Towards Relating Ciao Assertions and LPTP Theorems *

Marco Pérez[1,2] Pedro Lopez-Garcia[3,2] Jose F. Morales[1,2] Manuel V. Hermenegildo[1,2] Fred Mesnard[4]

[1]Universidad Politécnica de Madrid (UPM) Madrid, Spain [2]IMDEA Software Institute, Madrid, Spain

[3]Spanish Council for Scientific Research [4]Université de La Réunion

{marco.perez,pedro.lopez,josef.morales,manuel.hermenegildo}@imdea.org, frederic.mesnard@univ-reunion.fr

Abstract interpretation–based verification is a central component of the Ciao Prolog system, enabling expressive specifications of properties of programs, predicates, and execution states. Independently, the LPTP (Logic Programming Theorem Proving) framework offers a first-order logical formalism for expressing and proving properties of predicates. In this paper, we address a fundamental issue in relating these two frameworks: studying the translation of Ciao assertions into LPTP formulae and identifying a partial correspondence between assertion-based and logic-based specifications. We introduce a systematic translation scheme, characterize assertion classes according to their logical encodability, and propose approximation strategies and auxiliary constructs for non-translatable cases, and finally analyze the resulting soundness and completeness trade-offs. We argue that our proposal enables a tight integration of Ciao 's assertion checking with LPTP-based deductive verification, thereby leveraging their complementary capabilities.

1 Introduction

The landscape of formal verification is defined by a fundamental trade-off between automation and generality. Abstract interpretation offers a powerful framework for fully automated analysis: by reasoning over abstract domains, it allows for "push-button" verification of program properties. However, this automation comes at the cost of precision and expressiveness; properties must be approximated, and complex logical relationships may be lost in the abstraction. Conversely, interactive theorem proving offers maximum generality and precision, capable of reasoning about arbitrarily complex mathematical properties. Yet, this power requires significant manual intervention, making it labor-intensive and difficult to scale to large codebases.

In the context of logic programming, Ciao's assertion framework and the LPTP (Logic Programming Theorem Proving) system serve as clear representatives of these respective paradigms. Abstract interpretation–based verification is a central component of the Ciao system's preprocessor, CiaoPP [1, 3, 6, 4, 2], enabling expressive specifications of properties of programs, predicates, and execution states. Within this framework, assertions serve as machine-understandable specifications that support program validation, debugging, and optimization through static analysis. In turn, the LPTP framework [7, 8, 9] offers a logical formalism aimed at representing logic programs and their properties as first-order formulae suitable for interactive theorem proving, also suitable for partial automation using modern proof systems [5]. While both systems target the verification of logic programs, they do so from complementary perspectives: Ciao assertions are rooted in the approximation of execution states, and can thus cover operational aspects, whereas LPTP is centered on the logical aspects for which it offers proof-theoretic generality. Consequently, it makes compelling sense to combine these distinct paradigms within a single verification platform. A hybrid system can leverage abstract interpretation to automatically discharge simple proof obligations and infer invariants, while reserving the heavy lifting of theorem proving for complex properties that escape static analysis.

*Partially funded by MICIU projects CEX2024-001471-M *María de Maeztu* and TED2021-132464B-I00 *PRODIGY*, and by the European Union GA 101154447 NEAT. We would also like to thank the anonymous reviewers for their very useful and constructive feedback.

In this paper, we address the problem of relating these two frameworks by studying the translation of Ciao assertions into LPTP formulae. We aim to enable a tight integration of Ciao's assertion checking with LPTP-based deductive verification. Our contributions are as follows: Based on some background and preliminary information we develop in Sections 2–6 our formalization. We then identify in Section 7 a partial correspondence between assertion-based and logic-based specifications, showing that while many assertions translate faithfully, others resist direct encoding due to intrinsic semantic mismatches. We then introduce in Sections 8 and 9 a partial translation scheme and characterize assertion classes according to their logical encodability. We finally propose in Section 9 approximation strategies and auxiliary constructs for those assertions that fall outside the direct expressive scope of first-order logic, and we conclude in Section 10.

