COMPUTER SCIENCE 2026. Vol. 36. Issue 1. Pp. 159–178.

MSC2020: 03F65

© *M. Joudakizadeh, A. P. Beltiukov*

DEDUCTIVE PROGRAM SYNTHESIS USING LOGIC PROGRAMMING

This article presents an approach to deductive program synthesis using Gentzen's sequent calculus within the framework of logic programming. By leveraging sequent calculus as a formal system for structured logical inference, our method automates the derivation of provably correct programs from specifications expressed in negation-free first-order predicate logic. We formalize the syntax and semantics of sequent calculus, implementing its core inference rules (introduction and elimination rules) as predicates in logic programming to enable scalable synthesis. Practical examples demonstrate the transformation of logical specifications into executable programs. The approach ensures formal correctness through a constructive semantics inspired by Kleene's realizability, with synthesized programs operating in a subrecursive language to guarantee termination. We evaluate the method's strengths, including its reliability for safety-critical systems, and its limitations, such as computational complexity for unbounded constructions. Compared to AI-driven synthesis, our approach prioritizes formal guarantees, complementing modern trends like relational programming. Future research directions include optimizing computational efficiency and extending applicability to complex real-world problems.

Keywords: deductive synthesis, sequent calculus, logic programming, Prolog, logical inference, programming automation, artificial intelligence.

DOI: 10.35634/vm260108

Introduction

Deductive program synthesis represents a transformative approach in computer science, aiming to automate the creation of executable programs from formal logical specifications, thereby reducing human error and enhancing software reliability. Consider a scenario where a safety-critical system, such as an autonomous vehicle's navigation software, is automatically derived from a high-level logical specification, ensuring provable correctness without manual coding. This vision, central to artificial intelligence since the 1950s [1], leverages formal logic to systematically construct programs that are provably correct, making deductive synthesis particularly valuable for safety-critical applications in aerospace, medical systems, and beyond [2, 3].

This study proposes a novel methodology for deductive program synthesis, utilizing Gentzen's sequent calculus [4] implemented within the logic programming framework of Prolog [5]. In our approach, Prolog is employed not in its standard operational semantics but as a logical language, utilizing a proof-building strategy with gradual increase in inference depth (iterative deepening) to avoid loops and stack overflows, ensuring completeness for finite proofs [6]. Our method accepts specifications in a negation-free first-order predicate logic and automates the synthesis process by encoding sequent calculus inference rules as Prolog predicates, enabling scalable and verifiable program generation. The synthesized programs operate within a subrecursive programming language, ensuring termination [7], and are supported by a correctness theorem that guarantees adherence to specifications under all valid conditions.

The objectives of this study are multifaceted:

- Formalize the syntax and semantics of sequent calculus for representing logical specifications.
- Implement inference rules in logic programming to automate the synthesis process.

- Demonstrate the transformation of logical specifications into executable programs through practical examples.
- Evaluate the approach's strengths, limitations, computational complexity, and scalability.
- Explore potential applications in automated program synthesis, artificial intelligence, and safety-critical systems.

Sequent calculus, with its structured representation of logical inference as sequents ($G \to B$), where G is a finite set of premises (formulas or variables) and B is the conclusion formula, asserts that G entails B, providing a robust framework for deductive synthesis [4]. Logic programming, grounded in Horn clauses and resolution mechanisms, offers an ideal platform for implementing these rules, supporting declarative programming and automated logical inference [5, 8]. Our methodology integrates the theoretical rigor of sequent calculus with the practical automation of logic programming, ensuring provable correctness of synthesized programs.

In the broader context of program synthesis, our work contrasts with modern AI-driven approaches, such as those using large language models (LLMs) for code generation [9–12]. While LLMs excel in rapid prototyping, they often lack formal correctness guarantees, producing code prone to errors [13]. Our deductive approach prioritizes provable correctness, making it suitable for domains where reliability is critical. Additionally, our work aligns with emerging trends in relational programming, such as finite-choice logic programming [14], which extends logic programming to enumerate multiple solutions, potentially enabling applications in constraint solving and type inference [15, 16]. Recent advances in learning deductive reasoning [17] suggest a hybrid future where AI-generated specifications could be refined using our deductive framework, addressing challenges in specification ambiguity [18].

This study systematizes existing deductive synthesis approaches while proposing novel methods for implementing sequent calculus in logic programming. We address practical challenges, such as quantifier handling and computational complexity, and suggest strategies for optimization and integration with other paradigms, such as functional programming [19]. By bridging theoretical foundations with practical automation, our work advances the vision of reliable, automated software development, continuing the legacy of logical foundations discussed at symposia like the Symposium on Logical Foundations of Computer Science held in Pereslavl-Zalessky [20].

§ 1. Related works

Deductive program synthesis, rooted in formal logic and proof theory, is a vibrant field that integrates theoretical rigor with practical automation to address the complexity of modern software systems. This section provides a comprehensive review of foundational and contemporary contributions, organized into six thematic areas: foundational logic, logic programming, classical deductive synthesis, modern synthesis paradigms, relational programming, and formal verification tools. This structure contextualizes our work within the broader landscape of program synthesis.

1.1. Foundational logic and proof theory

The theoretical foundations of deductive synthesis are rooted in formal logic and proof theory. Gentzen's sequent calculus [4], introduced in the 1930s, provides a structured framework for logical inference, representing reasoning as sequents ($G \to B$). Its modularity and cut-elimination properties make it ideal for automated theorem proving and program synthesis, forming the backbone of our methodology. Kleene's work on metamathematics [21] introduced constructive realizability, where proofs correspond to executable programs, a principle we adapt for synthesis. Kolmogorov's interpretation of intuitionistic logic [22] and Gödel's dialectica interpretation [23] provide semantic foundations for mapping logical formulas to computational structures. Markov's

theory of algorithms [24] formalizes computational processes, ensuring termination in our subrecursive language [7]. These works collectively establish the logical foundations for deductive synthesis, distinguishing it from less formal approaches.

1.2. Logic programming

Logic programming, conceptualized by Kowalski [8], treats logical formulas as executable programs, enabling automated reasoning through Horn clauses. The development of Prolog by Colmerauer and Roussel [5] introduced a declarative language that has become a cornerstone for deductive synthesis due to its robust support for logical queries. Warren's optimizations [25] enhanced logic programming's efficiency, making it suitable for encoding complex inference rules, as in our implementation. Bratko's work [6] demonstrates logic programming's versatility in AI applications, while Genesereth and Chaudhri [26] highlight its continued relevance. Miller's survey [27] connects logic programming to proof theory, emphasizing the structural properties of sequents that support our approach. In contrast, functional programming, as described by Bird [19], offers a different paradigm, but our focus on logic programming leverages its declarative nature for automated synthesis.

1.3. Classical deductive program synthesis

Deductive synthesis, pioneered by Manna and Waldinger [2], treats program synthesis as a theorem-proving task, using transformation rules and mathematical induction. Nepejvoda's studies [28, 29] highlight the synergy between proof theory and programming, advocating for formal methods in automation. Tyugu's early work [30, 31] introduced nonprocedural specifications, a concept we extend with our negation-free predicate logic. Beltiukov's method for synthesizing recursive programs [16] addresses bounded constructions, aligning with our subrecursive constraints. Joudakizadeh and Beltiukov [32] proposed a two-level realization of logical formulas, combining features of Kleene's realizability and Gödel's dialectica interpretation to ensure correctness and support for synthesized programs, which informs our theoretical framework. Arhangelsky and Taitslin's logic for data description [15] supports specification formalization, relevant to our approach. These classical works provide the theoretical and practical foundations for our methodology, emphasizing correctness and automation.

