language, for example, Pascal. The basic language is assumed to be extended by adding the annotations which are formalized comments in the basic programs and relevant for the semantics of the program annotated. In particular, every annotation-assertion is evaluated and if it is false, the execution is inadmissible (beyond the context of program applications). So, annotations-assertions are intended to state certain properties of the program at its particular "places", and these properties can be used for solving problems of program concretization.

For example, the annotated procedure

```
PROCEDURE E4;
```

```
BEGIN {$ Z=1; W=2; $}
   IF X>0 THEN Z:=Z+1 ELSE BEGIN Y:=X+1; Z:=W END
   {$ DEAD(Y); $}
END;
```

where DEAD(Y) sets that the current value of Y is unused under any application of E4 can be correctly transformed into

```
PROCEDURE E5;  BEGIN Z:=2 END;
```

It is assumed that the following properties hold.

Annotations added to a basic program specify a covering context. It is guaranteed that any actual application from the context described will be admissible by annotations, but some admissible applications may be beyond the actual context.

Annotated programs are subjected to concretizing transformations as a whole. It means that the transformations can change not only the basic program but their annotations as well.

Annotations intended to specify the context can be represented in the form of directives as well. Unlike assertion being predicate constraints on admissible memory states, annotation-directive can be either a statement that will change current memory state every time the annotation is reached during possible execution of the program annotated [10] or name of a concretizing transformation allowed for application at the corresponding annotated program point by the context [3].

Below an example of annotated Pascal-function which computes $\sum (-1)^i x^{2i+1}/(2i+1)!$ is presented. The example illustrates how assertions and directives can be used to form tracing algorithm which gathers some information about program execution to verify its correctness.

```
FUNCTION E6 (X,E:REAL):REAL;
   VAR A,B,C,D,S:REAL; {$ I:INTEGER; $}
   {$ FUNCTION P (X:REAL;N:INTEGER):REAL;
      BEGIN IF N=0 THEN P:=1 ELSE P:=P(X,N-1)*X END;
   FUNCTION F (N:INTEGER):INTEGER;
      BEGIN IF N=0 THEN F:=1 ELSE F:=F(N-1)*N END;
   FUNCTION ELEM (X:REAL; N:INTEGER):REAL;
      BEGIN ELEM:=P(-1,N)*P(X,2*N-1)/F(2*N-1) END;
   FUNCTION SUM (X:REAL;N:INTEGER):REAL;
      VAR I:INTEGER; S:REAL;
```

```
   BEGIN S:=0; FOR I:=1 TO N DO S:=S+ELEM(X,I);  SUM:=S  END;
$}
BEGIN A:=-2; B:=0; C:=X; S:=X; D:=-SQR(X); {$ I:=0; $}
   WHILE ABS(C)>E DO
      BEGIN {$ A=S*I-2; B=2*I*(2*I+1); C=ELEM(X,I);
         S=SUM(X,I); ABS(C)>E; $}
         A:=A+8; B:=B+A; C:=C*D/B; S:=S+C; {$ I:=I+1; $}
      END; {$ S=SUM(X,I); ABS(ELEM(X,I))<=E; $} E4:=S
END;
```

Model for annotated programming

A program model described below is based on large-scale program schemata that covers a broad class of programs and their transformations [5, 6].

Let $S = \{s\}$ be a set of memory states such that for any state $s \in S$ a partition of the set of all variables $V = \{v\}$ into two sets $A(s)$ and $I(s)$ of *accessible* and *inaccessible* variables, respectively, is given and for every accessible $v \in A(s)$ its value $s(v)$ is defined. Let s^1 and s^2 be two memory states. s^1 and s^2 are *equal on a set* of the variables $W \subseteq V$ if for any $v \in W$ either $s^1(v) = s^2(v)$ or $v \in I(s^1) \cap I(s^2)$. s^1 *expands* s^2 (denoted by $s^2 \leqslant s^1$) if s^1 and s^2 are equal on the set $A(s^2)$.

