MATHEMATICS 2024. Vol. 34. Issue 4. Pp. 469–485.

MSC2020: 03F65

© *M. Joudakizadeh, A. P. Beltiukov*

TWO-LEVEL REALIZATION OF LOGICAL FORMULAS FOR DEDUCTIVE PROGRAM SYNTHESIS

This paper presents a novel approach to interpreting logical formulas for synthesizing algorithms and programs. The proposed method combines features of Kleene realizability and Gödel's "dialectica" interpretation but does not rely on them directly. A simple version of positive predicate logic without functions is considered, including conjunction, disjunction, implication, and universal and existential quantifiers. A new realizability semantics for formulas and sequents is described, which considers not just a realization of a formula, but a realization with additional support. The realization roughly corresponds to Kleene realizability. The support provides additional data in favor of the correctness of the realization. The support must confirm that the realization works correctly for the formula under any valid conditions of application. A proof language is presented for which a correctness theorem is proved showing that any derivable sequent has a realization and support confirming that this realization works correctly for this formula under any valid conditions with a suitable interpreter for the programs used.

Keywords: logical formulas, algorithm synthesis, program synthesis, predicate logic, calculus of sequents, proofs, interpretation of logical formulas, artificial intelligence.

DOI: 10.35634/vm240401

In computer science, the synthesis of algorithms and programs has long been a key research area aimed at automating the creation of correct and efficient software [1]. However, as software systems become increasingly complex, traditional approaches to synthesis often prove insufficient to reflect the subtle relationships between program behavior, correctness, and interaction with the environment [2]. This paper presents a new approach to interpreting logical formulas, combining concepts of Kleene realizability and Gödel's "dialectica" interpretation. Our method offers a deeper understanding of problems in automatic and automated synthesis of algorithms and programs, providing a reliable basis for reasoning about complex computational processes. Kleene realizability, introduced in the mid-20th century, provides a computational interpretation of intuitionistic logic, linking proofs with algorithms [3]. Gödel's dialectical interpretation, in turn, offers a way to understand the interaction between different logical systems [4]. Although both approaches have significantly influenced the development of computation theory, they have limitations when applied to modern program synthesis tasks. The present work overcomes these limitations by introducing a new semantic structure that combines elements of both realizability and the "dialectica" interpretation. This structure allows for a more explicit interpretation of program behavior, considering not only the algorithms themselves but also their support modules and their interaction with the environment. The foundation of this work is a new semantics for formulas and sequents, defined through an arbitration procedure that reflects the interaction between programs, their support modules, and the environments in which these programs are executed. The paper also presents a sequent calculus for which correctness is proved with respect to a certain interpretation. It considers the interpretation of proofs as programs that generate algorithms and their support modules, which is useful for automating reasoning in the synthesis of highly reliable programs. The proposed more explicit semantic model for describing program behavior and correctness can be useful in the field of formal methods for program synthesis and verification, automatic error detection, and programming using artificial intelligence [5]. The proposed approach is important for developing reliable and verifiable software. In light of the current state

of software development, this approach becomes particularly relevant. The growing complexity of software systems, increasing volumes of processed data, and rising requirements for security and reliability create a need for new, more advanced methods of program synthesis and verification. Our proposed method allows for more accurate modeling of the interaction of programs with their environment, which is critical for developing reliable control systems, the Internet of Things, and other modern applications. Moreover, this approach can provide a more flexible tool for dealing with uncertainty and incompleteness of data, which is especially important in the context of machine learning and artificial intelligence. It is important to note that the proposed semantic structure is not limited to theoretical reasoning. It can be viewed as a foundation for creating a new generation of software development tools. These tools may include advanced development environments that can provide developers with a deeper understanding of their programs' behavior, automatic code verifiers that can detect complex and non-obvious errors, as well as automatic program synthesis systems that can create correct and efficient algorithms based on high-level specifications. The proposed approach can also be useful for education in computer science. Including the proposed concepts in curricula can contribute to the emergence of a new generation of programmers with a deeper understanding of programming logic and the ability to create higher quality software. In the following sections, we will examine in detail the theoretical foundations of our approach, present formal definitions and proofs.

§ 1. Background

The interpretation of logical formulas plays a crucial role in the automatic and automated synthesis of algorithms and programs. Two classical approaches that have had a significant influency on this field, Kleene's realizability and Gödel's "dialectica" interpretation, laid the foundation for modern methods of program synthesis.

1.1. Kleene realizability

Realizability, proposed by S. Kleene in 1945 [3], is a method of interpreting intuitionistic mathematics. The main idea is to link the truth of a mathematical statement with the existence of a constructive proof or algorithm [6]. Within this approach, a formula is said to be realizable if there exists an effective procedure (or realization) that confirms the truth of this formula.

Specific realizability conditions are defined for various logical constructions. An atomic proposition is realizable if it is true. A conjunction is realizable if both of its terms are realizable. A disjunction is realizable if at least one of the statements is realizable, and we can effectively determine which one. An implication is realizable if there exists an algorithm that can construct a realization of the conclusion for any realization of the premise. A universal quantifier is realizable if there exists an algorithm that constructs a realization for the corresponding statement for any value of the bound variable. An existential quantifier is realizable if we can effectively find a value and a realization for the corresponding statement.

This approach provides a constructive interpretation of logic and mathematics, which is especially important for computer science and computability theory. Kleene realizability allows us to view logical formulas as specifications for algorithms, directly linking logic with computation.

N. A. Shanin made a significant contribution to the development of a constructive understanding of mathematical judgments in his works [7]. He developed a constructive interpretation of classical logic, which in many ways anticipated modern approaches to program synthesis based on logical specifications. His research on the constructive understanding of mathematical judgments influenced the formulation of verification procedures in modern automatic program synthesis systems.

1.2. Gödel's "dialectica" interpretation

The "dialectica" interpretation, developed by Kurt Gödel in 1958 [4], offers an alternative approach to understanding the constructive truth of logical formulas. In this interpretation, truth is viewed as the result of a game between two participants: a "proposer" (or "defender") and an "opponent" [8].

According to this interpretation, each formula is viewed as a game between the proposer, who tries to prove the truth of the formula, and the opponent, who seeks to disprove it. Specific game rules are defined for various logical connectives and quantifiers. For example, when proving a conjunction, the proposer must be ready to prove both statements, whereas for a disjunction, it is sufficient to prove one of them at the proposer's choice. When proving an implication, the roles are reversed: the opponent must prove the premise, and the proposer must prove the conclusion.

A formula is considered true if the proposer has a winning strategy, i. e., a way to prove the formula regardless of the opponent's actions. This approach provides a more dynamic and interactive understanding of logic, which is especially useful in the context of interactive proof systems and automatic program synthesis.

1.3. Current state in the field of algorithm and program synthesis

Currently, the synthesis of algorithms and programs is an actively developing research area that combines methods of logical inference, machine learning, and heuristic search. Modern approaches often rely on a combination of various techniques, including inductive programming and example-based synthesis [9].