1.4. Modern program synthesis paradigms

Modern program synthesis encompasses deductive, inductive, and AI-driven approaches. Gulwani et al.'s survey [1] categorizes techniques into enumerative search, constraint solving, stochastic search, and programming by examples, noting that deductive synthesis excels in correctness but faces scalability challenges. Huang et al.'s reconciliation of enumerative and deductive synthesis [33] proposes cooperative techniques for conditional linear integer arithmetic, inspiring our strategies. Feser's work on database applications [34] demonstrates deductive synthesis for data-driven tasks, a potential application for our method. Todorova and Orozova [35] highlight educational challenges in deductive methods, emphasizing the need for accessible implementations, which our logic programming-based approach addresses. The Logic at Botik '89 symposium [20] underscored the integration of formal methods with practical applications, a principle our work upholds.

AI-driven synthesis, particularly with large language models (LLMs), has gained prominence. Hou et al.'s review [9] and Ramirez-Rueda et al.'s survey [13] highlight LLMs' strengths in rapid code generation but note their lack of formal guarantees. Sevenhuijsen et al.'s VeCoGen [11] and Ferdowsi et al.'s validation of AI-generated code [12] explore formal verification of LLM outputs, suggesting a hybrid future where deductive methods refine AI-generated code. Barke et al.'s Hysynth [10] uses LLMs to guide synthesis, while Kalyan et al.'s neural-guided deductive search [18] integrates machine learning with deductive reasoning. Morishita et al.'s

work [17] on learning deductive reasoning proposes synthetic corpora, a direction we consider for handling ambiguous specifications. Ding and Qiu's concurrent string transformation synthesis [36] extends deductive methods to new domains, reinforcing their versatility.

1.5. Relational programming and finite-choice logic

Relational programming, exemplified by finite-choice logic programming [14], extends logic programming to enumerate multiple solutions, contrasting with Datalog's single-model semantics. This paradigm, rooted in answer set programming, aligns with our method's potential to handle problems requiring multiple valid solutions, such as constraint solving or type inference. Our sequent calculus rules can be viewed as relational transformations, suggesting future integration with finite-choice frameworks to enhance expressiveness. This connection positions our work at the intersection of deductive synthesis and relational programming, offering opportunities for bidirectional program synthesis.

1.6. Formal verification and tools

Formal verification tools like Isabelle/HOL [37] and model checking [38] complement deductive synthesis by verifying synthesized programs. Martin-Löf's constructive mathematics [39] and Constable's work on automatic program writers [40] provide theoretical support for our constructive semantics. These tools and theories ensure that our synthesized programs meet stringent correctness criteria, aligning with the principles discussed at Logic at Botik '89 [20]. Recent advancements, such as Andriushchenko et al.'s deductive controller synthesis [41], extend these ideas to probabilistic hyperproperties, suggesting potential applications of our methodology in stochastic systems.

1.7. Positioning of our work

Our work distinguishes itself by integrating the theoretical rigor of sequent calculus with the practical automation of logic programming, addressing both classical and modern challenges in deductive synthesis. Unlike Manna and Waldinger's manual transformations [2], our approach automates the synthesis process, improving scalability. Compared to Huang et al.'s hybrid framework [33], we maintain the purity of deductive reasoning while leveraging logic programming's efficiency. Our methodology is more general than domain-specific approaches like Feser's [34] or Itzhaky et al.'s [42], applicable to a wide range of logical programming tasks. By building on the proof-theoretic foundations surveyed by Miller [27] and incorporating insights from relational programming [14], our work offers a robust framework for automated program synthesis, with potential applications in AI, safety-critical systems, and beyond.

§ 2. Methodology

The methodology presented in this article leverages sequent calculus as a formal system for deductive program synthesis within the logic programming framework. Developed by Gerhard Gentzen in the 1930s [4], sequent calculus provides a structured approach to logical inference, enabling both the proof of theorems in first-order logic and the transformation of logical specifications into executable programs. This section outlines the theoretical foundations of sequent calculus, including its syntax and semantics, its implementation in logic programming, using Prolog with modified semantics, and practical examples illustrating the deductive synthesis process. We emphasize the formalization of inference rules, their programmatic implementation, and the approach's applicability to real-world tasks such as automated data processing, constraint solving, and verification in safety-critical systems [3, 41]. The illustrative examples presented in subsequent sections are precise (unambiguous, with well-defined logical structures) and complete (covering all possible cases), ensuring correct synthesis [2].

2.1. The logical language used

We adopt a positive (negation-free) first-order predicate logic language to ensure the reliability of synthesized programs, where reliability refers to provable correctness (adherence to specifications) and guaranteed termination under valid conditions, rather than efficiency. Instead of directly negating a formula A, we use a formula of the form $(A \Rightarrow \text{Error})$, where Error is a formula whose realizations are messages about specific errors detectable in the domain. This approach ensures that synthesized programs explicitly detect and handle all possible errors, enhancing correctness and robustness.

The language includes subject variables and constants, predicates, and logical connectives: conjunction ($\&$), disjunction ($\vee$), implication ($\Rightarrow$), and quantifiers (universal $\forall$ and existential $\exists$). For simplicity, functions are excluded in this version, focusing on predicate-based specifications that can be extended as needed.

2.2. Constructive realization semantics of the language

For program synthesis, we adopt a constructive realization semantics for the language's formulas, inspired by Kleene's work [21]. Each constant atom formula is associated with a set of objects called *realizations*, which are constructive objects uniquely defined by strings of symbols in a fixed alphabet, excluding parentheses or commas for simplicity.

The domain consists of constructive objects, considered as subject constants, similarly free of parentheses or commas. The notation $b : B$ indicates that object b is a realization of formula B. Realizations of compound formulas are defined as follows:

$$(b, c) : (B\&C) \Leftrightarrow (b : B)\&(c : C),$$

$$(i, b) : (B_0 \vee B_1) \Leftrightarrow b : B_i,$$

$$(d, b) : \exists x B(x) \Leftrightarrow b : B(d),$$

where d is a subject constant, and $B(d)$ is the result of substituting d for all free instances of variable x in $B(x)$.

For universal quantification and implication, we assume an algorithmic language with programs free of unbalanced parentheses or commas outside parentheses, executed by some interpreter u. The language is subrecursive, ensuring termination of all programs on all inputs [7]. We employ subrecursiveness to avoid dealing with program inapplicability in certain cases: the interpreter of a subrecursive programming language is always applicable. In practice, a universal (Turing-complete) language is never required — a subrecursive class suffices, simplifying the definitions of connections between programs and formulas [7]. The result of executing program f on input d is denoted $u(f, d)$. Realizations are defined as:

$$f : \forall x B(x) \Leftrightarrow \forall d : U(u(f, d) : B(d)),$$

$$f : (B \Rightarrow C) \Leftrightarrow \forall b : B(u(f, b) : C),$$

where $\forall d : U$ ranges over all subject constants in the domain, and $\forall b : B$ ranges over all realizations of B.

Additional constraints on the algorithmic language ensure compatibility with sequent calculus rules, discussed below. The subrecursive nature guarantees termination, while the constructive semantics ensures correctness, as realizations correspond to executable computations satisfying the logical specifications.

