MODELS OF SEQUENTIAL PROGRAMS USED TO STUDY
FUNCTIONAL EQUIVALENCE OF PROGRAMS

R. I. Podlovchenko

UDC 51:681.3.06

Introduction

In the present article we consider problems which arise when we study the functional equivalence of
sequential programs, which constitute the earliest object of study in programming (subsequently they are called
programs). Functional equivalence of programs, definable by computability of the same function by equivalent
programs, is one of their fundamental properties.

In the given work the primary attention is devoted to problems of the methodology of developments con-
ected with the study of program equivalence.

We know that the investigation of whatever property of programs begins with formalization, as a concept,
both of the program itself and this property -- with the creation of the appropriate formalism. And although in
practice it has been established that any such formalism leads to so complex a concept of the program that the
study of the chosen property requires modeling both the program and the property itself, nevertheless the task
of constructing the formalism of the program is not crossed off -- it is necessary as the object of modeling.

In connection with this, in the first section we consider the formalism of a program, present the general
principles of modeling programs and a given relation between them; we consider realization of these principles
while studying the functional equivalence of programs.

Here we describe two models of a program: a scheme with memory and a scheme without memory. Each
of them is considered together with the equivalence relation which models the functional equivalence of the
programs. In the first case this relation is also called their functional equivalence.

In the study of program models, priority is given to the scheme with memory, in view of which a central
place in the investigations is allotted to functional equivalence of schemes with memory. The "plot of action"
here is provided by the fact that the problem of functional equivalence turned out to be unsolvable in the set of
all schemes with memory. In the subsequent investigations of this problem we distinguish between two direc-
tions:

- isolation of classes of schemes with a solvable problem of functional equivalence;
- search of solvable forms of equivalence which are stronger than functional equivalence.

Both directions use a certain classification of the schemes.

Here we use an adaptable method of scheme classification: uniting into a single class all schemes con-
structed from operators and predicates of a certain set of them that has been specified earlier. This set is
called the base, while the class of schemes is called the class over this base.

In the subsequent three sections we demonstrate methods that are applicable in the investigation of the
directions indicated above.

It is obvious that the search for stronger forms than functional equivalence is preceded by the introduction
of a certain set of equivalences. In Sec. 2 the mechanism of formation of equivalences of schemes with memory,
using the concept of history of travel along the scheme, is revealed. In the set of equivalences constructed we
isolate privileged equivalences; after this, by a stipulation consisting of the circumstance that from correct
equivalence there follows some of the privileged ones, we introduce the concept of correctness of equivalence.
We consider the problem of solvability of a property of correct equivalences. In the investigation of the prob-
lem of functional equivalence in the chosen class of schemes over the base there arises in the first instance
the following problem: Free from the memory, modeling schemes with memory by schemes without memory,
and model the functional equivalence of schemes with memory by an appropriate relation between schemes without memory. Section 3 is devoted to the formulation and solution of this problem.

Section 4 enters into the set of problems arising while working with schemes without memory; methods of solution of these problems are discussed.

1. Modeling Programs and Problems Arising in the Investigation of Functional Equivalence of Programs

The formalism of a program embraces the definition of the program as a certain constructive object, description of the procedure of implementation of the program on a state of the memory regarded as the initial state, determination of one or several functions which are confronted with the program by means of its execution procedure, and, finally, introduction of equivalence relations between programs determined by the stipulation that the programs of the functions compared coincide with one another.

The models of a program to be considered are based on the following formalism of the program.

The program is represented in the form of two components: a finite oriented graph and a mapping matching certain instructions with the nodes of this graph. The graph specifies the succession relation between the instructions of the program (we call it the control relation). For these purposes on the graph an input is singled out (a node at which the instruction always carried out first is located) and an output (a node at which the concluding instruction is located); for each node of the graph, except its output, by means of one or two arcs issuing from the node its successors are indicated. In the first case the node is called a converter, while in the second case it is called a recognizer; two arcs leading from a recognizer to its successors are marked by the numbers 0 and 1.

The instructions being used in the construction of the program have a distinctive feature: The arguments of the functions to be computed in them are stored in the memory cells. The functions themselves are classified according to the following criterion: is it required or not required to store in the memory the value of this function just computed? The functions whose values are not stored in the memory are called logical functions.

An instruction using nonlogical functions in the simplest case* has the form

\[ r_0 := F(r_1, \ldots, r_n) \]  

or

\[ r_0 := r_i. \]  

In (1) by the symbol \( F \) we have denoted a certain concrete \( n \)-placed operation (for example, the operation of addition of two numbers); its arguments are assumed to be stored in cells \( r_1, \ldots, r_n \), the result of the operations being entered into the cell \( r_0 \) for storage. In (2) we use a single-placed identity operation, the symbol of which is omitted; its argument is stored in the cell \( r_i \), while the result is placed in the cell \( r_0 \). The instruction (1) is called the conferring instruction, while (2) is called the sending instruction. These instructions are placed at converter nodes.

An instruction using a logical function (it is also said to be logical) has the form

\[ P(r_1, \ldots, r_n). \]  

where \( P \) is a concrete \( n \)-placed relation (for example, the equality relation of numbers), while \( r_1, \ldots, r_n \) are the cells in which the arguments of this relation are stored. The logical instruction is placed at the recognizer node.

In Fig. 1 we show an example of a program with four conferring instructions and one logical instruction (in the first two instructions a zero-placed operation is used: make equal to 1). The input of the program is marked by an arc arriving at it "from nowhere"; the output is not provided with an instruction, since the latter in standard manner is assumed to be equal to \( r := r \) (an instruction not altering the contents of the memory cells) and is omitted.

Thus, for the specification of a program we must have a certain set of instructions. It is taken as finite and, consequently, uses a finite set of cells occupied by the instructions; we call it the set of cells with memory. The number stored in a cell is called its state, while the situation in which each cell of the memory is in a certain state is called the state of the memory.

*For the sake of simplicity of presentation we confine ourselves to this case.