Specification-based program synthesis [9] aims to develop programs that are guaranteed to meet formal specifications. This approach is especially important in critical systems where a high degree of confidence in program correctness is required.

The integration of formal verification methods into the synthesis process [10, 11] ensures the correctness of generated programs. This direction is particularly important for creating reliable software in critical areas.

Despite significant progress, important challenges remain in the field of algorithm and program synthesis. The scalability of synthesis methods when working with large and complex programs remains a serious challenge. The efficiency of generated programs often falls short of programs written manually by experienced developers. The interpretability of automatically synthesized programs is also a problem, as it is not always easy to understand or explain how the generated program works.

Current research is aimed at overcoming these limitations and developing more powerful and practical methods for algorithm and program synthesis. The integration of classical approaches, such as Kleene realizability and Gödel's dialectical interpretation, with modern formal verification methods appears promising in this field.

§ 2. Proposed approach

2.1. Overview of the proposed interpretation method

The proposed approach to interpreting logical formulas is based on the idea of constructive truth. Unlike classical logic, where the truth of a formula is determined through correspondence with a model, this approach links the truth of a formula to its potential realization as an algorithm.

The distinct feature of this method is that it considers not only the realization of the formula itself but also two additional components: support and opposition. These components allow us to more precisely define the conditions under which a given realization actually corresponds to the formula.

Formally, we define the truth of a formula by presenting a triple (b'', b, b'), where:

- b is the realization of the formula,
- b'' is the support for this realization,
- b' is the opposition to this realization.

The truth of the formula is determined through a special arbitration procedure

$$ar(b'', b, B, b')$$

which checks whether b is indeed a correct realization of formula B given the support b'' and the opposition b'.

2.2. Support and opposition

Support and opposition enable a more precise determination of the conditions under which a formula's realization is considered correct.

Support (b'') represents additional information or an algorithm that helps justify the correctness of the formula's realization. It may include:

- additional data necessary to verify the algorithm's operation,
- auxiliary procedures.

The role of support for an implicative or universal formula is to provide all necessary arguments and resources confirming that the realization indeed corresponds to the formula for all permissible input values (or to establish the inadmissibility of these values).

Opposition (b') represents a set of conditions or test data aimed at refuting the correctness of the realization. The role of oppositions is to test the robustness of the realization to various scenarios and to ensure that it truly conforms to the formula in all possible situations.

The interaction between support and opposition is as follows. The arbitration procedure $ar(b'', b, B, b')$ evaluates how well support b'' assists realization b in withstanding opposition b'. If the realization successfully meets all the challenges posed by the oppositions with the help of support, it is considered correct for the given formula.

This approach allows for a clearer analysis of algorithm properties and their alignment with logical specifications. It also opens up new possibilities for automatic program synthesis: not only to generate algorithms for problem-solving but also to automatically create modules to detect errors in the application of these algorithms.

§ 3. Formalization

The following provides a detailed description of the approach outlined. Essentially, it is based on an interpretation of logical formulas that combines features of the Kleene realization and Gödel's interpretation of "dialectic", but does not rely directly on them.

3.1. Data structures

As a (simplified) mathematical model of the system of data structures processed by the considered and synthesized algorithms, we consider a set of words in some finite alphabet and the structures obtained from them by applying the operation of constructing finite tuples (possibly multiple times): (x, y), (x, y, z), $((x, y), (x, z))$, etc.

3.2. Logical language syntax

For problem formulations, we will use a simple variant of positive (without negations) predicate logic without functions, with conjunction, disjunction, implication, and quantifiers for universal and existential statements. We use a language without negations because it is best suited for applications of deductive synthesis of reliable programs, where no possibilities for input values can be discarded a priori through logical negation; instead, all possible errors must be anticipated and handled. The language without functions is chosen for simplicity, to avoid additional complexity in the presentation.

The syntax of the logical language is described in Backus normal form, with terminal symbols in apostrophes, non-terminal symbols in angle brackets, and $\{\dots\}$ denoting iteration:

$$\begin{gathered}< formula >::=< predicate >' ('\{< name >',' \} < name >')' \\ |'('< formula >< connective >< formula >')' \\ | < quantifier >< name >< formula >, \\ < connective >::=' \&'|' \vee' |' \Rightarrow', \\ < quantifier >::=' \forall'|' \exists'.\end{gathered}$$

It is assumed that certain languages for the concepts $< predicate >$ and $< name >$ are fixed, avoiding ambiguities in syntactic parsing. The specific content of $< predicate >$ depends on the domain in which the language is applied. The concept $< name >$ refers to the names of variables and constants.

For logical inference, we use sequents of the form:

$$< sequent >::= \{< name >\}\{< formula >\}' \rightarrow'< formula > .$$

The meaning of the sequent

$$x_1 \dots x_m \, G_1 \dots G_n \rightarrow B$$

is close to the meaning of the formula

$$\forall x_1 \dots \forall x_m (G_1 \& \dots \& G_n \Rightarrow B),$$

but it differs slightly concerning the structures of processed data (see definition below).

In the sequent

$$x_1 \dots x_m \, G_1 \dots G_n \rightarrow B,$$

all names were different, and all free variables in formulas were found in the list $x_1 \dots x_m$.

Note that such encoding can be applied in such a way that formulas are also represented as considered data structures. This will be assumed in what follows.

3.3. Semantics of sequents and formulas

Assume a fixed interpreter u for programs written in the form of our data structures. The notation $u(f, x)$ denotes the result of interpreting a program f by interpreter u on input data x. For simplicity and efficiency, assume that the interpreter u always terminates. Thus, we do not claim universal computational models here but rely on some sub-recursive computability. Some properties of interpreter u will become clear later. For brevity, we will write $u(f, x)$ as $f(x)$.

3.3.1 Semantics of formulas

We believe that formulas and sequents define some constructive tasks. These tasks may have solutions, justifications (supports) for these solutions, and an environment in which these solutions and their justifications operate. The task of justifying a solution is to determine the correctness of the environment in which this solution is applied: if the solution does not produce the correct result, but was launched under unacceptable conditions, this does not indicate that the solution itself is incorrect. In other words, justification allows us to separate errors in the solution itself from errors in the application of this solution.

For this, it is considered that some algorithmic arbitration (judging) procedure ar is specified to determine the truth of formulas and sequents. The ar procedure has 4 arguments: $ar(b'', b, B, b')$, where b'', b, b' are data structures, B is a logical formula expressing some constructive task, b is a candidate for solving this task, a realization of this formula, b'' is support for the solution b (its justification), b' is opposition to this solution (the solution environment: test data, values and conditions under which b and b'' will be executed). From here on, names with an even number of apostrophes "play for the formula", with an odd number names "play against the formula". It is assumed that the ar algorithm should produce a logical value, the meaning of which is that b'' justifies the correctness of the solution b of the problem B under the conditions of b'.