The approach's applicability to real-world tasks is demonstrated through its potential in domains requiring formal correctness, such as safety-critical systems (e. g., aviation software verification), database query optimization, and constraint-based problem solving (e. g., scheduling

or resource allocation). While subsequent sections focus on illustrative examples to highlight mechanics, the methodology scales to such complex tasks by leveraging logic programming's efficient resolution mechanism, as seen in expert systems and knowledge bases [6].

§ 3. Theoretical foundations of sequent calculus

Sequent calculus, introduced by Gerhard Gentzen in the 1930s [4], is a formal system in mathematical logic designed for structured representation of logical inference. Its modularity and explicit inference steps, supported by the cut-elimination theorem, make it ideal for deductive program synthesis [2]. The cut-elimination theorem ensures that proofs can be constructed without the cut rule, simplifying automation by providing a normal form for proofs [4,27]. A sequent is defined as $G \to D$, where G is a finite sequence of premises (formulas or variables implicitly associated with universal quantifiers), and D is a single logical formula representing the conclusion. The sequent $G \to D$ asserts that the premises in G entail the conclusion D. Unlike resolution-based systems [8] or Hilbert-style predicate calculus [21], sequent calculus explicitly represents inference steps, enabling automation of proof construction and program synthesis in logical programming languages like Prolog [5].

The core rules of sequent calculus — axioms, generation rules, and usage rules — form the foundation for proof construction and program synthesis. Axioms establish basic assertions, generation rules construct compound formulas (e. g., conjunctions, disjunctions, implications, and quantifiers) from simpler ones, and usage rules decompose compound premises during inference. These rules are systematically applied to derive proofs that are transformed into executable programs, leveraging constructive realizability semantics [21, 23]. This section details the syntax, semantics, and implementation of these rules, emphasizing their role in automating deductive synthesis for applications like constraint solving and safety-critical system verification [3, 41].

3.1. Axioms

Axioms in sequent calculus are fundamental assertions requiring no further proof, serving as the starting point for proof construction. They take the form:

$$pr_i : G_n, \ldots, G_i, \ldots, G_0 \to G_i,$$

where pr_i (Premise Reproduction or PRojection) is the axiom name, i is the index parameter, and G_i is the i-th element of the premise sequence G. Formally, an axiom is expressed as:

$$pr_i : (G \to B) \leftarrow B = G_i,$$

where $=$ denotes syntactic equality, ensuring a character-by-character match between the premise and the conclusion. This validates premises as conclusions, providing a foundation for complex proofs.

Axioms and rules are written as:

$$\text{name} : \text{conclusion} \leftarrow \text{premises},$$

where name is the rule or axiom identifier, conclusion is the sequent being derived, and premises are the applicability conditions or subgoals.

3.2. Generation rules

Generation rules construct compound formulas from simpler ones, enabling the synthesis of complex logical specifications. The generation rules are:

- **Conjunction Generation (cg)**:

$$cg(f,h) : (G \to (B\&C)) \leftarrow f : (G \to B), h : (G \to C).$$

This rule combines two sequents proving formulas B and C to derive their conjunction $B\&C$. In constructive realizability semantics [21], the realization of $B\&C$ is a pair (b, c), where $b : B$ and $c : C$, ensuring correct computation of both components.

- **Disjunction Generation (dg)**:

$$dg_i(f) : (G \to (B_0 \vee B_1)) \leftarrow f : (\Gamma \to B_i), i \in \{0, 1\}.$$

This rule proves a disjunction $B_0 \vee B_1$ by proving one of B_0 or B_1, with index i indicating the selected formula. The realization is a pair (i, b), where $b : B_i$, aligning with disjunction semantics [22].

- **Implication Generation (ig)**:

$$ig(f) : (G \to (B \Rightarrow C)) \leftarrow f : (G, B \to C).$$

This rule forms an implication $B \Rightarrow C$ by adding B to the premise sequence and proving C. The realization is a program f such that for any $b : B$, $u(f, b) : C$, where u is an interpreter.

- **Universal Quantifier Generation (ag)**:

$$ag(f) : (G \to \forall x B(x)) \leftarrow f : (G, a \to B(a)), \text{ where } a \text{ is a fresh name.}$$

This rule introduces a universal quantifier $\forall x B(x)$, using a fresh name a not in Γ, preventing variable conflicts [4]. The realization is a program f such that for any subject constant d, $u(f, d) : B(d)$.

- **Existential Quantifier Generation (eg)**:

$$eg_i(f) : (G \to \exists x B(x)) \leftarrow f : (G \to B(a)), G[i] = a.$$

This rule proves an existential quantifier $\exists x B(x)$ by substituting a name a present in G. The realization is a pair (d, b), where $b : B(d)$.

3.3. Usage rules

Usage rules decompose compound premises during inference, enabling analysis of complex logical structures. The key usage rules are:

- **Conjunction Usage (cu)**:

$$cu_i(f) : (G \to D) \leftarrow f : (G, B, C \to D), G_i = (B\&C).$$

This rule replaces a conjunction $B\&C$ in the premise sequence with its components B and C, allowing their use in proving D.

- **Disjunction Usage (du)**:

$$du[i](f_0, f_1) : (G \to D) \leftarrow f_j : (G, B_j \to D), j \in \{0, 1\}, G_i = (B_0 \vee B_1).$$

Known as case analysis, this rule requires proving D for both branches of a disjunction $B_0 \vee B_1$, ensuring completeness [4].

- **Implication Usage (iu)**:

$$iu_i(f,h) : (G \to D) \leftarrow f : (G \to B), h : (\Gamma, C \to D), G_i = (B \Rightarrow C).$$

This rule uses an implication $B \Rightarrow C$ by proving B and using C to derive D, reflecting implication semantics.

- **Universal Quantifier Usage (au)**:

$$au_{i,j}(f) : (G \to D) \leftarrow f : (G, B(a) \to D), G_i = \forall x B(x), G_j = a.$$

This rule substitutes a name a into a universal quantifier $\forall x B(x)$ to use $B(a)$ in proving D.

- **Existential Quantifier Usage (eu)**:

$$eu_i(f) : (G \to D) \leftarrow f : (G, a, B(a) \to D), G_i = \exists x B(x), \text{ where } a \text{ is a fresh name.}$$

This rule introduces a fresh name a for the variable x in $\exists x B(x)$, facilitating inference without conflicts.

