Theoretical Computer Science 90 (1991) 37-46
Elsevier

Transformational approach to program concretization

V. Kasyanov
Institute of Informatics Systems, Siberian Division of the USSR Academy of Sciences Novosibirsk 630090, USSR

Abstract

Kasyanov, V., Transformational approach to program concretization, Theoretical Computer Science 90 (1991) 37-46.

This paper focuses on the problem of program concretization by applying correctness-preserving transformations of annotated programs. According to the approach presented, 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 one on the context-defined ranges of inputs and outputs and is better than it by quality criteria given by the context. Tools for program concretizations via annotated program transformations are considered.

1. Introduction

Transformation techniques are gaining in importance for both theoretical and technological programming. Systems of equivalent (or correctness-preserving) 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 the program transformation paradigm is to essentially improve the construction, reliability, maintenance and extendability 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 this paper, we outline the transformational approach to program concretization, whereby a given general-purpose program can be correctly transformed into a 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 the 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.

The transformational approach described below was considerably influenced by the works of Professor A.P. Ershov.

2. Concretization problem

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

Unlike the equivalent transformations that preserve the functions calculated by the programs being transformed, *generalizing* transformation can convert a source program to a program that solves a more general problem than the source program (for example, a function calculated by the result program can be obtained from the 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]. 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].

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 the 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 (for example, 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 most needs of concretizations are satisfied by using such universal tools of program text construction as macrogenerators and editors. But the approach of automatization of concretizations is not convenient for programmers because it make high demands. Using this approach an end user must programme all specialization processes of its own program.

3. 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 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 the following special-purpose procedure of higher quality

```
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 program 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 also be represented in the form of directives. Unlike annotation-assertion which are predicate constraints on admissible memory states, *annotation-directive* is either a statement that will change the current memory state every time the annotation is reached during possible execution of the program annotated [10] or the name of some 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 a 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:=X; 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=8*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; $}
  E6:=S
END;
```

In [10] a model of programs annotated with assertions and directives which is based on large-scale program schemata covering a broad class of programs and their transformations [3, 11] is described, and the equivalence and generalization relations between annotated and basic programs are defined.

For example, it is assumed that 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
```

and 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.

4. Transformation machine for program concretization

The class of correct transformations of annotated programs covers various kinds of work with basic programs [3, 10]. 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 of performing 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]. Transformation rules used as TM instructions are of the three types:

(1) instructions for moving active points about the programs processed; they make one or a 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 and which alone are able to modify the program processed.

Every ordered pair of annotated program fragments is called a rule. A rule is correct if both its fragments have the same meaning in all admissible program executions. An *elementary* (or *basic*) *transformation* is a set of correct rules. Every elementary transformation is either applicable to a given point of the program, or unapplicable. In the first case, the transformation can be applied to this point. Any transformation application replaces an occurrence of the left side of a certain rule by the right side of this rule.

An elementary transformation system is defined as the instruction set of TM. Thus unlike the transformation machine described in [13], TM employs no instructions whose application correctness depends not only on the fragment transformed, but on the program as a whole. So, every program in the TM instruction language defines a correct transformation of any annotated program, i.e. it is a *program processor* in a sense of the definition of [13]. Program processor describes a concretization transformation if it is total and defines a partial order on the set of the annotated programs.

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 $\Rightarrow$ *statement*) and elementary transformations that originate from object domain laws (e.g., $1+2 \Rightarrow 3$; $exp \times 1 \Rightarrow exp$; $exp/0 \Rightarrow$ *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 construction, 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. Any property transformation can modify only annotations of the program processed.

The transformation implemented by a TM program can be applied either as a fully automatic process or a 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 a procedure with transformations as formal parameters. For example, there are built-in procedures to realize algorithms of data flow analysis, to convert various constructions of an annotated program into canonical forms, for logical inference and so on.

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

5. 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 a 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 [3].

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

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

(2) *Concretizators making annotated programs clearer and more self-descriptive.* They annotate the 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.

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

(4) *Verification tools* aimed at a statical check of a source annotated program for correctness and supplementing it with annotations which present discrepancies discovered in the program. For example, the verification tools can elicit the so-called implausibility properties (redundant actions, noninitialized variables, infinite execution, useless objects, over-complicated data organization, etc.) due to certain discrepancies between the source 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].

(5) *Reducers.* They eliminate redundant objects and constructions from source annotated programs. Reducers are aimed at improving a given program 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 also be replaced by concretizators. 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 for annotated programs can be used as tools for partial evaluation and specialization of basic programs.

Concretization systems 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 that are usually impossible (for example, to optimize the debugging version of a source program).

Reliability of tools implemented by means of the concretization system is provided by applying only such transformation rules 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 the basic language of a concretization system and its implementation language 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].

6. Conclusion

Usually the 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 a program to be executed. In this paper, an attempt is made to suggest tools and techniques for 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 one on the context-defined ranges of inputs and outputs and is better than it by quality criteria given by the context.

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

References

[1] A.P. Ershov, ALPHA - an automatic programming system of high efficiency, *J. ACM* **13**(1) (1966) 17-24.

[2] K.N. Kennedy, A survey of compiler optimization, in: *Program Flow Analysis: Theory and Applications* (Prentice-Hall, Englewood Cliffs, NJ, 1981) 5-54.

[3] V.N. Kasyanov, *Optimizing Transformations of Programs* (Nauka, Moscow, 1988) (in Russian).

[4] A.P. Ershov, 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) 12-26 (in Russian).

[5] H. Partsh and R. Steinbruggen, Program transformation systems, *ACM Comput. Surveys* **15**(3) (1983) 199-236.

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

[7] A.P. Ershov, On the partial computation principle, *Inform. Process. Lett.* **6**(2) (1977) 38-41.

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

[9] V.N. Kasyanov, Program concretization problems, in: *Problems of Theoretical and Systems Programming* (Novosibirsk State University, Novosibirsk, 1982) 35-45 (in Russian).

[10] V.N. Kasyanov, *Annotated Program Transformations*, Lecture Notes in Computer Science **405** (Springer, Berlin, 1989) 171-180.

[11] V.N. Kasyanov, Basis for program optimization, in: *Proc. IFIP Congress 83* (North-Holland, Amsterdam, 1983) 315-320.

[12] A.P. Ershov, Theory of program schemata, in: *Proc. IFIP Congress 71* (North-Holland, Amsterdam, 1971) 28-45.

[13] A.P. Ershov, *The Transformational Machine: Theme and Variations*, Lecture Notes in Computer Science **118** (Springer, Berlin, 1981) 16-32.

[14] V.N. Kasyanov and V.K. Sabelfeld, Tools for program transformations, in: *Informatika-88: Actes du Seminaire Franco-Soviétique*, (INRIA, Roquencourt, 1988) 89-100.

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