It is believed that the meaning of $ar(p'', p, P(\mathbf{a}), p')$, where P is a predicate and $\mathbf{a}$ are names of objects, will be predetermined by the interpretation of our logical language as applied to each specific domain in which this language will be used.

Let us fix the following rules for determining other values of the function ar.

The rule for conjunction is the following:

$$ar((b'', c''), (b, c), (B\&C), (b', c')) = (ar(b'', b, B, b')\&ar(c'', c, C, c')).$$

Here and in the following rules the first occurrence of the logical connective sign (here $\&$) is a formal symbol, the second one is an operation on logical values.

The rule for disjunction is the following:

$$ar(b'', (i, b), (B_0 \vee B_1), b') = ar(b'', b, B_i, b'),$$

where $i = 0, 1$.

The rule for implication is the following:

$$ar((b'_{b''bc'}, c''_{b''b}), c_b, (B \Rightarrow C), ((b'', b), c')) =$$
$$= (ar(b'', b, B, b'_{b''bc'}((b'', b), c')) \Rightarrow ar(c''_{b''b}(b'', b), c_b(b), C, c')).$$

Here and below, to control the correctness of the application of functions, the expected arguments are written in the lower index of the function sign; it should be noted that in the counteraction $((b'', b), c')$ of the entire formula $(B \Rightarrow C)$ there is a part (b'', b), "playing for B"; it supplies test data for checking the implementation of the entire implication, but the part (b'', b) itself also needs to be checked, which is carried out by the program $b'_{b''bc'}$, having all the information necessary for this $((b'', b), c')$; if this check passes, then, using the implementation b, program c implements formula C, in which it is "assisted" by program $c''_{b''b}$, using additional support information b'' (program $c''_{b''b}$ is prohibited from using information c', since this would be "unfair peeking at tests when solving a programming problem").

The rule for existence is the following:

$$ar(b'', (a, b), \exists x B(x), b') = ar(b'', b, B(a), b').$$

Here and below $B(a)$ is the correct substitution of the name a instead of all free occurrences of the variable x in formula $B(x)$.

The rule for universal quantifier is the following:

$$ar(b''_x, b_x, \forall x B(x), (a, b')) = ar(b''_x(a), b_x(a), B(a), b').$$

It is considered that the value of ar for arguments b'' and b of a form not specified in this list are false; if the arguments b'' and b have the correct form and b' is incorrect, then the value of ar is true.

A formula B is called ar-valid if there are b and b'' such that for any b' $ar(b'', b, B, b')$ holds. Here b is called the realization of the formula B, b'' is the support of this realization, and b' is the counteraction (opposition) to the realization and support of the formula B.

3.4. Semantics of sequents

Let us extend the arbitration procedure and ar-validity from formulas to sequents. The arbitration formula for a sequent, where $\mathbf{x}$ are names, $\mathbf{G}$ are formulas, and B is a formula, looks as follows:

$$ar((\mathbf{g}'_{xg''gb'}, \mathbf{b}''_{xg''g}), b_{xg}, \mathbf{xG} \to B, (\mathbf{a}, \mathbf{g}'', \mathbf{g}, b'))$$

where:

- b_{xg} is a realization of the sequent (i. e., the module that generates the realization of B);
- $(\mathbf{g}'_{xg''gb'}, \mathbf{b}''_{xg''g})$ is a support this realization:
  - $b''_{xg''g}$ is a program for generating a support for the realization of formula B;
  - $\mathbf{g}'_{xg''gb'}$ are programs for generating counteractions to formulas $\mathbf{G}$;
- $(\mathbf{a}, \mathbf{g}'', \mathbf{g}, b')$ is a counteraction to the sequent:
  - $\mathbf{a}$ are values of variables $\mathbf{x}$ (names of objects in the subject area);
  - $\mathbf{g}$ are realizations of formulas $\mathbf{G}$ ($\mathbf{g}$ can be considered a vector function);
  - $\mathbf{g}''$ are supports for this formula realizations $\mathbf{G}$ ($\mathbf{g}''$ can be considered a vector function);
  - b' is a counteraction to the realization of formula B.

Based on this meaning of the introduced designations, we obtain the following definition:

$$\begin{gathered}((\mathbf{g}'_{xg''gb'}, b''_{xg''g}), b_{xg}, \mathbf{xG}(\mathbf{x}) \to B(\mathbf{x}), (\mathbf{a}, \mathbf{g}'', \mathbf{g}, b')) = \\ = ar(g''_1, g_1, G_1(\mathbf{a}), g'_{xg''gb'}(\mathbf{a}, \mathbf{g}'', \mathbf{g}, b')_1) \dots ar(g''_n, g_n, G_n(\mathbf{a}), g'_{xg''gb'}(\mathbf{a}, \mathbf{g}'', \mathbf{g}, b')_n) \Rightarrow \\ \Rightarrow ar(b''_{xg''g}(\mathbf{a}, \mathbf{g}'', \mathbf{g}), b_{xg}(\mathbf{a}, \mathbf{g})), B(\mathbf{a}), b').\end{gathered}$$

Here we also assume that the incorrectness of the first and second arguments leads to the falsity of the function ar, and if they are correct, the incorrectness of the fourth argument leads to its truth.

For brevity, in the following places, where it is obvious, we will omit the parameters $\mathbf{x}$ and $\mathbf{a}$ from the formulas.

3.5. Calculus and proof language

To justify the constructive truth of formulas and sequents, we will use proofs. The correctness of proofs in the constructive sense is usually understood as the possibility of extracting from these proofs realizations and their supports for the formulas being proven. Here we will use an extreme case of such extraction: the proof itself can be interpreted so that it will be executed as an algorithm producing the necessary realization and its support. This will be justified below. We will write proofs in the form of functional terms. In general, the language of functional terms has the following syntax:

$$< term >::=< name > [\prime(\prime\{< term >\prime,\prime\} < term >\prime)\prime]$$

Here [...] is an optional part. Typically $< name >$ are non-empty strings of letters and/or digits, interpreted as names of functions to apply to the arguments that follow them. Note that proofs can also be encoded as our processable data structures.

The statement that the term p is a proof of the sequent S is written in the form $p : S$. In the calculus given below, it is precisely such statement that is proved. The proof of the formula B is a proof of the sequent $\rightarrow B$. The postulates of the calculus described will be written in the form $p : S \Leftarrow A$, where $p : S$ is the conclusion of the postulate, A are the conditions of its application and its premises, each of which also has a similar form $(q : T)$

3.5.1 List of postulates

Here are all the postulates of our calculus.

An axiom scheme is a postulate without premises:

$$pr(i) : \mathbf{xG} \rightarrow G_i,$$

where i is a notation of a natural number, G_i is the ith element of the list G. No other axioms (postulates without premises) are considered.