3.4. Implementation in logic programming

For practical implementation, we use logic programming, specifically Prolog with a modified operational semantics, whose declarative nature and resolution mechanism make it ideal for automating sequent calculus inference [5, 8]. The examples of core rules are implemented as follows:

Listing 1: Prolog implementation of sequent calculus rules

```
% Axiom (Premise Reproduction)
seq(pr(I), G, B) :- extr(G, I, B).
% seq(F,G,B) means sequent F:G->B
% where F is a proof term, G is a premises list, B is a conclusion
   formula
% B is the I-th element of G

% Conjunction Generation
seq(cg(F, H), G, and(B, C)) :- seq(F, G, B), seq(H, G, C).

% Disjunction Generation
seq(dg0(F), G, or(B0, B1)) :- seq(F, G, B0).
seq(dg1(F), G, or(B0, B1)) :- seq(F, G, B1).

% Implication Generation
seq(ig(F), G, imp(B, C)) :- seq(F, con(G, B), C).
% con adds new element to the premis

% Universal Quantifier Generation
seq(ag(F), G, all(X, BX)) :- seq(F, con(G, A), BA),
    subst(BX, X, A, BA), newname(G, A).
%subst(BX, X, A, BA) --- substituting A instead of X into BX gives BA
%newname(G,A) generates a fresh name for G

% Existential Quantifier Generation
seq(eg(I, F), G, exi(X, BX)) :- seq(F, G, BA), extr(G, I, A),
```

```
    subst(BX, X, A, BA), name(A).
% A is a name

%% Other rules are programmed similarly...

% Auxiliary Predicates
extr(con(G, B), 0, B).
extr(con(G, C), s(I), B) :- extr(G, I, B).

%% Other predicates definitions are omitted...
```

The predicate $seq(f, G, B)$ represents the sequent $f : (G \to B)$, where f is the proof term, Γ is the premise sequence, and B is the conclusion. Auxiliary predicates $extr$, $subst$, $newname$, $name$, and function con, handle premise extraction, premise addition, substitution, and name management, with $newname$ ensuring fresh names to avoid conflicts [4]. To prevent infinite loops in Prolog's default depth-first search, we adopt iterative deepening, as recommended in [6], ensuring termination for subrecursive programs [7]. Verification, defined as proof-theoretic validation, ensures that synthesized programs implement the logical specifications [27], critical for safety-critical applications [3].

3.5. Deductive synthesis process

The deductive synthesis process transforms a logical specification into an executable program through:

1. **Problem formalization**: The problem is expressed as a sequent $G \to B$, where G is a sequence of premises and B is the conclusion. The specification must be precise (unambiguous, well-defined) and complete (covering all cases) [2].

2. **Logical inference construction**: Sequent calculus rules are applied to build a proof tree validating the sequent, using axioms, generation, and usage rules. The proof tree, represented as a term (e. g., $cg(f, h)$), encodes the inference structure.

3. **Program extraction**: The proof tree is transformed into a program, where each rule corresponds to an operator. For instance, $cg(f, h)$ translates to a program computing realizations for $B\&C$ by combining subprograms for B and C. The program's correctness — adherence to the specification — is ensured by the soundness of sequent calculus [4, 27].

4. **Program execution**: The program can be executed in Prolog or translated into other languages. For example, it is not hard to program operators pr, ..., eu in JavaScript, and so proof terms can be executed directly as programs. The subrecursive nature ensures termination [7], suitable for tasks like constraint solving or database query synthesis [34].

This process transitions from abstract specifications to executable programs with provable correctness, defined as adherence to the specification via realizability semantics and sequent calculus soundness [21, 23, 27]. The approach applies to real-world tasks like data processing, constraint satisfaction (e. g., scheduling), and safety-critical system verification [3, 41].

§ 4. Application examples

To illustrate the proposed methodology, we consider several test problems demonstrating the deductive synthesis process.

4.1. Example 1: Proving a simple sequent

Consider the task of proving the sequent:

$$f_0 :\to \forall x(P(x) \Rightarrow \exists y P(y)).$$

This sequent asserts that for any x, if $P(x)$ is true, there exists a y for which $P(y)$ is also true. The inference process is as follows:

$$f_0 = ag(f_1),$$
$$f_1 : a \to (P(a) \Rightarrow \exists y P(y)),$$
$$f_1 = ig(f_2),$$
$$f_2 : a\ P(a) \to \exists y P(y),$$
$$f_2 = eg_1(f_3),$$
$$f_3 : a\ P(a) \to P(a),$$
$$f_3 = pr_0.$$

The final inference is $f_0 = ag(ig(eg_1(pr_0)))$. It can be used as a program. Here operators have obvious algorithmic semantics, that allows them to transform initial realization of the axiom to a desired program, that here combines these arguments to the pair that realizes required conclusion. For example, operator eg works as follows: if $f_2 = eg_i(f_3)$, then f_2 is the function, that is defined by the equation

$$f_2(g) = (g_i, f_3(g));$$

if g is the list of values corresponding to the premise $a,\ P(a)$, then $g_1 = a$ (first element of g), $g_0 = P(a)$. All other operators can be defined in similar ways. Of course, here and in the following examples we observe some requirements to the algorithmic language, that should be able to express this proof-as-a-program.

This example demonstrates how the rules ag, ig, eg, and pr are used to construct a proof. The implementation automates this process, generating a program that confirms the sequent's correctness.

4.2. Example 2: Computing values for a sequent

Consider a problem involving the sequent:

$$f : a\ (P(a)\&Q(a)) \to (\exists x P(x)\&\exists y Q(y)).$$

The inference process is as follows:

$$f = cu_0(cg(eg_3(f_1), eg_3(f2))),$$
$$f_1 : a\ (P(a)\&Q(a))\ P(a)\ Q(a) \to P(a),$$
$$f_1 = pr_1,$$
$$f_2 : a\ (P(a)\&Q(a))\ P(a)\ Q(a) \to Q(a),$$
$$f_2 = pr_0.$$

The final inference is $f = cu_0(cg(eg_3(pr_1), eg_3(pr_0)))$.

It is not hard to see, that for value computation we have:

$$f(3,(4,6)) = ((3,4),(3,6)).$$

4.3. Example 3: Computing values for a complex sequent

Consider a problem involving the sequent:

$$f : \exists x P(x)\ (\forall y(P(y) \Rightarrow \exists z Q(y,z))) \rightarrow \exists u \exists w Q(u,w).$$

This sequent asserts that if there exists an x such that $P(x)$ is true, and for all y, if $P(y)$ is true then there exists a z such that $Q(y,z)$ is true, then there exist u and w such that $Q(u,w)$ is true.

The inference process is as follows:

$$f = eu_1(f_1),$$
$$f_1 \colon \exists x P(x)\ \forall y(P(y) \Rightarrow \exists z Q(y,z))\ a\ P(a) \rightarrow \exists u \exists w Q(u,w),$$
$$f_1 = eg_1(f_2),$$
$$f_2 \colon \exists x P(x)\ \forall y(P(y) \Rightarrow \exists z Q(y,z))\ a\ P(a) \rightarrow \exists w Q(a,w),$$
$$f_2 = au_{2,1}(f_3),$$
$$f_3 \colon \exists x P(x)\ \forall y(P(y) \Rightarrow \exists z Q(y,z))\ a\ P(a)\ (P(a) \Rightarrow \exists z Q(a,z)) \rightarrow \exists w Q(a,w),$$
$$f_3 = iu_0(f_4, f_5),$$
$$f_4 \colon \exists x P(x)\ \forall y(P(y) \Rightarrow \exists z Q(y,z))\ a\ P(a)\ (P(a) \Rightarrow \exists z Q(a,z)) \rightarrow P(a),$$
$$f_4 = pr_1,$$
$$f_5 \colon \exists x P(x)\ \forall y(P(y) \Rightarrow \exists z Q(y,z))\ a\ P(a)\ (P(a) \Rightarrow \exists z Q(a,z))\ \exists z Q(a,z) \rightarrow \exists w Q(a,w),$$
$$f_5 = eu_0(f_6),$$
$$f_6 \colon \exists x P(x) \forall y(P(y) \Rightarrow \exists z Q(y,z))\, a P(a) (P(a) \Rightarrow \exists z Q(a,z)) \exists z Q(a,z)\ b\, Q(a,b) \rightarrow \exists w Q(a,w),$$
$$f_6 = eg_1(f_7),$$
$$f_7 \colon \exists x P(x)\ \forall y(P(y) \Rightarrow \exists z Q(y,z))\ a\ P(a)\ (P(a) \Rightarrow \exists z Q(a,z))\ \exists z Q(a,z)\ b\ Q(a,b) \rightarrow Q(a,b),$$
$$f_7 = pr_0,$$

The final inference is:

$$f = eu_1(eg_1(au_{2,1}(iu_0(pr_1, eu_0(eg_1(pr_0)))))),$$

For value computation, given the function:

$$g(x,y) = (x + y, x * y),$$

we compute:

$$f((2,3), g) = (2, (5,6)).$$

These examples illustrate how logical inferences are transformed into computational functions. The results confirm the correctness of the synthesized programs.