A *program* π is a tuple (g,f,p,r,a,d) which consists of

(1) a *flowgraph* $g = (X,U,x_0,y_0)$, where $x_0 \in X$ is the *entry* statement having no ingoing arcs (i.e. $IN(x_0) = \emptyset$) and only one outgoing arc denoted by u_0 (i.e. $OUT(x_0) = \{u_0\}$) and $y_0 \in X$ is the *exit* statement having no outgoing arcs (i.e. $OUT(y_0) = \emptyset$), and for every arc $u = (x^1,x^2) \in U$ the functions **source**$(u) = x^1$ and **target**$(u) = x^2$ are defined;

(2) a function of *memory transformation* $f : X \Rightarrow (S \Rightarrow S)$;

(3) a function of *control transfer* $p : X \Rightarrow (S \Rightarrow U)$;

(4) *argument* and *result* functions $a,r: X \Rightarrow (S \Rightarrow 2^V)$;

(5) *applicability* predicate $d : X \Rightarrow (S \Rightarrow \{$**true**,**false**$\})$,

such that for any $v \in V$, $x \in X$ and $s,s^1,s^2 \in S$ the following properties hold:

(1) the memory states s and $f(x)(s)$ are equal on the set $V \setminus r(x)(s)$;

(2) if s^1 and s^2 are equal on $a(x)(s^1)$ then $d(x)(s^1) = d(x)(s^2)$, $p(x)(s^1) = p(x)(s^2)$, $a(x)(s^1) = a(x)(s^2)$, $r(x)(s^1) = r(x)(s^2)$ and the memory states $f(x)(s^1)$ and $f(x)(s^2)$ are equal on the set $r(x)(s^1)$;

(3) if $I(s) \cap a(x)(s) \neq \emptyset$, then $d(x)(s)$ is **false**;

(4) $a(x_0)(s) = A(f(x_0)(s))$, $a(y_0)(s) = \emptyset$ and for every $x \in \{x_0, y_0\}$ the memory states s and $f(x)(s)$ are equal on the set $A(f(x)(s))$;

(5) $p(x)(s)$ is an arc u outgoing from x, i.e. **source**(u) = x.

In other words, for any x the functions $f(x)$, $a(x)$, $r(x)$, $p(x)$ and $d(x)$ describe semantics of the statement x. The execution of x at a given memory state s terminates normally if $d(x)(s)$ is **true**, and results in defining the new memory state $f(x)(s)$ and the new executed statement **target**$(p(x)(s))$. During the execution current values of variables $v \in a(x)(s)$ are used and new values are assigned to variables $v \in r(x)(s)$. For example, if x is the statement IF V1> 0 THEN V2:=1/(V3-1) ELSE V4:=1/(V5+1) and s^1 and s^2 are two memory states such that $s^1(V1)>0$, $s^1(V3)=1$, $s^2(V1) \leqslant 0$ and $s^2(V5) \neq -1$, then $a(x)(s^1) = \{V1, V3\}$, $r(x)(s^1) = \{V2\}$, $a(x)(s^2) = \{V1, V5\}$, $r(x)(s^2) = \{V4\}$, $d(x)(s^2)$ is **true** and $d(x)(s^1)$ is **false**.

The program π computes function $\pi : S \Rightarrow S$ defined by the following rules. The value of the function for a given memory state s_1 is *defined* (π is *applicable* to s_1) and equal to s_2 if there is a finite sequence **seq**$(\pi, s_1) = (x_0 = x^0, s^0, u^0, x^1, s^1, u^1, \ldots, x^n, s^n, u^n, x^{n+1} = y_0)$ called an *execution sequence* of π on s_1 such that $s^0 = f(x_0)(s_1)$, $s_2 = f(y_0)(s^n)$ and for any i the following properties hold: $d(x^i)(s^{i-1})$ is **true**, $s^i = f(x^i)(s^{i-1})$, $u^i = p(x^i)(s^{i-1})$ and $x^i =$ **target**(u^{i-1}). If there is no finite execution sequence **seq**(π, s_1) then π is *inapplicable* to s_1 and the value $\pi(s_1)$ is *undefined*. If the value $\pi(s_1)$ is defined then the variables $v \in a(x_0)(s_1)$ are called the *arguments* of $\pi(s_1)$.

Let π^1 and π^2 be two programs. π^1 *generalizes* π^2 if for any memory state s^2 which π^2 is applicable to, there is such a memory state s^1 that s^1 is equal to s^2 on the set of arguments of $\pi^2(s^2)$, π^1 is applicable to s^1 and $\pi^2(s^2) \leqslant \pi^1(s^1)$. π^1 and π^2 are *equivalent* programs if they compute the same function.