The rule for generating a conjunction is as follows:

$$cg(f, h) : \mathbf{xG} \rightarrow (B\&C) \Leftarrow f : \mathbf{xG} \rightarrow B, h : \mathbf{xG} \rightarrow C.$$

The two rules for generating disjunction are as follows:

$$dg_i(f) : \mathbf{xG} \rightarrow (B_0 \vee B_1) \Leftarrow f : \mathbf{xG} \rightarrow B_i,$$

where i is 0 or 1.

The rule for generating an implication is as follows:

$$ig(f) : \mathbf{xG} \rightarrow (B \Rightarrow C) \Leftarrow f : \mathbf{xG}B \rightarrow C.$$

The rule for generating universality is as follows:

$$ag(f) : \mathbf{xG} \rightarrow \forall y B(y) \Leftarrow f : \mathbf{x}z\mathbf{G} \rightarrow B(z),$$

where z is a new name that did not appear in $\mathbf{x}$.

The existence generation rule is as follows:

$$eg(i, f) : \mathbf{xG} \rightarrow \exists y B(y) \Leftarrow f : \mathbf{xG} \rightarrow B(x_i).$$

The rule for using conjunction is as follows:

$$cu(i, f) : \mathbf{xG} \rightarrow D \Leftarrow G_i \equiv (B\&C), f : \mathbf{xG}BC \rightarrow D,$$

where $\equiv$ is the graphical (literal) equality sign.

The rule for using disjunction is as follows:

$$du(i, f_0, f_1) : \mathbf{xG} \to D \Leftarrow G_i \equiv (B_0 \vee B_1), f_0 : \mathbf{xG}B \to D, f_1 : \mathbf{xG}C \to D.$$

The rule for using implication is as follows:

$$iu(i, f, h) : \mathbf{xG} \to D \Leftarrow G_i \equiv (B \Rightarrow C), f : \mathbf{xG} \to B, h : \mathbf{xG}C \to D.$$

The rule for using universality is as follows:

$$au(i, j, f) : \mathbf{xG} \to D \Leftarrow G_i \equiv \forall y B(y), f : \mathbf{xG}B(x_i) \to D.$$

The rule for using existence is as follows:

$$eu(i, f) : \mathbf{xG} \to D \Leftarrow G_i \equiv \exists y B(y), f : \mathbf{xG}B(z) \to D,$$

where z is a new name that was not found among $\mathbf{x}$.

3.6. Correctness of the Calculus

Theorem 3.1 (on the correctness of calculus). *It is possible to choose an interpreter u such that any derivable sequent is ar-valid.*

3.7. Proof of the Theorem 3.1

The proof is carried out by induction on the construction of the derivation of sequents.

3.7.1 Axioms are ar-universally valid

Let $pr(i) : \mathbf{xG} \to B$. This means that $B \equiv G_i$. Let us construct $\mathbf{g'}_{xg''gb'}$, $\mathbf{b}_{xg}$ and $\mathbf{b''}_{xg''g}$ such that, for any $\mathbf{a}$, $\mathbf{g''}$, $\mathbf{g}$, b',

$$ar((\mathbf{g'}_{xg''gb'}, b''_{xg''g}), b_{xg}, \mathbf{xG} \to B, (\mathbf{a}, \mathbf{g''}, \mathbf{g}, b')),$$

i. e.,

$$ar(g''_1, g_1, G_1, g'_{xg''gb'}(\mathbf{a}, \mathbf{g''}, \mathbf{g}, b')_1) \& \ldots \& ar(g''_n, g_n, G_n, g'_{xg''gb'}(\mathbf{a}, \mathbf{g''}, \mathbf{g}, b')_n) \Rightarrow$$
$$\Rightarrow ar(b''_{xg''g}(\mathbf{a}, \mathbf{g''}, \mathbf{g}), b_{xg}(\mathbf{a}, \mathbf{g}), B, b').$$

For this, it is sufficient to take:
$b_{xg}(\mathbf{a}, \mathbf{g}) = g_i$,
$b''_{xg''g}(\mathbf{a}, \mathbf{g''}, \mathbf{g}) = g''_i$,
$g'_{xg''gb'}(\mathbf{a}, \mathbf{g''}, \mathbf{g}, b')_i = b'$,
$g'_{xg''gb'}(\mathbf{a}, \mathbf{g''}, \mathbf{g}, b')_j = 0$,
for j different from i (these values can be taken arbitrarily, they do not worsen the result).

This assumes that there are corresponding operations in the programming language for the interpreter. Then we have

$$ar(g''_i, g_i, G_i, g'_{xg''gb'}(\mathbf{a}, \mathbf{g''}, \mathbf{g}, b')_i) \Rightarrow ar(b''_{xg''g}(\mathbf{a}, \mathbf{g''}, \mathbf{g})), b_{xg}(\mathbf{a}, \mathbf{g}), B, b'),$$

since the same thing is written on the left and on the right, and additional premises do not worsen the result. We will further assume that the notation $f : \mathbf{xG} \to B$ can be interpreted in such a way that the term f has a value whose form is determined by the equality:

$$f = ((\mathbf{g'}_{xg''gb'}, b''_{xg''g}), b_{xg}),$$

and the following condition is satisfied:

$$ar((\mathbf{g}'_{xg''gb'}, b''_{xg''g}), b_{xg}, \mathbf{xG} \to B, (\mathbf{a}, \mathbf{g}'', \mathbf{g}, b'))$$

for any $\mathbf{a}$, $\mathbf{g}''$, $\mathbf{g}$, b'. Note that the language of proof terms itself is an algorithmic language and this section essentially describes its operational semantics. The language described is unusual in that its terms define not single algorithms but pairs of them: the program itself and its support. This can be useful for programming very important tasks. Thus, from the above it follows that

$$pr(i) = ((\mathbf{g}'_{xg''gb'}, b''_{xg''g}), b_{xg}),$$

where $\mathbf{g}'_{xg''gb'}$, $b''_{xg''g}$ and b_{xg} are defined above. We interpret the proofs obtained by the remaining postulates in the same way. The proof of the correctness of each rule is a proof of a rather complex implication, which consists of constructing all the missing elements and checking that everything required in the conclusion is in the premise. In this case, the operation of the rules ig and ag leads to programming the implementations and their supports. Though this programming is simple. It is reduced to combining an existing program with some of its arguments so that when interpreting the resulting object, add the missing arguments and finally execute the required program.