§ 5. Results

This section presents the results of applying the deductive synthesis methodology based on sequent calculus to selected test problems. The problems demonstrate the construction of logical inferences, their transformation into executable programs, and the computation of specific values based on given functions. The results highlight the correctness, automation, and flexibility of the proposed approach, as well as its potential for addressing complex tasks in automated programming. The implementation automated both the proof of the sequents and the value computation, demonstrating the practical applicability of the approach.

5.1. Analysis of the results

The analysis of the test problems underscores the effectiveness and versatility of sequent calculus as a tool for deductive program synthesis. Key aspects identified during the experiments include:

- **Correctness:** The considered sequents were successfully proven using sequent calculus rules. Computed values in the test problems confirmed the alignment of synthesized programs with the original specifications. This reflects the formal rigor of the approach, which ensures the absence of logical errors in proofs.
- **Automation:** The implementation fully automated the inference and computation processes, minimizing the need for manual intervention. This is particularly significant for scaling the approach to more complex tasks, where manual proof construction becomes impractical.
- **Flexibility:** Sequent calculus proved applicable to a wide range of logical constructs, including basic operations (conjunction, disjunction), quantifiers (existential and universal), and complex implications. The diversity of test problems demonstrates the approach's versatility.
- **Practical significance:** The synthesized programs can be directly applied in real-world applications, such as data processing or program verification.
- **Modularity:** The structure of sequent calculus, separating rules into generation and usage, simplifies the construction of complex proofs. This allows tasks to be divided into subtasks, enhancing readability and reproducibility of results.

5.2. Comparison with alternative approaches

To provide a deeper analysis, a comparative review was conducted with other deductive synthesis methods, such as transformational synthesis [2] and inductive synthesis [1]. Unlike transformational synthesis, which requires sequential transformation of specifications into programs, sequent calculus offers a more formalized and automated process based on logical inference. Compared to inductive synthesis, which relies on input-output examples, the proposed approach ensures greater rigor through formal proofs, although it requires more precise specifications.

5.3. Limitations and improvements

Despite the achieved results, the approach has certain limitations:

- **Computational complexity:** Constructing proofs for sequents with numerous premises or quantifiers can be computationally expensive.
- **Specification requirements:** Logical specifications must be clearly defined, which can be challenging in real-world tasks with incomplete or ambiguous information.
- **Limited interpretability:** Proof trees, while formally correct, may be difficult for developers unfamiliar with sequent calculus to understand.

To address these limitations, the following improvements are proposed:

- **Algorithm optimization:** Techniques such as caching intermediate results or parallel execution in logic programming can reduce computational complexity [25].

- **Integration with other methods:** Combining deductive synthesis with inductive approaches can simplify specification formalization by leveraging data examples to refine requirements.
- **Improved interpretability:** Developing visual tools for displaying proof trees (e. g., using graphical libraries) can enhance the approach's accessibility to a broader audience.

5.4. Application prospects

The results of the test problems highlight the potential of sequent calculus for addressing a wide range of computer science challenges. Potential application areas include:

- **Automated generation of verified programs:** Sequent proofs can be used to provide software correctness, particularly in safety-critical systems such as aviation or medical equipment.
- **Intelligent systems:** Automated logical inference can be applied in expert systems and knowledge processing systems requiring complex logical query handling.
- **Data processing:** Synthesized programs can be used to analyze large datasets with well-defined logical dependencies.

Conclusion

This study aimed to develop and investigate a methodology for deductive program synthesis using sequent calculus within the framework of logic programming. The primary goal was to create an approach that automates the transformation of formal logical specifications into executable programs. This involved reducing human effort in software development, ensuring the correctness of synthesized programs, and enhancing programming efficiency through rigorous logical inference implemented via sequent calculus and logic programming. The results demonstrate achievement of the stated objectives, with examples validating formalization, implementation, and evaluation.

The results of the study demonstrate that the proposed methodology successfully addresses the stated objectives across various types of logical specifications. Analysis of test problems showed that applying sequent calculus rules implemented in logic programming enables the automatic generation of correct programs. These examples highlight the method's ability to ensure formal rigor and practical applicability of synthesized programs.

This work makes a significant contribution to the field of automated programming by proposing a formal and systematic approach to synthesizing programs from logical specifications. The use of sequent calculus provides a structured and modular logical inference process, enhancing the reliability, reproducibility, and maintainability of results. Integration with logic programming amplifies the method's practical value, leveraging the advantages of declarative programming to automate complex computations. Compared to alternative approaches, such as transformational or inductive synthesis, the proposed methodology stands out for its high degree of formalization and automation, making it particularly valuable for tasks requiring strict logical precision.

Despite its successes, the method has several limitations that must be considered. First, the computational complexity of constructing proofs for complex sequents can be significant, limiting the approach's scalability when handling large or deeply nested specifications. Second, the method requires precise and complete logical specifications, which may be challenging in real-world scenarios with ambiguous or partially defined requirements. Finally, while the generated proof trees are formally correct, they may be difficult for developers without deep knowledge of sequent calculus to interpret, reducing the method's accessibility to a broader audience.

To address these limitations, the following directions for future research are proposed. Optimizing computational processes, such as through caching intermediate results or employing more efficient search strategies in logic programming, could reduce computational overhead and enhance scalability. Integrating deductive synthesis with inductive approaches could enable handling less formalized specifications by using data examples to refine requirements. Additionally, developing visualization tools and simplifying proof trees could improve the interpretability of results, making the method more accessible for practical applications. These improvements pave the way for broader adoption of the proposed approach in areas such as automated program verification, intelligent system development, and big data processing.