Let a nonempty set of objects called *annotations* be given. It is assumed that the set is divided into two disjoint subsets: *assertions* E = {e} and *directives* Q = {q}. Every assertion $e \in E$ is a predicate on S. A memory state s is said to be *admissible* with respect to e, denoted $s \leq e$, if e(s) is **true**. It is assumed that E contains minimum and maximum elements $\perp$ and $\top$ such that any memory state is admissible with respect to $\top$ and inadmissible with respect to $\perp$. Every directive $q \in Q$ is a statement on S. In other words, the functions f, a and r and the predicate d of any program are extended on the set Q. It is assumed that Q contains an *identity* directive q_0 such that $q_0(s)=s$, $a(q_0)(s)=r(q_0)(s)=\emptyset$ for any $s \in S$, and for any $s \in S$ and any $W \subseteq V$ there is such a directive $q_{s,W} \in Q$ that for all $s^1 \in S$ the following three properties hold: $a(q_{s,W})(s^1)=\emptyset$, $r(q_{s,W})(s^1)=W$ and s and $f(q_{s,W})(s^1)$ are equal on W .

Annotated program π^1 is a triple (π,m,t) where π is a program on which π^1 is *based*, m and t are annotating functions which attach to every arc u of π some assertion $m(u) \in E$ and directive $t(u) \in Q$. Like basic programs, any annotated program π^1 computes a function $\pi^1: S \Rightarrow S$. The function is defined by the following rules. For a given $s_1 \in S$ the value $\pi^1(s_1)$ is *defined* (π^1 is *applicable* to s_1) and equal to s_2 if there is a finite execution sequence $\mathbf{seq}(\pi^1,s_1) = (x^0=x_0, \bar{s}^0, u^0, \tilde{s}^0, x^1, \bar{s}^1, u^1, \tilde{s}^1, \ldots, x^n, \bar{s}^n, u^n, \tilde{s}^n, x^{n+1}=y_0)$ such that $\bar{s}^0 = f(x_0)(s_1)$, $s_2 = f(y_0)(\tilde{s}^n)$ and for any i the following properties hold: $d(x^i)(\tilde{s}^{i-1})$ and $d(t(u^i))(\bar{s}^i)$ are **true**, $\bar{s}^i = f(x^i)(\tilde{s}^{i-1})$, $u^i = p(x^i)(\tilde{s}^{i-1})$, $x^i = \mathbf{target}(u^{i-1})$, $\bar{s}^i \leq m(u^i)$ and $\tilde{s}^i = t(u^i)(\bar{s}^i)$. Thus, the equivalence and generalization relations are defined on the set of all annotated and basic programs.

For example, the annotated function

```
FUNCTION POWER1 (X:REAL; N:INTEGER):REAL;
   BEGIN  {$ N:=5; $}  Y:=1;
      WHILE N>0 DO BEGIN WHILE NOT ODD(N) DO
        BEGIN N:=N DIV 2; X:=SQR(X)  END; N:=N-1; Y:=Y*X
        END; POWER1:=Y
   END
```

is equivalent to the basic function

```
FUNCTION POWER2 (X:REAL; N:INTEGER);
```

```
   BEGIN Y:=SQR(SQR(X))*X; POWER2:=Y END
```
generalizes the annotated function
```
FUNCTION POWER3 (X:REAL;N:INTEGER);
   BEGIN {$ N=5; $} POWER3:=SQR(SQR(X))*X END
```
but is not equivalent to it.

Transformation machine for program concretization

The class of correct transformations of annotated programs covers various kinds of work with basic programs. It contains both all equivalent transformations and a number of such nonequivalent transformations which specialize or generate a basic program to be transformed, in particular partial evaluation.

So, the approach permits specializing and generalizing transformations of basic programs to reduce to equivalent transformations of annotated programs and to employ for their investigation equivalent transformation techniques developed in terms of program schemata theory [12].

Another advantage of the approach outlined above is the possibility to perform global transformations of basic programs by iterative application of elementary transformations of annotated programs.