3.7.2 The conjunction generation rule from ar-universally valid premises yields an ar-universally valid conclusion

We should define $cg(f, h)$ such that

$$cg(f, h) : \mathbf{xG} \to (B\&C) \Leftarrow f : \mathbf{xG} \to B, h : \mathbf{xG} \to C.$$

Let $f = ((\mathbf{g}'_{xg''gb'}, b''_{xg''g}), b_{xg})$ and

$$ar((\mathbf{g}'_{xg''gb'}, b''_{xg''g}), b_{xg}, \mathbf{xG} \to B, (\mathbf{a}, \mathbf{g}'', \mathbf{g}, b'))$$

for any $\mathbf{a}$, $\mathbf{g}''$, $\mathbf{g}$, b', i. e.,

$$\begin{gathered} ar(g''_1, g_1, G_1, \mathbf{g}'_{xg''gc'}(\mathbf{a}, \mathbf{g}'', \mathbf{g}, c')_1)\& \ldots \& \\ \&ar(g''_n, g_n, G_n, \mathbf{g}'_{xg''gc'}(\mathbf{a}, \mathbf{g}'', \mathbf{g}, c')_n) \Rightarrow \\ \Rightarrow ar(c''_{xg''g}(\mathbf{a}, \mathbf{g}'', \mathbf{g}), c_{xg}(\mathbf{a}, \mathbf{g}), C, c'). \end{gathered}$$

Let's define $cg(f, h) = ((\mathbf{g}'_{xg''gb'c'}, d''_{xg''g}), d_{xg})$ so that

$$ar((\mathbf{g}'_{xg''gb'c'}, d''_{xg''g}), d_{xg}, \mathbf{xG} \to (B\&C), (\mathbf{a}, \mathbf{g}'', \mathbf{g}, (b', c')))$$

for any $\mathbf{a}$, $\mathbf{g}''$, $\mathbf{g}$, b', c', i. e.,

$$\begin{gathered} ar(g''_1, g_1, G_1, g'_{xg''gb'c'})(\mathbf{a}, \mathbf{g}'', \mathbf{g}, (b', c'))_1)\& \ldots \& \\ \&ar(g''_n, g_n, G_n, g'_{xg''gb'c'}(\mathbf{a}, \mathbf{g}'', \mathbf{g}, (b', c'))_n) \Rightarrow \\ \Rightarrow ar(d''_{xg''g}(\mathbf{a}, \mathbf{g}'', \mathbf{g}), d_{xg}(\mathbf{a}, \mathbf{g}), (B\&C), (b', c')). \end{gathered}$$

For this, it is enough to take

$d_{xg}(\mathbf{a}, \mathbf{g}) = (b_{xg}(\mathbf{a}, \mathbf{g}), c_{xg}(\mathbf{a}, \mathbf{g}))$,
$d''_{xg''g}(\mathbf{a}, \mathbf{g}'', \mathbf{g}) = (b''_{xg''g}(\mathbf{a}, \mathbf{g}'', \mathbf{g}), c''_{xg''g}(\mathbf{a}, \mathbf{g}'', \mathbf{g}))$,
$g'_{xg''gb'c'}(\mathbf{a}, \mathbf{g}'', \mathbf{g}, (b', c'))_i =$
if
$\mathbf{not}(ar(g''_i, g_i, G_i, g'_{xg''gb'}(\mathbf{a}, \mathbf{g}'', \mathbf{g}, b')_i)$
then
$g'_{xg''gb}(\mathbf{a}, \mathbf{g}'', \mathbf{g}, b')_i$
else
$g'_{xg''gc'}(\mathbf{a}, \mathbf{g}'', \mathbf{g}, c')_i$.

3.7.3 The rules for generating disjunctions from ar-generally valid premises yield ar-generally valid conclusions

This assertion is proved similarly to the previous assertion.

3.7.4 A rule for generating an implication from an ar-valid premise yields an ar-valid conclusion

Let $ig(f)$ should be defined such that

$$ig(f) : \mathbf{xG} \to (B \Rightarrow C) \Leftarrow f : \mathbf{xG}B \to C.$$

Let $f = ((\mathbf{g}'_{xg''b''gbc'}, b''_{xg''b''gbc'}), c''_{xg''b''gb}, c_{xgb})$ and, for any $\mathbf{a}$, $\mathbf{g}''$, b'', $\mathbf{g}$, b, c', the implication is fulfilled:

$$\begin{gathered} ar(g''_1, g_1, G_1, \mathbf{g}'_{xg''b''gbc'}(\mathbf{a}, (\mathbf{g}'', b''), (\mathbf{g}, b), c')_1)\& \ldots \& \\ \&ar(g''_n, g_n, G_n, \mathbf{g}'_{xg''b''gbc'}(\mathbf{a}, (\mathbf{g}'', b''), (\mathbf{g}, b), c')_n)\& \\ \&ar(b'', b, B, b'_{xg''b''gbc'}(\mathbf{a}, (\mathbf{g}'', b''), (\mathbf{g}, b), c'))) \Rightarrow \\ \Rightarrow ar(c''_{xg''b''gb}(\mathbf{a}, (\mathbf{g}'', b''), (\mathbf{g}, b)), c_{xgb}(\mathbf{a}, (\mathbf{g}, b)), C, c'). \end{gathered}$$

Let's define $ig(f) = ((\mathbf{g}'_{xg''gb''bc'}, d''_{xg''g}), d_{xg})$ so that, for any $\mathbf{a}$, $\mathbf{g}''$, $\mathbf{g}$, b'', b, c', the implication is fulfilled

$$\begin{gathered} ar(g''_1, g_1, G_1, g'_{xg''gb''bc'}(\mathbf{a}, \mathbf{g}'', \mathbf{g}, (b'', b, c'))_1)\& \ldots \& \\ \&ar(g''_n, g_n, G_n, g'_{xg''gb''bc'}(\mathbf{a}, \mathbf{g}'', \mathbf{g}, (b'', b, c'))_n) \Rightarrow \\ \Rightarrow ar(d''_{xg''g}(\mathbf{a}, \mathbf{g}'', \mathbf{g}), d_{xg}(\mathbf{a}, \mathbf{g}), (B \Rightarrow C), (b'', b, c')). \end{gathered}$$

To do this, it is sufficient that the following is done:
$d_{xg}(\mathbf{a}, \mathbf{g}) = c_b$,
$d''_{xg''g}(\mathbf{a}, \mathbf{g}'', \mathbf{g}) = (b'_{b''bc}, c''_{b''b})$,
$g'_{xg''gb''bc}(\mathbf{a}, \mathbf{g}'', \mathbf{g}, (b'', b, c'))_i = g'_{xg''b''gbc'}(\mathbf{a}, (\mathbf{g}'', b''), (\mathbf{g}, b), c')_i$,
where c_b is the realization of the formula $(B \Rightarrow C)$, $(b'_{b''bc}, c''_{b''b})$ is the support for the formula $(B \Rightarrow C)$. To do this, we will take a special code as c_b: $mix(c_{xgb}, \mathbf{a}, \mathbf{g})$, where mix is the mixed (delayed) computation operator symbol, which is executed by the interpreter as follows:

$$mix(c_{xgb}, \mathbf{a}, \mathbf{g})(b) = c_{xgb}(\mathbf{a}, \mathbf{g}, b).$$

As $b'_{b''bc'}$ we will take a special code $mix'(b'_{xg''b''gbc'}, \mathbf{a}, (\mathbf{g}''), (\mathbf{g}))$ with the property:

$$mix'(b'_{xg''b''gbc'}, \mathbf{a}, (\mathbf{g}''), (\mathbf{g}))(b'', b, c') = b'_{xg''b''gbc'}(\mathbf{a}, (\mathbf{g}'', b''), (\mathbf{g}, b), c').$$

As $c''_{b''b}$ we will take a special code $mix''(c''_{xg''b''gb}, \mathbf{a}, (\mathbf{g}''), (\mathbf{g}))$ with the property:

$$mix''(c''_{xg''b''gb}, \mathbf{a}, (\mathbf{g}''), (\mathbf{g}))(b'', b) = c''_{xg''b''gb}(\mathbf{a}, (\mathbf{g}'', b''), (\mathbf{g}, b)).$$

The interpreter is required to be able to execute these codes.