REFERENCES

1. Gulwani S., Polozov O., Singh R. Program synthesis, *Foundations and Trends in Programming Languages*, 2017, vol. 4, nos. 1–2, pp. 1–119. https://doi.org/10.1561/2500000010
2. Manna Z., Waldinger R. A deductive approach to program synthesis, *ACM Transactions on Programming Languages and Systems*, 1980, vol. 2, no. 1, pp. 90–121. https://doi.org/10.1145/357084.357090
3. Smith D. R., Nedunuri S. Deductive model refinement, *NASA Formal Methods. 16th International Symposium, NFM 2024, Moffett Field, CA, USA, June 4–6, 2024, Proceedings*, Cham: Springer, 2024, pp. 147–165. https://doi.org/10.1007/978-3-031-60698-4_9
4. Gentzen G. Investigations into logical deduction, *Studies in Logic and the Foundations of Mathematics*, 1969, vol. 55, pp. 68–131. https://doi.org/10.1016/S0049-237X(08)70822-X
5. Colmerauer A., Roussel P. The birth of Prolog, *History of programming languages — II*, New York: ACM, 1996, pp. 331–367. https://doi.org/10.1145/234286.1057820
6. Bratko I. *Prolog programming for artificial intelligence*, New York: Addison Wesley, 2001. https://archive.org/details/prologprogrammin0000brat_l1m9
7. Constable R. L., Borodin A. B. Subrecursive programming languages, part I: Efficiency and program structure, *Journal of the ACM*, 1972, vol. 19, issue 3, pp. 526–568. https://doi.org/10.1145/321707.321721
8. Kowalski R. Predicate logic as programming language, *Information Processing, Proceedings of the 6th (IFIP) Congress 1974*, Amsterdam: North-Holland, 1974, pp. 569–574. https://www.researchgate.net/publication/221330242_Predicate_Logic_as_Programming_Language
9. Hou Xinyi, Zhao Yanjie, Liu Yue, Yang Zhou, Wang Kailong, Li Li, Luo Xiapu, Lo David, Grundy John, Wang Haoyu. Large Language Models for software engineering: A systematic literature review, *ACM Transactions on Software Engineering and Methodology*, 2024, vol. 33, no. 8, article no. 220, pp. 1–79. https://doi.org/10.1145/3695988
10. Anaya Gonzalez E., Barke S., Berg-Kirkpatrick T., Kasibatla S. R., Polikarpova N. HYSYNTH: context-free LLM approximation for guiding program synthesis, *Advances in Neural Information Processing Systems 37*, Neural Information Processing Systems Foundation, 2024, pp. 15612–15645. https://doi.org/10.52202/079017-0499
11. Sevenhuijsen M., Etemadi K., Nyberg M. VeCoGen: Automating generation of formally verified C code with Large Language Models, *2025 IEEE/ACM 13th International Conference on Formal Methods in Software Engineering (FormaliSE)*, IEEE, 2025, pp. 101–112. https://doi.org/10.1109/FormaliSE66629.2025.00017
12. Ferdowsi K., Huang R., James M. B., Polikarpova N., Lerner S. Validating AI-generated code with live programming, *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, New York: ACM, 2024, article number: 143, pp. 1–8. https://doi.org/10.1145/3613904.3642495
13. Ramírez-Rueda R., Benítez-Guerrero E., Mezura-Godoy C., Bácenas E. A systematic literature review of 10 years of research on program synthesis and Natural Language Processing, *Programming and Computer Software*, 2024, vol. 50, no. 8, pp. 725–741. https://doi.org/10.1134/S0361768824700737

14. Martens C., Simmons R. J., Arntzenius M. Finite-choice logic programming, *Proceedings of the ACM on Programming Languages*, 2025, vol. 9, no. POPL, article number: 13, pp. 362–390. https://doi.org/10.1145/3704849
15. Arhangelsky D. A., Taitslin M. A. A logic for data description, *Logic at Botik '89. Symposium on Logical Foundations of Computer Science, Pereslavl-Zalessky, USSR, July 3–8, 1989, Proceedings*, Berlin–Heidelberg: Springer, 1989, pp. 2–11. https://doi.org/10.1007/3-540-51237-3_2
16. Beltiukov A. P. Automatical synthesis of programs with recursions, *Formal Methods in Programming and Their Applications. International Conference, Academgorodok, Novosibirsk, Russia, June 28 – July 2, 1993. Proceedings*, Berlin–Heidelberg: Springer, 2005, pp. 414–422. https://doi.org/10.1007/BFb0039723
17. Morishita T., Morio G., Yamaguchi A., Sogawa Y. Learning deductive reasoning from synthetic corpus based on formal logic, *arXiv:2308.07336 [cs.AI]*, 2023. https://doi.org/10.48550/arXiv.2308.07336
18. Kalyan A., Mohta A., Polozov O., Batra D., Jain P., Gulwani S. Neural-guided deductive search for real-time program synthesis from examples, *arXiv:1804.01186 [cs.AI]*, 2018. https://doi.org/10.48550/arXiv.1804.01186
19. Bird R., Wadler P. *Introduction to functional programming*, New York: Prentice Hall, 1998. https://archive.org/details/introductiontofu0000bird
20. Meyer A. R., Taitslin M. A. (Eds.). *Logic at Botik'89. Symposium on Logical Foundations of Computer Science, Pereslavl-Zalessky, USSR, July 3–8, 1989, Proceedings*, Berlin–Heidelberg: Springer, 1989. https://doi.org/10.1007/3-540-51237-3
21. Kleene S. C. *Introduction to metamathematics*, Amsterdam: North-Holland, 1952. https://zbmath.org/0047.00703
22. Kolmogoroff A. Zur Deutung der intuitionistischen Logik, *Mathematische Zeitschrift*, 1932, vol. 35, no. 1, pp. 58–65. https://doi.org/10.1007/BF01186549
23. Gödel K. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes, *Dialectica*, 1958, vol. 12, no. 3/4, pp. 280–287. https://doi.org/10.1111/j.1746-8361.1958.tb01464.x
24. Markov A. A. *Theory of algorithms*, Jerusalem: Isreal Program for Scientific Translations, 1954.
25. Warren D. H. An abstract Prolog instruction set, *Technical Report 309*, 1983. SRI International. https://www.sri.com/wp-content/uploads/2021/12/641.pdf
26. Genesereth M., Chaudhri V. K. *Introduction to logic programming*, Cham: Springer, 2020. https://doi.org/10.1007/978-3-031-01586-1
27. Miller D. A survey of the proof-theoretic foundations of logic programming, *Theory and Practice of Logic Programming*, 2021, vol. 22, issue 6, pp. 859–904. https://doi.org/10.1017/S1471068421000533
28. Nepejvoda N. N. A proof theoretical comparison of program synthesis and program verification, *Logic, methodology and philosophy of science: proceedings of the 6th International Congress of Logic, Methodology, and Philosophy of Science. Hannover, 1979*, Amsterdam: North-Holland, 1982.
29. Nepejvoda N. N. The connection between the proof theory and computer programming, *6th International Congress of Logic, Methodology and Philosophy of Science*, Hannover, 1979, vol. 1, pp. 7–11.
30. Tyugu E. H. A programming system with automatic program synthesis, *Methods of Algorithmic Language Implementation*, Berlin–Heidelberg: Springer, 1977, pp. 251–267. https://doi.org/10.1007/3-540-08065-1_16
31. Tyugu E. Using a problem solver in CAD, *Artificial intelligence and pattern recognition in computer aided design. Proceedings of the IFIP working conference organized by working group 5.2, computer aided design, Grenoble, France, March 17–19, 1978*, Amsterdam, New York, Oxford: North-Holland, 1978, pp. 245–263.
32. Joudakizadeh M., Beltiukov A. P. 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. https://doi.org/10.35634/vm240401
33. Huang Kangjing, Qiu Xiaokang, Shen Peiyuan, Wang Yanjun. Reconciling enumerative and deductive program synthesis, *PLDI 2020: Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*, New York: ACM, 2020, pp. 1159–1174. https://doi.org/10.1145/3385412.3386027