2 Background

Ciao Assertions. Ciao assertions act as a bridge between the program text and the formal verification machinery of the Ciao Preprocessor (CiaoPP) [1, 3]. Assertions are first-class syntactic entities that can be processed statically, dynamically, or effectively ignored during standard execution. A primary assertion unit is the `pred` declaration, which simultaneously describes call patterns, success states, and computational behavior:

$$\texttt{:-}\ \mathit{Status}\ \ \texttt{pred}\ \ p(X_1,\ldots,X_n) : \mathit{Pre} \Rightarrow \mathit{Post} + \mathit{Comp}.$$

which is syntactic sugar for three distinct assertion types that isolate different operational phases [6, 4]:

- **Calls Assertions (`:- calls`):** The *Pre* field specifies properties that must hold *at the moment the predicate is invoked*. This is inherently operational, constraining the set of admissible execution states (e.g., `ground(X)`, `list(L)`).
- **Success Assertions (`:- success`):** The *Post* field describes the properties that must hold in the success state. It is a logical implication: *if* the precondition holds at the call, *then* the postcondition must hold upon success.
- **Comp Assertions (`:- comp`):** The *Comp* field captures global computational properties of the execution, such as determinism, termination, cost, or non-failure.

Status is a qualifier of the meaning of the assertion:

- `true`: the assertion expresses properties inferred by static analysis (correct over-approximation of the concrete semantic properties).
- `check`: the assertion expresses properties that must hold at run-time, i.e., that the analyzer should prove (or else generate run-time checks for). This is the *default* status, and can be omitted.
- `checked`: the analyzer proved that the property holds in all executions.
- `false`: the analyzer proved that the property does not hold in some execution.
- `trust`: provides the analyzer with information that it should *assume*. This can be useful for, e.g., describing external or unknown code or improving analysis precision.

CiaoPP generally proves or disproves assertions by safely approximating program properties through abstract interpretation-based static analysis (we return to this issue in Section 5).

LPTP Formulae and the Inductive Extension. The LPTP (Logic Programming Theorem Proving) system operates on a different theoretical basis, treating logic programs not as instructions for a resolution engine but as mathematical objects in an *inductive extension* of first-order logic [7, 8]. This framework transforms a Prolog program P into a formal theory $IND(P)$ that includes Clark's completion and specific induction schemas for each predicate. Properties in LPTP are expressed using three primary logical operators **S**, **F**, **T** that model the operational semantics of success, failure and termination respectively. Unlike Ciao assertions, which, as mentioned before, are typically checked using abstract domains (lattices of approximations), LPTP formulae are verified via interactive proof using a Gentzen-style sequent calculus [9]. This

allows LPTP to prove complex properties that require the development of specific domains in abstract interpretation.

3 SLD Semantics

We will denote by $\mathscr{C}$ the universal set of constraints. We let $\overline{\exists}_L\theta$ be the constraint θ restricted to the variables of the syntactic object L. We denote constraint entailment by $\models$, so that $c_1 \models c_2$ denotes that c_1 entails c_2.

An *atom* has the form $p(t_1,...,t_n)$ where p is a predicate symbol and the t_i are terms. A *literal* is either an atom or a constraint. A *goal* is a finite sequence of literals. A *rule* is of the form H `:-` B where H, the *head*, is an atom and B, the *body*. A fact is a rule whose *body* is the literal `true`.

A *constraint logic program*, or *program*, is a finite set of rules. The *definition* of an atom A in program P, $defn_P(A)$, is the set of variable renamings of rules in P such that each renaming has A as a head and has distinct new local (but not head) variables. We assume that every rule in the program has an associated unique identifier. Rule identifiers are denoted by c, usually subscripted.

The operational semantics of a program is in terms of its "derivations" which are sequences of reductions between "states". A *state* $\langle G,\theta,c\rangle$ consists of a goal G, a constraint store (or *store* for short) θ, and a clause identifier c. A state $\langle L :: G,\theta,\_\rangle$, where L is a literal and :: denotes concatenation of sequences, can be *reduced* as follows: if L is a constraint and $\theta \wedge L$ is satisfiable, it is reduced to $\langle G,\theta\wedge L,\varepsilon\rangle$. Note that `true` is considered a constraint (builtin) such that $\theta \wedge$ true $\equiv \theta$; if L is an atom, it is reduced to $\langle B :: G,\theta,c\rangle$ for some rule $(L$ `:-` $B) \in defn_P(L)$ whose identifier is c.