3.7.5 The rule of generating universality from an ar-valid premise yields an ar-valid conclusion

This statement is proved similarly to the previous statement with the introduction of similar program codes.

3.7.6 The rule of generating existence from an ar-valid premise yields an ar-valid conclusion

This statement is substantiated similarly to the previous rules.

3.7.7 The rule of using conjunction from an ar-valid premise yields an ar-valid conclusion

Let $cu(i,f)$ be defined such that

$$cu(i,f):\mathbf{xG}\to D\Leftarrow G_i\equiv(B\&C),f:\mathbf{xG}BC\to D.$$

Let $f=((\mathbf{g}'_{xg''b''c''gbcd'},b'_{xg''b''c''gbcd'},c'_{xg''b''c''gbcd'}),d''_{xg''b''c''gbc},d_{xgbc})$ and, for any $\mathbf{a}$, $\mathbf{g}''$, b'', c'', $\mathbf{g}$, b, c, d', the implication holds:

$$\begin{gathered}ar(g''_1,g_1,G_1,g'_{xg''b''c''gbcd'}(\mathbf{a},(\mathbf{g}'',b'',c''),(\mathbf{g},b,c),d')_1)\&\ldots\&\\ \&ar(g''_n,g_n,G_n,g'_{xg''b''c''gbcd'}(\mathbf{a},(\mathbf{g}'',b'',c''),(\mathbf{g},b,c),d')_n)\&\\ \&ar(b'',b,B,b'_{xg''b''c''gbcd'}(\mathbf{a},(\mathbf{g}'',b'',c''),(\mathbf{g},b,c),d'))\&\\ \&ar(c'',c,C,c'_{xg''b''c''gbcd'}(a,(g'',b'',c''),(g,b,c),d'))\Rightarrow\\ \Rightarrow ar(d''_{xg''b''c''gbc}(\mathbf{a},(\mathbf{g}'',b'',c''),(\mathbf{g},b,c)),d_{xgbc}(\mathbf{a},(\mathbf{g},b,c)),D,d').\end{gathered}$$

Let's define $cu(i,f)=((\mathbf{g}'_{xg''gd'},d''_{xg''g}),d_{xg})$ so that, for any $\mathbf{a}$, $\mathbf{g}''$, $\mathbf{g}$, d', the implication holds:

$$\begin{gathered}ar(g''_1,g_1,G_1,g'_{xg''gd'}(\mathbf{a},\mathbf{g}'',\mathbf{g},d')_1)\&\ldots\&\\ \&ar(g''_n,g_n,G_n,g'_{xg''gd'}(\mathbf{a},\mathbf{g}'',\mathbf{g},d')_n)\Rightarrow\\ \Rightarrow ar(d''_{xg''g}(\mathbf{a},\mathbf{g}'',\mathbf{g}),d_{xg}(\mathbf{a},\mathbf{g}),D,d').\end{gathered}$$

For this purpose, it is sufficient that the following is satisfied:
$d_{xg}(\mathbf{a},\mathbf{g})=d_{xgbc}(\mathbf{a},(\mathbf{g},b,c))$,
$d''_{xg''g}(\mathbf{a},\mathbf{g}'',\mathbf{g})=d''_{xg''b''c''gbc}(\mathbf{a},(\mathbf{g}'',b'',c''),(\mathbf{g},b,c))$,
$g'_{xg''gd'}(\mathbf{a},\mathbf{g}'',\mathbf{g},d')_i=$
if
$\mathbf{not}(ar(g''_i,g_i,G_i,g'_{xg''b''c''gbcd'}(\mathbf{a},(\mathbf{g}'',b'',c''),(\mathbf{g},b,c),d')_i))$
then
$g'_{xg''b''c''gbcd'}(\mathbf{a},(\mathbf{g}'',b'',c''),(\mathbf{g},b,c),d')_i$
else
if
$\mathbf{not}(ar(b'',b,B,b'_{xg''b''c''gbcd'}(\mathbf{a},(\mathbf{g}'',b'',c''),(\mathbf{g},b,c),d'))$
then
$(b''_{xg''b''c''gbcd'}(\mathbf{a},(\mathbf{g}'',b'',c''),(\mathbf{g},b,c),d'),0)$
else
$(0,c''_{xg''b''c''gbcd'}(\mathbf{a},(\mathbf{g}'',b'',c''),(\mathbf{g},b,c),d'))$.
At j other than i, let

$$g'_{xg''gd'}(\mathbf{a},\mathbf{g}'',\mathbf{g},d')_j=g'_{xg''b''c''gbcd'}(\mathbf{a},(\mathbf{g}'',b'',c''),(\mathbf{g},b,c),d')_i.$$

The correctness of all other rules is justified similarly. □

§ 4. Applications

4.1. Potential applications in algorithm and program synthesis

The proposed approach enables the use of the following new capabilities in the field of automatic synthesis of algorithms and programs.

1. Synthesis of more reliable programs: Incorporating the concepts of support and opposition allows generating not only the programs themselves but also modules for verifying the correct application of these programs. This can significantly enhance the reliability of synthesized programs [12].

2. Adaptive synthesis: The proposed approach allows the creation of algorithms that can adapt to various execution conditions, represented through opposition. This is particularly useful in the context of software development for dynamic and uncertain environments [13].

3. Explainable artificial intelligence: Generating support along with the algorithm can be a step towards creating explainable AI models, which is a critical requirement in several key applications [14].

4. Optimization of algorithms: Analyzing the interaction between implementation, support, and opposition can be used for the automatic optimization of algorithms, considering different usage scenarios [15].

4.2. Practical application

The proposed approach to interpreting logical formulas, combining elements of Kleene's realizability and Gödel's interpretation, offers new opportunities for solving practical problems in program synthesis. Let's consider some potential areas of application for this method.

In the development of safety-critical systems requiring high reliability and verifiability, the proposed approach can be used for the automatic synthesis of programs that are correct by design. For example, in developing software for medical devices, this method can be applied to create algorithms for drug dosage control, ensuring that the synthesized program takes into account all possible scenarios and constraints.