To construct annotated program transformation tools, we may make use of the concept of an abstract device which has elementary transformations as its instruction set and is called a transformation machine [13].

Various processes of correct transformations of annotated programs seem to have a relatively small number of underlying elementary transformations being correct in the class of all annotated programs. Thus, it is possible to develop a transformation machine (TM), whose data and instructions are the annotated programs and their transformations, respectively [14]. Transformations used as TM instructions are of the three types: (1) instructions for moving active points about the programs processed; they make one or few points of the program accessible for transformations; (2) control instructions to express higher level transformation rules in terms of lower ones; (3)

elementary transformations which are rules of correct transformations of annotated programs which alone are able to modify the program processed.

Thus unlike the transformation machine described in [13], TM employs no instructions whose application correctness depends not only on the fragment transformed , but on program as a whole. So, every program in the TM instruction language is a program processor,i.e. it defines a correct transformation of any annotated program.

The set of all elementary transformations of TM is subdivided into four subsets: property and schematic transformations to be outlined below, elementary transformations which reflect the semantics of language constructions (e.g. CASE const OF const: statement; sequence END => statement) and elementary transformations that originate from object domain laws (e.g., 1+2=>3; exp*1=>exp; exp/0 =>error-division-by-zero).

The subset of the schematic transformations includes removing and inserting inaccessible fragments; removing and inserting useless computations; replacing the terms according to their properties; replacing the variables; deadlock standardization; copying the fragments and pasting copies together; folding and unfolding for functions and procedures,removing and inserting unessential branches.

Property transformations are intended to generate new annotations by extracting information from a basic program constructions, to propagate information taking into account the property modification which originates from a relevant language construction and to update annotations through the new information logically inferred from current annotations.

The transformation implemented by TM can be either applied automatically or as programmer-guided manipulation of annotated programs. This process may involve significant system-programmer interactions.

TM instruction language also allows writing procedures to define more complex rules in terms of elementary ones and contains a set of built-in procedures. For example, there are built-in procedures for data flow analysis for the extraction of such properties as equality of terms, ranges of variables and a

number of properties which can be described by finite sets of predicates. Different strategies of program transformations can be expressed in the instruction language as procedure with transformations being formal parameters. For example, there are built-in procedures to realize algorithms of data flow analysis, to convert various constructions of annotated program into canonical forms, for logical inference and so on.

Instruction set of TM must be extensible. But programmer must be able to prove the correctness of added basic transformations. So, there is a great interest in constructing such a metamechanism which assists the programmer with extension of instruction set of TM.

Tools for program concretization

The transformation approach described above enables us to construct program transforming tools of various types. An example is a program transformer that realizes a collection of connected program processors and is used as technological module in the programming environment. Also, the implementation is possible of the so-called concretization systems being an integrated device for constructing program concretizators.

With respect to main criteria of program quality, among program concretizators the following types of tools can be distinguished.

Source-to-source optimizers. They aim at improving basic programs in conventional for the optimizing compiler way, but they transform programs on source language level and take into account the parameters of both compilation and execution environment.

Concretizators making annotated programs more clear and self descriptive. They annotate program by assertions on its semantic properties (such as invariants for term equality, control flow graph and so on), improve the program structure by renaming objects, inserting descriptions, etc.

Instrumentation tools. They make debugging version of a source program by adding basic language statements which test program properties described in the annotations.

Verification tools aimed at a static check of a source annotated program for correctness and supplementing it with annotations which present discrepancies discovered in the program. For example, the tools can elicit the so-called implausibility properties (redundant actions, non-initialized variables, infinite execution, useless objects, over-complicated data organization and etc.) due to certain discrepancies between the program text and the executions which it represents; a test for implausibility permits static detection of some dynamic errors and formal detection of some informal errors [15].

Reducers. They eliminate redundant objects and constructions from a source annotated programs. Reducers are aimed at improving a program given according to all main qualitative criteria by way of the maximal use of the information contained in its annotations.

It should be noted that some conventional tools in which program processing does not always terminate or goes beyond the limits of a basic language can be replaced by concretizators as well. For instance, instead of an interpreter, the programmer's environment may utilize a concretizator which performs a basic program from transformations of the program annotated and constructs the evaluation trace in the annotations having user-defined form. Other concretizators of annotated programs can be used as tools for partial evaluation and specialization of basic programs.