We assume for simplicity that the underlying constraint solver is complete. We use $S \overset{c}{\rightsquigarrow}_P S'$ to indicate that in program P a reduction can be applied to state S using clause c to obtain state S'. Also, $S \overset{C}{\rightsquigarrow}{}^*_P S'$ indicates that there is a sequence of reduction steps from state S to state S' using the sequence of clauses C. When program P is understood from the context, we write $S \overset{c}{\rightsquigarrow} S'$ and $S \overset{C}{\rightsquigarrow}{}^* S'$. When we want to express $S \overset{c}{\rightsquigarrow} S'$ or $S \overset{C}{\rightsquigarrow}{}^* S'$ for some c or C that do not need to specify, we write $S \rightsquigarrow S'$ or $S \rightsquigarrow^* S'$ respectively.

A *derivation* from state S for program P is a finite sequence of states $S_0,S_1,\ldots,S_n$, $n \geq 0$, where S_0 is S, each S_i is of the form $\langle G_i,\theta_i,c_i\rangle$, and for each $1 \leq i \leq n$, $S_{i-1} \overset{c_i}{\rightsquigarrow}_P S_i$. I.e., the derivation represents the sequence of reductions $S \overset{c_1}{\rightsquigarrow}_P S_1 \cdots S_{i-1} \overset{c_i}{\rightsquigarrow}_P S_i \cdots \overset{c_n}{\rightsquigarrow}_P S_n$. Given a non-empty derivation d, its initial and last state are denoted by $init\_state(d)$ and $last\_state(d)$ respectively. The sequence of clauses used in a derivation is denoted by $clseq(d)$. For example, if d is the derivation above, then $clseq(d) = c_1 \cdot c_2 \cdots c_n$. Note that a derivation d can be characterized by $clseq(d)$ and $init\_state(d)$. In addition, $curr\_goal(d)$ and $curr\_store(d)$ denote the first subgoal and the store in $last\_state(d)$, respectively. E.g., if $S_n = \langle g :: G,\theta,c\rangle$, $curr\_goal(d) = g$ and $curr\_store(d) = \theta$. We also denote by $curr\_clause(d)$ the clause used to derive S_n, i.e., c. The set of all derivations from a state S for program P is represented by function $derivations(P,S)$. When P is clear from the context, we use $derivations(S)$ instead for simplicity. A *query* is a pair (L,θ) where L is a goal and θ a store. We extend the $derivations(P,S)$ function to queries, by associating the initial state $\langle L,\theta,\varepsilon\rangle$ to query (L,θ) as follows:

$$derivations(P,I) = \begin{cases} derivations(P,\langle L,\theta,\varepsilon\rangle) & \text{if } I \text{ is a query } (L,\theta) \\ derivations(P,S) & \text{if } I \text{ is a state } S \end{cases} \quad (1)$$

We further extend the *derivations* function to deal with sets of *queries*, denoted by $\mathscr{Q}$, as follows: $derivations(P,\mathscr{Q}) = \bigcup_{Q\in\mathscr{Q}} derivations(P,Q)$. Note that $derivations(P,I)$ can be infinite, but any derivation $d \in derivations(P,I)$ is always a finite sequence of states.

A derivation d is *finished* iff $last\_state(d)$ cannot be reduced. Note also that $derivations(P,I)$ contains not only finished derivations but also all intermediate derivations. A finished derivation d is *successful* iff $last\_state(d)$ is of the form $\langle nil,\theta,c\rangle$, where *nil* denotes the empty sequence. A finished derivation is *failed* iff $last\_state(d)$ is not of the form $\langle nil,\theta',c\rangle$.

Definition 1 (Successor relation between derivations) *Given a derivation $d = S_0, S_1, \ldots, S_n$, $n \geq 0$, we say that derivation d' is a successor of it, denoted by $d \, succ \, d'$, iff d' is of the form $d = S_0, S_1, \ldots, S_n, S_{n+1}$ and there is a clause c_{n+1} such that $S_n \overset{c_{n+1}}{\rightsquigarrow}_P S_{n+1}$. We denote by $successors(d)$ the set of all successors of derivation d, i.e., $successors(d) = \{d' \mid d \, succ \, d'\}$.*

Derivations and Trees. Given state S, we represent $derivations(P,S)$ as an SLD-tree, denoted by $tree(derivations(P,S))$ or simply $tree(P,S)$, whose root is S.