In the field of financial technology, the proposed approach can be applied to create trading algorithms and risk analysis tools. Using the constructed interpretation, it is possible to synthesize programs that not only operate efficiently under standard conditions but also correctly handle exceptional market situations.

Consider a simple example of applying our method to solve a specific programming task. Suppose we need to synthesize a program for sorting an array that must be correct and efficient. Using our approach, we can represent the task specification as a logical formula:

$$\forall x\,(\text{array}(x) \Rightarrow \exists y\,(\text{sorted}(y) \wedge \text{transmutation}(x, y)))$$

Here x represents the input array, and y is the sorted output array. The program synthesis system based on the proposed approach can automatically construct a proof of this formula, from which the sorting algorithm will then be extracted. In this case, the implementation will be the sorting algorithm itself, the support will provide for operation when the application condition $\text{array}(x)$ is violated, and the counteractions will supply different values of x and try to find errors in the execution of the operators: "$\text{sorted}(y)$" and "$\text{permutation}(x, y)$".

Thus, our approach not only allows for the synthesis of correct programs but also provides additional guarantees of their correctness, which is especially important in critical applications.

§ 5. Conclusion

The described approach to interpreting logical formulas combines elements of Kleene realizability and Gödel's "dialectica" interpretation, but does not rely on them directly. This methodology is proposed to deepen the understanding of tasks in the field of automatic and automated synthesis of algorithms and programs, ensuring the reliability of their solutions.

The approach is based on a simple mathematical model of data structures, presented in the form of words in a finite alphabet and structures obtained from them by applying the operation of constructing finite tuples. This allows working with a wide range of algorithmic tasks while maintaining mathematical rigor and clarity of formulations.

To formulate problems, a simple version of positive predicate logic without negations is used, including conjunction, disjunction, implication, and universal and existential quantifiers. The choice of such a language is particularly well-suited for the deductive synthesis of reliable programs, as it allows considering all possible input values and providing for the processing of all possible errors.

The semantics of formulas and sequents is defined through the concept of constructive tasks, their solutions, supports for these solutions, and the environment in which these solutions and their supports operate. A special arbitration procedure is introduced to determine the truth of formulas and sequents, which allows formal evaluation of the correctness of proposed solutions.

A calculus is presented that uses functional terms as proof notations. The calculus includes axioms and rules for generating and using logical connectives and quantifiers. The correctness of this calculus is proved by showing that any derivable sequent is valid within the defined arbitration semantics.

One of the innovations of the approach is the presentation of the language of proof terms as an algorithmic language with an unusual feature: its terms specify not single algorithms, but pairs of them that are the program itself and its support. This can be especially useful in programming critical tasks where a high level of reliability and formal verification is required.

The proposed approach provides a rigorous foundation for reasoning about algorithm synthesis and synthesis itself, with an emphasis on reliability and formal correctness. Overall, this methodology may lead to the creation of more reliable and correct software systems in the future.

REFERENCES

1. Gulwani S., Polozov O., Singh R. Program synthesis, *Foundations and Trends® in Programming Languages*, 2017, vol. 4, issues 1–2, pp. 1–119. https://doi.org/10.1561/2500000010
2. Alur R., Bodik R., Juniwal G., Martin M. M. K., Raghothaman M., Seshia S. A., Singh R., Solar-Lezama A., Torlak E., Udupa A. Syntax-guided synthesis, *2013 Formal Methods in Computer-Aided Design*, Portland: IEEE, 2013, pp. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385
3. Kleene S. C. On the interpretation of intuitionistic number theory, *The Journal of Symbolic Logic*, 1945, vol. 10, issue 4, pp. 109–124. https://doi.org/10.2307/2269016
4. Gödel K. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes, *Dialectica*, 1958, vol. 12, issues 3–4, pp. 280–287. https://doi.org/10.1111/j.1746-8361.1958.tb01464.x
5. Seshia S. A., Desai A., Dreossi T., Fremont D. J., Ghosh S., Kim E., Shivakumar S., Vazquez-Chanlatte M., Yue Xiangyu. Formal specification for deep neural networks, *Automated Technology for Verification and Analysis: 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7–10, 2018, Proceedings*, Cham: Springer, 2018, pp. 20–34. https://doi.org/10.1007/978-3-030-01090-4_2
6. Buss S. R. (Ed.). *Handbook of proof theory*, Amsterdam: Elsevier, 1998.
7. Shanin N. A. A constructive interpretation of mathematical judgments, *Trudy Matematicheskogo Instituta imeni V. A. Steklova*, 1958, vol. 52, pp. 226–311. https://www.mathnet.ru/eng/tm1319
8. Kohlenbach U. Gödel's functional interpretation and its use in current mathematics, *Dialectica*, 2008, vol. 62, issue 2, pp. 223–267. https://doi.org/10.1111/j.1746-8361.2008.01141.x
9. Solar-Lezama A., Rabbah R., Bodík R., Ebcioğlu K. Programming by sketching for bit-streaming programs, *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, Association for Computing Machinery, 2005, pp. 281–294. https://doi.org/10.1145/1065010.1065045

10. Hoare C. A. R. An axiomatic basis for computer programming, *Communications of the ACM*, 1969, vol. 12, issue 10, pp. 576–580. https://doi.org/10.1145/363235.363259
11. Dijkstra E. W. Guarded commands, nondeterminacy and formal derivation of programs, *Communications of the ACM*, 1975, vol. 18, issue 8, pp. 453–457. https://doi.org/10.1145/360933.360975
12. Srivastava S., Gulwani S., Foster J. S. From program verification to program synthesis, *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Association for Computing Machinery, 2010, pp. 313–326. https://doi.org/10.1145/1706299.1706337
13. Katz G., Barrett C., Dill D. L., Julian K., Kochenderfer M. J. Reluplex: An efficient SMT solver for verifying deep neural networks, *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24–28, 2017, Proceedings, Part I*, Cham: Springer, 2017, pp. 97–117. https://doi.org/10.1007/978-3-319-63387-9_5
14. Guidotti R., Monreale A., Ruggieri S., Turini F., Giannotti F., Pedreschi D. A survey of methods for explaining black box models, *ACM Computing Surveys (CSUR)*, 2018, vol. 51, issue 5, article number: 93, pp. 1–42. https://doi.org/10.1145/3236009
15. Schkufza E., Sharma R., Aiken A. Stochastic superoptimization, *ACM SIGARCH Computer Architecture News*, 2013, vol. 41, issue 1, pp. 305–316. https://doi.org/10.1145/2490301.2451150

Received 19.08.2024
Accepted 25.11.2024

Milad Joudakizadeh, Post-Graduate Student, Department of Theoretical Foundations of Computer Science, Udmurt State University, ul. Universitetskaya, 1, Izhevsk, 426034, Russia.
ORCID: https://orcid.org/0000-0002-6167-6237
E-mail: dzhudakizade@udsu.ru

Anatoly Petrovich Beltiukov, Doctor of Physics and Mathematics, Professor, Department of Theoretical Foundations of Computer Science, Udmurt State University, ul. Universitetskaya, 1, Izhevsk, 426034, Russia.
ORCID: https://orcid.org/0000-0002-3433-9067
E-mail: belt.udsu@mail.ru

Citation: M. Joudakizadeh, A. P. Beltiukov. Two-level realization of logical formulas for deductive program synthesis, *Vestnik Udmurtskogo Universiteta. Matematika. Mekhanika. Komp'yuternye Nauki*, 2024, vol. 34, issue 4, pp. 469–485.