Concretization systems [3] are based on the transformation machine concept and support operational environments ensuring safe and rapid programming of a variety of program processors, as well as their application in combinations usually impossible (e.g., to optimize the debugging version of a source program).

Reliability of tools implemented by means of the concretization systems is provided by applying only such transformations that preserve the meaning of the program processed. The language level for writing transformation tools is getting higher, which contributes to a greater automation of program development. It should be noted that tools can be extended and implement self-descriptive processes of program

transformation (the history of development is presented by a sequence of applied transformations).

In the environment supported by a concretization system it seems practical to create experimental tools for program transformation as well as tools for "single" and "individual" applications, i.e. tools constructed to transform a specific program or designed for one programmer.

If basic and implementation languages of concretization system are the same, mutual applications of program processors will be possible which would provide us with the opportunity to make a compiler from an interpreter, a compiler generator from a partial evaluator and other applications usually considered as motivations for partial evaluation [7,8].

Conclusion

Usually process of program development by successive application of transformations starts with specification (that is a formal statement of a problem or its solution) and ends with an execution program. In the paper, an attempt is made to formulate tools and techniques of annotated programming, whereby a general-purpose program can be annotated by known information about a specific context of its applications and correctly transformed into a specialized program which is equivalent to the original on the context-defined ranges of inputs and outputs and is better than it by quality measure given by the context.

Tools and techniques of annotated program transformations can be used for partial evaluation, compiling, program generation (including compiler generation), and metaprogramming without order-of-magnitude loss of efficiency.

References

1. *Ershov A.P.* ALPHA - an automatic programming system of high efficiency, J.ACM, vol.13, N 1, 1966, p.17-24.

2. *Kennedy K.N.* A survey of compiler optimization.- In: Program Flow Analysis: Theory and Applications, Englewood Cliffs, Prentice-Hall, 1981, p.5-54.

3. *Kasyanov V.N.* Optimizing transformations of programs, Nauka, Moscow, 1988.- 336 p. (in Russian).

4. *Ershov A.P.* The transformational approach in software engineering, In: Software Engineering, Abstracts of the reports to the All-Union Conference, Plenary sessions and general material, Institute of Cybernetics, Ukrainian Academy of Science, Kiev, 1979, p.12-26.

5. *Partsh H., Steinbruggen R.* Program transformation systems, ACM Comput. Surveys, vol.15, N 3, 1983, p.199-236.

6. *Feather M.S.* A survey and classification of some program transformation approaches and techniques, In: Program Specification and Transformation, North-Holland, Amsterdam, 1987, p.165-195.

7. *Ershov A.P.* On the partial computation principle, Information Processing Letters, vol.6, N 2, 1977, p.38-41.

8. *New Generation Computing*, Special Issue: Selected Papers from the Workshop on Partial Evaluation and Mixed Computation, vol.6, Nos. 2,3, 1988.

9. *Kasyanov V.N.* Practical approach to program optimization, Preprint N 135, Computing Center, Siberian Branch of the USSR Academy of Sciences, Novosibirsk, 1978.- 43 p. (in Russian).

10. *Kasyanov V.N.* Annotated program transformations, In: Lecture Notes in Computer Science, vol.405, 1989, p.171-180.

11. *Kasyanov V.N.* Basis for program optimization, In: Proc. IFIP Congress 83, North-Holland, Amsterdam, 1983, p.315-320.

12. *Ershov A.P.* Theory of program schemata, In: Proc. IFIP Congress 71, North-Holland, Amsterdam, 1971, p.28-45.

13. *Ershov A.P.* The transformational machine: theme and variations, In: Lecture Notes in Computer Science, vol.118, 1981, p.16 -32.

14. *Kasyanov V.N., Sabelfeld V.K.* Tools for program transformations, In: Informatika-88: Actes du seminaire Franco-Sovetique, INRIA, Roquencourt, 1988, p.89-100.

15. *Kasyanov V.N., Pottosin I.V.* Application of optimization techniques to correctness problems, In: Constructing Quality Software, Proc. IFIP TC 2 Working Conf., North-Holland, Amsterdam, 1979, p.237-248.