Definition 2 (Branch) *A branch is a (possibly infinite) subset $\mathscr{B} \subseteq derivations(P,S)$ such that all the derivations in $\mathscr{B}$ can be arranged as a sequence $d_0, \ldots, d_{i-1}, d_i, \ldots$ such that for all $i \geq 1$, $d_i \in successors(d_{i-1})$.*

Note that a branch $\mathscr{B}$ can be either *finite* or *infinite*, depending on whether the set $\mathscr{B}$ has a *finite* or *infinite* number of elements, respectively.

Given a *finite* branch $\mathscr{B}$, ordered as $d_0, d_1, \ldots, d_n$ according to the successor relation, we denote its last derivation as $last\_deriv(\mathscr{B}) = d_n$. We sometimes represent $\mathscr{B}$ as just the derivation d_n by abuse of notation, since the rest of derivations $d_0, d_1, \ldots, d_{n-1}$ is redundant and can be obtained from d_n.

Definition 3 (Leaf of a finite branch) *Given a* finite *branch $\mathscr{B}$, we define its leaf as $leaf(\mathscr{B}) = last\_state(last\_deriv(\mathscr{B}))$.*

Definition 4 (Tree of a set of derivations) *We define $tree(P,S) = \{\mathscr{B} \in derivations(P,S) \mid \mathscr{B}$ is a branch$\}$*

In the graphical representation of a tree, the prefixes that are common to several branches are depicted together. The leaf of a branch $\mathscr{B}$ is expanded with a sibling, each one per state $S \in \{last\_state(A) \mid A \in successors(last\_deriv(\mathscr{B}))\}$.

Let $\mathscr{B}$ be a branch. We say that $\mathscr{B}$ is *finished* iff it is *finite* and $last\_deriv(\mathscr{B})$ is *finished*; $\mathscr{B}$ is *successful* iff it is *finished* and $last\_deriv(\mathscr{B})$ is *successful*; $\mathscr{B}$ is *failed* iff it is *finished* and $last\_deriv(\mathscr{B})$ is *failed*.

3.1 Properties of Queries Referred to SLD-Trees

We now express the properties of a query Q in terms of its complete SLD-Tree, i.e., the one resulting from the breadth-first evaluation of Q (also referred to as the resolution tree). We will call these properties "universal."

Given a query Q of the form (L, θ) for program P, if there is a *successful* branch $\mathscr{B} \in tree(P,Q)$ such that $leaf(\mathscr{B})$ is of the form $\langle nil, \theta', c \rangle$, then we say that the constraint $\overline{\exists}_L \theta'$ is an *answer* to Q. We denote by $answers(P,Q)$ the set of all answers to query Q.

A query Q *does not terminate* for P, denoted $does\_not\_terminate(P,Q)$, iff $tree(P,Q)$ has at least one *infinite* branch.

Note that the condition "$tree(P,Q)$ has at least one infinite branch" is equivalent to "$tree(P,Q)$ is infinite or $derivations(P,Q)$ is infinite." It also includes the case where the tree has an infinite number of branches.[1]

A query Q *terminates* for P, denoted by $terminates(P,Q)$, iff $tree(P,Q)$ does not have any infinite branch. It is the negation of *does_not_terminate*:

$$terminates(P,Q) \leftrightarrow \neg does\_not\_terminate(P,Q)$$

Hence, $does\_not\_terminate(P,Q) \leftrightarrow \neg terminates(P,Q)$.

A query Q *succeeds* for P, denoted by $succeeds(P,Q)$, iff $tree(P,Q)$ has one *successful* branch at least. Note that any other branch can be either *successful*, *failed*, or *infinite*.

[1] Note that that if there is an infinite number of finite branches, then necessarily there is at least one infinite branch. For example, if we call `nat(X)` with the standard definition, the branch that always chooses the recursive clause is infinite.