34. Feser J. K. *Inductive and deductive synthesis for database applications*, Doctor of Philosophy in Computer Science and Engineering Dissertation, Massachusetts Institute of Technology, 2023. https://hdl.handle.net/1721.1/151281
35. Todorova M., Orozova D. Training difficulties in deductive methods of verification and synthesis of program, *International Journal of Advanced Computer Science and Applications*, 2018, vol. 9, no. 7. https://doi.org/10.14569/IJACSA.2018.090703
36. Ding Yuantian, Qiu Xiaokang. A concurrent approach to string transformation synthesis, *Proceedings of the ACM on Programming Languages*, 2025, vol. 9, no. PLDI, article no. 233, pp. 2131–2155. https://doi.org/10.1145/3729336
37. Nipkow T., Wenzel M., Paulson L. C. (Eds.). *Isabelle/HOL: A proof assistant for higher-order logic*, Berlin–Heidelberg: Springer, 2002. https://doi.org/10.1007/3-540-45949-9
38. Clarke E. M., Grumberg O., Peled D. *Model checking*, London–Cambridge: MIT Press, 1999. https://www.researchgate.net/publication/220689159_Model_Checking
39. Martin-Löf P. Constructive mathematics and computer programming, *Studies in Logic and the Foundations of Mathematics*, 1982, vol. 104, pp. 153–175. https://doi.org/10.1016/S0049-237X(09)70189-2
40. Constable R. L. *Constructive mathematics and automatic program writers*, Ithaca: Cornell University, 1970. https://hdl.handle.net/1813/5943
41. Andriushchenko R., Bartocci E., Češka M., Pontiggia F., Sallinger S. Deductive controller synthesis for probabilistic hyperproperties, *Quantitative Evaluation of Systems: 20th International Conference, QEST 2023, Antwerp, Belgium, September 20–22, 2023, Proceedings*, Cham: Springer, 2023, pp. 288–306. https://doi.org/10.1007/978-3-031-43835-6_20
42. Itzhaky S., Peleg H., Polikarpova N., Rowe R. N. S., Sergey I. Deductive synthesis of programs with pointers: Techniques, challenges, opportunities, *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I*, Cham: Springer, 2021, pp. 110–134. https://doi.org/10.1007/978-3-030-81685-8_5

Received 23.10.2025
Accepted 21.01.2026

Milad Joudakizadeh, PhD Student, Department of Computing Technologies and Intellectual Systems of Big Data, 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 Computing Technologies and Intellectual Systems of Big Data, 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. Deductive program synthesis using logic programming, *Vestnik Udmurtskogo Universiteta. Matematika. Mekhanika. Komp'yuternye Nauki*, 2026, vol. 36, issue 1, pp. 159–178.

КОМПЬЮТЕРНЫЕ НАУКИ 2026. Т. 36. Вып. 1. С. 159–178.

М. Джудакизаде, А. П. Бельтюков

Дедуктивный синтез программ с использованием логического программирования

Ключевые слова: дедуктивный синтез, исчисление секвенций, логическое программирование, Prolog, логический вывод, автоматизация программирования, искусственный интеллект.

УДК 510.649

DOI: 10.35634/vm260108

Эта статья представляет подход к дедуктивному синтезу программ с использованием секвенциального подхода Генцена в рамках логического программирования. Используя секвенциальное исчисление как формальную систему для структурированного логического вывода, наш метод автоматизирует вывод доказуемо корректных программ из спецификаций, выраженных в логике предикатов первого порядка без отрицаний. Мы формализуем синтаксис и семантику секвенциального исчисления, реализуя его основные правила вывода (правила введения и удаления) в виде предикатов в логическом программировании для обеспечения масштабируемого синтеза. Практические примеры демонстрируют преобразование логических спецификаций в исполняемые программы. Подход обеспечивает формальную корректность через конструктивную семантику реализуемости Клини, при этом синтезированные программы работают в субрекурсивном языке, чтобы гарантировать завершение вычислительных процессов. Мы оцениваем сильные стороны метода, включая его надежность для систем с критической безопасностью, и его ограничения, такие как вычислительная сложность для неограниченных конструкций. В сравнении с синтезом, управляемым ИИ, наш подход ставит на первое место формальные гарантии, дополняя современные тенденции. Направления будущих исследований включают оптимизацию вычислительной эффективности и расширение применимости к сложным задачам реального мира.

СПИСОК ЛИТЕРАТУРЫ

1. Gulwani S., Polozov O., Singh R. Program synthesis // Foundations and Trends in Programming Languages. 2017. Vol. 4. Nos. 1–2. P. 1–119. https://doi.org/10.1561/2500000010
2. Manna Z., Waldinger R. A deductive approach to program synthesis // ACM Transactions on Programming Languages and Systems. 1980. Vol. 2. No. 1. P. 90–121. https://doi.org/10.1145/357084.357090
3. Smith D. R., Nedunuri S. Deductive model refinement // NASA Formal Methods. 16th International Symposium, NFM 2024, Moffett Field, CA, USA, June 4–6, 2024, Proceedings. Cham: Springer, 2024. P. 147–165. https://doi.org/10.1007/978-3-031-60698-4_9
4. Gentzen G. Investigations into logical deduction // Studies in Logic and the Foundations of Mathematics. 1969. Vol. 55. P. 68–131. https://doi.org/10.1016/S0049-237X(08)70822-X
5. Colmerauer A., Roussel P. The birth of Prolog // History of programming languages — II. New York: ACM, 1996. P. 331–367. https://doi.org/10.1145/234286.1057820
6. Bratko I. Prolog programming for artificial intelligence. New York: Addison Wesley, 2001. https://archive.org/details/prologprogrammin0000brat_l1m9
7. Constable R. L., Borodin A. B. Subrecursive programming languages, part I: Efficiency and program structure // Journal of the ACM. 1972. Vol. 19. Issue 3. P. 526–568. https://doi.org/10.1145/321707.321721
8. Kowalski R. Predicate logic as programming language // Information Processing, Proceedings of the 6th (IFIP) Congress 1974. Amsterdam: North-Holland, 1974. P. 569–574. https://www.researchgate.net/publication/221330242_Predicate_Logic_as_Programming_Language
9. Hou Xinyi, Zhao Yanjie, Liu Yue, Yang Zhou, Wang Kailong, Li Li, Luo Xiapu, Lo David, Grundy John, Wang Haoyu. Large Language Models for software engineering: A systematic literature review // ACM Transactions on Software Engineering and Methodology. 2024. Vol. 33. No. 8. Article number: 220. P. 1–79. https://doi.org/10.1145/3695988

