TRANSFORMATION APPROACH TO PROGRAM CONCRETIZATION

V. N. Kas'yanov

Program concretization is treated as a transformation of a given program solving a general problem into a variety of specialized versions solving certain subproblems.

All programming techniques and programming automation systems rely on von Neumann’s principle of a modifiable stored program. It is impossible to visualize a modern computing system without software–hardware support of various processes of program generation, modification, and execution. In these processes, the program as a rule is not a syntactic, uninterpreted object. Most program manipulation processes require conservation of some invariant, which is linked in a certain way with the semantics of the program being transformed. Therefore, a comprehensive analysis of such semantic transformations is essential for the development of reliable and efficient automated programming systems.

The most important class of semantic transformations are the so-called functionally equivalent transformations, which preserve the function realized by the transformed program. The study of these transformations is the subject of the theory of program schemas.

Traditionally, the problem of semantic transformations of programs is associated with the construction of compilers and the development of optimizing transformation methods and algorithms. Given a program in a high-level language, the compiler is expected to automatically generate working programs whose quality is comparable to that of manually coded programs [1-2].

This view of semantic transformations is undergoing radical changes in connection with automated construction of programs in high-level languages and the advent of program synthesis systems. Semantic transformations become an important component of controlled (both manual and automated) program manipulation, in which program development is viewed as a process of iterative application of semantic transformations to the initial specification, generally represented in the form of recursive definitions [3-5].

Substantive supplementary information about the program is beginning to be used on an ever increasing scale. This is gradually leading to the crystallization of an approach which may be termed concretizing programming [1-13] and which, according to Ershov [14], constitutes one of the three major forms of programming (alongside assembly and synthesizing programming).

This paper describes the transformation approach to program concretization, based on the analysis of semantic transformations of programs in the form of processes that simultaneously process the program with its annotations [6]. The annotations are associated with program semantics and are written in the form of comments in the program.

1. CONCRETIZING PROGRAM TRANSFORMATIONS

Numerous and fairly representative experiments with derivation of efficient algorithms from inefficient specifications point to the need for using semantic transformations, which either expand or restrict the function described by the program being developed.

Expanding transformations introduce definitions of new functions in terms of existing functions by adding new parameters or results. Restricting transformations, in a certain sense, are the inverse of expanding transformations. Although these non-equivalent transformations are usually applied less frequently than equivalent transformations during program development, their role is essential and often decisive. This important role of expanding and restricting transformations is not accidental and it is not limited to program development. Studies of mixed computations, which constitute a certain formalization of restricting transformations, show that many forms of using programs in practice are in fact mixed-computation processes [3, 9, 10, 13].

Similarly to the identification of optimizing transformations in the class of equivalent transformations, we can identify so-called concretizing transformations in the class of generalizing and restricting transformations. These concretizing transformations are intended to optimize the given program allowing for its application context [1, 6-8]. The problem of program concretization is formulated as a problem of correct transformation of a given program solving a general problem into various specialized versions better suited to solving certain subproblems. The goal of each such transformation is to improve the program by a certain performance criterion (e.g., time, space, reliability, documentability), while preserving its meaning on given subsets of input and output data.

Let us demonstrate the solution of the concretization problem for a simple Pascal procedure $E1$, which produces in $X$ the solution of a linear system of equations with coefficient matrix $A$ with zeros under the main diagonal:

```pascal
PROCEDURE E1 (A:MAT; B:VEC; VAR X:VEC);
VAR I,K:INT; Z:REAL;
BEGIN X[I]:=B[I]/A[I,I]; FOR I:=2 TO N DO
BEGIN Z:=0;
FOR J:=1 TO I-1 DO Z:=Z+A[I,J]*X[J];
X[I]:=(B[I]-Z)/A[I,I] END END;
```

If $E1$ is applied only to diagonal matrices $A$ and the performance criterion is program length, then the program $E1$ should be replaced with the following specialized version:

```pascal
PROCEDURE E2 (A:MAT; B:VEC; VAR X:VEC);
VAR I:INT;
BEGIN FOR I:=1 TO N DO X[I]:=B[I]/A[I,I] END;
```

In a different context, when $E1$ is only applied to find the first root of the system of equations, any reasonable performance criterion suggests replacing the general procedure $E1$ with the following specialized version:

```pascal
PROCEDURE E3 (A:MAT; B:VEC; VAR X:VEC);
BEGIN X[I]:=B[I]/A[I,I] END;
```

Note that so far the need for concretization is mainly satisfied by means of universal program construction tools, such as macro generators and editors. However, this approach to automated concretization is not always convenient, because it leads to unreliable specialized programs and places excessive demands on the user. It requires the user to supply specialized programs, usually in highly detailed terms that do not fit the specific application.

2. PROGRAM ANNOTATION

While concretization requires allowing for the application context of the program, modern high-level programming languages do not have sufficiently rich tools for context definition. It is therefore natural to pass from transformation of programs to transformation of programs with annotations.

Annotations are formalized comments relating to certain points in the base program. They establish the properties of the program and of the computations at these points. Annotations are assumed to be linked with program semantics, in particular, they make it possible to identify the class of inadmissible executions in the given application context and also the results of admissible executions that are not used in the given context.

Let us define the notion of annotated program more precisely. To this end, consider a fairly general notion of a program, based on a large-block schema. This concept fits broad classes of programs and program transformations [1].

Given are two nonempty sets of elements, $V = \{v\}$ and $C \neq \{c\}$, called variables and values. The set $S = \{s\}$ is the set of memory states — arbitrary functions $s: V \rightarrow C$. For any $s_1, s_2 \in S$, we denote by $Y(s_1)$ and $N(s_1)$ the set of variables $v$ for which the value $s_1(v)$ is respectively defined and undefined. We say that $s_1$ and $s_2$ are equal on some set $W \subseteq V$ if for any $v \in W$ we have either $v \in N(s_1) \cap N(s_2)$ or $s_1(v) = s_2(v)$.

A program is the 6-tuple $x = (g, f, p, r, a, d)$ consisting of the control graph $g = (X, U)$ — a directed graph with designated initial $x_0 \in X$ and final $y_0 \in X$ nodes, and five functions: memory transformation function $f: X \rightarrow (S \rightarrow S)$, succession function $p: (X \times \{y_0\}) \rightarrow (S \times U)$, result and argument functions $r, a: X \rightarrow (S \times 2^V)$, and applicability function $d: X \rightarrow (S \rightarrow \{true, false\})$. For any variable $v$, operator $x$, and memory states $s, s_1,$ and $s_2$, these functions have the following properties: 1) $s$ and $f(x)(s)$ are equal on the set $V - r(x)(s)$; 2) if $s_1$ and $s_2$ are equal on $a(x)(s_1)$, then $a(x)(s_1) = a(x)(s_2)$, $r(x)(s_1) = r(x)(s_2)$, $d(x)(s_1) = d(x)(s_2)$.