МАТЕМАТИКА 2024. Т. 34. Вып. 4. С. 469–485.

М. Джудакизаде, А. П. Бельтюков

Двухуровневая реализация логических формул для дедуктивного синтеза программ

Ключевые слова: логические формулы, синтез алгоритмов, синтез программ, логика предикатов, исчисление секвенций, доказательства, интерпретации логических формул, искусственный интеллект.

УДК 510.649

DOI: 10.35634/vm240401

В данной работе представлен новый подход к интерпретации логических формул для синтеза алгоритмов и программ. Предложенный метод сочетает в себе черты реализации Клини и интерпретации Гёделя «диалектика», но не опирается на них непосредственно. Рассматривается простой вариант позитивного языка логики предикатов без функций, с конъюнкцией, дизъюнкцией, импликацией и кванторами всеобщности и существования. Описана новая реализационная семантика формул и секвенций, в которой рассматривается не просто реализация формулы, а реализация с дополнительной поддержкой. Реализация примерно соответствует реализации Клини. Поддержка предоставляет дополнительные данные в пользу того, что реализация корректна. Поддержка должна подтвердить, что реализация работает корректно для формулы в любых корректных условиях применения. Представлен язык доказательств, для которого доказана теорема о корректности, показывающая, что любая выводимая секвенция имеет реализацию и поддержку, подтверждающую, что эта реализация работает правильно для этой формулы в любых корректных условиях при подходящем интерпретаторе используемых программ.

СПИСОК ЛИТЕРАТУРЫ

1. Gulwani S., Polozov O., Singh R. Program synthesis // Foundations and Trends® in Programming Languages. 2017. Vol. 4. Issues 1–2. P. 1–119. https://doi.org/10.1561/2500000010
2. Alur R., Bodik R., Juniwal G., Martin M. M. K., Raghothaman M., Seshia S. A., Singh R., Solar-Lezama A., Torlak E., Udupa A. Syntax-guided synthesis // 2013 Formal Methods in Computer-Aided Design. Portland: IEEE, 2013. P. 1–8. https://doi.org/10.1109/FMCAD.2013.6679385
3. Kleene S. C. On the interpretation of intuitionistic number theory // The Journal of Symbolic Logic. 1945. Vol. 10. Issue 4. P. 109–124. https://doi.org/10.2307/2269016
4. Gödel K. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes // Dialectica. 1958. Vol. 12. Issues 3–4. P. 280–287. https://doi.org/10.1111/j.1746-8361.1958.tb01464.x
5. Seshia S. A., Desai A., Dreossi T., Fremont D. J., Ghosh S., Kim E., Shivakumar S., Vazquez-Chanlatte M., Yue Xiangyu. Formal specification for deep neural networks // Automated Technology for Verification and Analysis: 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7–10, 2018, Proceedings. Cham: Springer, 2018. P. 20–34. https://doi.org/10.1007/978-3-030-01090-4_2
6. Buss S. R. (Ed.). Handbook of proof theory. Amsterdam: Elsevier, 1998.
7. Шанин Н. А. О конструктивном понимании математических суждений // Труды Математического института имени В. А. Стеклова. 1958. Т. 52. С. 226–311. https://www.mathnet.ru/rus/tm1319
8. Kohlenbach U. Gödel's functional interpretation and its use in current mathematics // Dialectica. 2008. Vol. 62. Issue 2. P. 223–267. https://doi.org/10.1111/j.1746-8361.2008.01141.x
9. Solar-Lezama A., Rabbah R., Bodík R., Ebcioğlu K. Programming by sketching for bit-streaming programs // Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. Association for Computing Machinery, 2005. P. 281–294. https://doi.org/10.1145/1065010.1065045
10. Hoare C. A. R. An axiomatic basis for computer programming // Communications of the ACM. 1969. Vol. 12. Issue 10. P. 576–580. https://doi.org/10.1145/363235.363259
11. Dijkstra E. W. Guarded commands, nondeterminacy and formal derivation of programs // Communications of the ACM. 1975. Vol. 18. Issue 8. P. 453–457. https://doi.org/10.1145/360933.360975

12. Srivastava S., Gulwani S., Foster J. S. From program verification to program synthesis // Proceedings of the 37th Annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. Association for Computing Machinery, 2010. P. 313–326. https://doi.org/10.1145/1706299.1706337
13. Katz G., Barrett C., Dill D. L., Julian K., Kochenderfer M. J. Reluplex: An efficient SMT solver for verifying deep neural networks // Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24–28, 2017, Proceedings, Part I. Cham: Springer, 2017. P. 97–117. https://doi.org/10.1007/978-3-319-63387-9_5
14. Guidotti R., Monreale A., Ruggieri S., Turini F., Giannotti F., Pedreschi D. A survey of methods for explaining black box models // ACM Computing Surveys (CSUR). 2018. Vol. 51. Issue 5. Article number: 93. P. 1–42. https://doi.org/10.1145/3236009
15. Schkufza E., Sharma R., Aiken A. Stochastic superoptimization // ACM SIGARCH Computer Architecture News. 2013. Vol. 41. Issue 1. P. 305–316. https://doi.org/10.1145/2490301.2451150

Поступила в редакцию 19.08.2024
Принята к публикации 25.11.2024

Джудакизаде Милад, аспирант, кафедра теоретических основ информатики, Удмуртский государственный университет, 426034, Россия, г. Ижевск, ул. Университетская, 1.
ORCID: https://orcid.org/0000-0002-6167-6237
E-mail: dzhudakizade@udsu.ru

Бельтюков Анатолий Петрович, д. ф.-м. н., профессор, кафедра теоретических основ информатики, Удмуртский государственный университет, 426034, Россия, г. Ижевск, ул. Университетская, 1.
ORCID: https://orcid.org/0000-0002-3433-9067
E-mail: belt.udsu@mail.ru

Цитирование: М. Джудакизаде, А. П. Бельтюков. Двухуровневая реализация логических формул для дедуктивного синтеза программ // Вестник Удмуртского университета. Математика. Механика. Компьютерные науки. 2024. Т. 34. Вып. 4. С. 469–485.