10. Anaya Gonzalez E., Barke S., Berg-Kirkpatrick T., Kasibatla S. R., Polikarpova N. HYSYNTH: context-free LLM approximation for guiding program synthesis // Advances in Neural Information Processing Systems 37. Neural Information Processing Systems Foundation, 2024. P. 15612–15645. https://doi.org/10.52202/079017-0499
11. Sevenhuijsen M., Etemadi K., Nyberg M. VeCoGen: Automating generation of formally verified C code with Large Language Models // 2025 IEEE/ACM 13th International Conference on Formal Methods in Software Engineering (FormaliSE). IEEE, 2025. P. 101–112. https://doi.org/10.1109/FormaliSE66629.2025.00017
12. Ferdowsi K., Huang R., James M. B., Polikarpova N., Lerner S. Validating AI-generated code with live programming // Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems. New York: ACM, 2024. Article number: 143. P. 1–8. https://doi.org/10.1145/3613904.3642495
13. Ramírez-Rueda R., Benítez-Guerrero E., Mezura-Godoy C., Bácenas E. A systematic literature review of 10 years of research on program synthesis and Natural Language Processing // Programming and Computer Software. 2024. Vol. 50. No. 8. P. 725–741. https://doi.org/10.1134/S0361768824700737
14. Martens C., Simmons R. J., Arntzenius M. Finite-choice logic programming // Proceedings of the ACM on Programming Languages. 2025. Vol. 9. No. POPL. Article number: 13. P. 362–390. https://doi.org/10.1145/3704849
15. Arhangelsky D. A., Taitslin M. A. A logic for data description // Logic at Botik '89. Symposium on Logical Foundations of Computer Science, Pereslavl-Zalessky, USSR, July 3–8, 1989, Proceedings. Berlin–Heidelberg: Springer, 1989. P. 2–11. https://doi.org/10.1007/3-540-51237-3_2
16. Beltiukov A. P. Automatical synthesis of programs with recursions // Formal Methods in Programming and Their Applications. International Conference, Academgorodok, Novosibirsk, Russia, June 28 – July 2, 1993. Proceedings. Berlin–Heidelberg: Springer, 2005. P. 414–422. https://doi.org/10.1007/BFb0039723
17. Morishita T., Morio G., Yamaguchi A., Sogawa Y. Learning deductive reasoning from synthetic corpus based on formal logic // arXiv:2308.07336 [cs.AI]. 2023. https://doi.org/10.48550/arXiv.2308.07336
18. Kalyan A., Mohta A., Polozov O., Batra D., Jain P., Gulwani S. Neural-guided deductive search for real-time program synthesis from examples // arXiv:1804.01186 [cs.AI]. 2018. https://doi.org/10.48550/arXiv.1804.01186
19. Bird R., Wadler P. Introduction to functional programming. New York: Prentice Hall, 1998. https://archive.org/details/introductiontofu0000bird
20. Meyer A. R., Taitslin M. A. (Eds.). Logic at Botik'89. Symposium on Logical Foundations of Computer Science, Pereslavl-Zalessky, USSR, July 3–8, 1989, Proceedings. Berlin–Heidelberg: Springer, 1989. https://doi.org/10.1007/3-540-51237-3
21. Kleene S. C. Introduction to metamathematics. Amsterdam: North-Holland, 1952. https://zbmath.org/0047.00703
22. Kolmogoroff A. Zur Deutung der intuitionistischen Logik // Mathematische Zeitschrift. 1932. Vol. 35. No. 1. P. 58–65. https://doi.org/10.1007/BF01186549
23. Gödel K. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes // Dialectica. 1958. Vol. 12. No. 3/4. P. 280–287. https://doi.org/10.1111/j.1746-8361.1958.tb01464.x
24. Марков А. А. Теория алгорифмов // Труды Математического института имени В. А. Стеклова. 1954. Т. 42. С. 3–375. https://www.mathnet.ru/rus/tm1178
25. Warren D. H. An abstract Prolog instruction set // Technical Report 309, SRI International. 1983. https://www.sri.com/wp-content/uploads/2021/12/641.pdf
26. Genesereth M., Chaudhri V. K. Introduction to logic programming. Cham: Springer, 2020. https://doi.org/10.1007/978-3-031-01586-1
27. Miller D. A survey of the proof-theoretic foundations of logic programming // Theory and Practice of Logic Programming. 2021. Vol. 22. Issue 6. P. 859–904. https://doi.org/10.1017/S1471068421000533
28. Nepejvoda N. N. A proof theoretical comparison of program synthesis and program verification // Logic, methodology and philosophy of science: proceedings of the 6th International Congress of Logic, Methodology, and Philosophy of Science. Hannover, 1979. Amsterdam: North-Holland, 1982.

29. Nepejvoda N. N. The connection between the proof theory and computer programming // Logic, methodology and philosophy of science: proceedings of the 6th International Congress of Logic, Methodology, and Philosophy of Science. Hannover, 1979. Amsterdam: North-Holland, 1982. Vol. 1. P. 7–11.
30. Tyugu E. H. A programming system with automatic program synthesis // Methods of Algorithmic Language Implementation. Berlin–Heidelberg: Springer, 1977. P. 251–267. https://doi.org/10.1007/3-540-08065-1_16
31. Tyugu E. Using a problem solver in CAD // Artificial intelligence and pattern recognition in computer aided design. Proceedings of the IFIP working conference organized by working group 5.2, computer aided design, Grenoble, France, March 17–19, 1978. Amsterdam–New York–Oxford: North-Holland, 1978. P. 245–263.
32. Joudakizadeh M., Beltiukov A. P. Two-level realization of logical formulas for deductive program synthesis // Вестник Удмуртского университета. Математика. Механика. Компьютерные науки. 2024. Т. 34. Вып. 4. С. 469–485. https://doi.org/10.35634/vm240401
33. Huang Kangjing, Qiu Xiaokang, Shen Peiyuan, Wang Yanjun. Reconciling enumerative and deductive program synthesis // PLDI 2020: Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation. New York: ACM, 2020. P. 1159–1174. https://doi.org/10.1145/3385412.3386027
34. Feser J. K. Inductive and deductive synthesis for database applications: dissertation … doctor of philosophy in computer science and engineering / Massachusetts Institute of Technology, 2023. https://hdl.handle.net/1721.1/151281
35. Todorova M., Orozova D. Training difficulties in deductive methods of verification and synthesis of program // International Journal of Advanced Computer Science and Applications. 2018. Vol. 9. No. 7. https://doi.org/10.14569/IJACSA.2018.090703
36. Ding Yuantian, Qiu Xiaokang. A concurrent approach to string transformation synthesis // Proceedings of the ACM on Programming Languages. 2025. Vol. 9. No. PLDI. Article no. 233. P. 2131–2155. https://doi.org/10.1145/3729336
37. Nipkow T., Wenzel M., Paulson L. C. (Eds.). Isabelle/HOL: A proof assistant for higher-order logic. Berlin–Heidelberg: Springer, 2002. https://doi.org/10.1007/3-540-45949-9
38. Clarke E. M., Grumberg O., Peled D. Model checking. London–Cambridge: MIT Press, 1999. https://www.researchgate.net/publication/220689159_Model_Checking
39. Martin-Löf P. Constructive mathematics and computer programming // Studies in Logic and the Foundations of Mathematics. 1982. Vol. 104. P. 153–175. https://doi.org/10.1016/S0049-237X(09)70189-2
40. Constable R. L. Constructive mathematics and automatic program writers. Ithaca: Cornell University, 1970. https://hdl.handle.net/1813/5943
41. Andriushchenko R., Bartocci E., Češka M., Pontiggia F., Sallinger S. Deductive controller synthesis for probabilistic hyperproperties // Quantitative Evaluation of Systems: 20th International Conference, QEST 2023, Antwerp, Belgium, September 20–22, 2023, Proceedings. Cham: Springer, 2023. P. 288–306. https://doi.org/10.1007/978-3-031-43835-6_20
42. Itzhaky S., Peleg H., Polikarpova N., Rowe R. N. S., Sergey I. Deductive synthesis of programs with pointers: Techniques, challenges, opportunities // Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I. Cham: Springer, 2021. P. 110–134. https://doi.org/10.1007/978-3-030-81685-8_5

Поступила в редакцию 23.10.2025
Принята к публикации 21.01.2026

Джудакизаде Милад, аспирант, кафедра вычислительных технологий и интеллектуальных систем больших данных, 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 вычислительных технологий и интеллектуальных систем больших данных, 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. Дедуктивный синтез программ с использованием логического программирования // Вестник Удмуртского университета. Математика. Механика. Компьютерные науки. 2026. Т. 36. Вып. 1. С. 159–178.