A query Q *fails* for P, denoted by $fails(P,Q)$, iff $tree(P,Q)$ has no *successful* branch. In other words, any branch is either *failed* or *infinite*. It is the negation of *succeeds*:

$$fails(P,Q) \leftrightarrow \neg succeeds(P,Q)$$

Note that both properties above, *succeeds* and *fails*, are orthogonal to the *termination* property. Thus, each of them include both cases, *terminates* and *does not terminate*. We can differentiate these two cases as follows:

A query Q *finitely succeeds* for P, denoted $finitely\_succeeds(P,Q)$, iff $tree(P,Q)$ has at least one *successful* branch, and has no *infinite* branches. Note that any branch that is not *successful* must be *failed*. It can be defined as: $finitely\_succeeds(P,Q) \leftrightarrow succeeds(P,Q) \wedge terminates(P,Q)$.

A query Q *infinitely succeeds* for P, denoted $infinitely\_succeeds(P,Q)$, iff $tree(P,Q)$ has at least one *successful* branch, and at least one *infinite* branch. Note that any other branch can be either *successful*, *failed*, or *infinite*. It can be defined as: $infinitely\_succeeds(P,Q) \leftrightarrow succeeds(P,Q) \wedge \neg terminates(P,Q)$.

A query Q *finitely fails* in P, denoted $finitely\_fails(P,Q)$, iff all branches of $tree(P,Q)$ are *failed* (note that it implies termination). Equivalently, $tree(P,Q)$ is *finite* and has no *successful* branch. It can be defined as follows: $finitely\_fails(P,Q) \leftrightarrow fails(P,Q) \wedge terminates(P,Q) \leftrightarrow \neg succeeds(P,Q) \wedge terminates(P,Q)$.

A query Q *infinitely fails* in P, denoted $infinitely\_fails(P,Q)$, iff $tree(P,Q)$ has at least one *infinite* branch, and has no *successful* branch. I.e., any other branch is either *failed* or *infinite*. Equivalently, $tree(P,Q)$ is *infinite* and has no *successful* branch. It can be defined as follows:
$infinitely\_fails(P,Q) \leftrightarrow fails(P,Q) \wedge \neg terminates(P,Q) \leftrightarrow \neg succeeds(P,Q) \wedge \neg terminates(P,Q)$.

Extending the definitions to sets of queries. We say that property *prop* holds for a set of queries Q and program P, denoted by *prop* (P,Q), iff $\forall Q \in \mathsf{Q} : prop(P,Q)$. For example, *prop* can be a property in the set of properties $\{terminates, does\_not\_terminate, succeeds, fails, finitely\_succeeds, infinitely\_succeeds, finitely\_fails, infinitely\_fails\}$.

3.2 Defining the Properties of Queries w.r.t. a Search Strategy

Each search strategy generates the derivations in a given order, which affects the observable behavior of the evaluation of a query.

Given a query Q and a program P, consider the set $derivations(P,Q)$ as defined in Section 3 for the breadth-first search strategy (complete SLD-tree). Let *str* denote any search strategy, and *df* the depth-first search strategy. We denote by $derivations_{str}(P,Q)$ the set of derivations generated by search strategy *str*. Such set can be defined in the same way as described in Section 3 for $derivations(P,Q)$, but using particular search and selection rules. In general, $derivations_{str}(P,Q) \subseteq derivations(P,Q)$, which can make the observable behavior of the evaluation of a query different from that of breadth-first. The order in which the derivations in $derivations_{str}(P,Q)$ are generated can also change the behavior of the program w.r.t. other properties, such as the order in which the solutions are produced. However we focus here on the properties described in Sections 3.1 and 3.2.

The definitions of such properties w.r.t. a search strategy *str* are the same that the definitions already given w.r.t. the breadth-first strategy, but referred to $derivations_{str}(P,Q)$ instead of $derivations(P,Q)$. I.e., the only difference in the definitions is that we replace $derivations(P,Q)$ with $derivations_{str}(P,Q)$. For example, according to Section 3, $tree(P,S) = tree(derivations(P,S))$, so that we define $tree_{str}(P,S) = tree(derivations_{str}(P,S))$, and do similarly with the rest of definitions, which are exactly the same: *branch*; *leaf*; *finite*, *finished*, *successful* and *failed* branch; properties about queries (*terminates*, *does_not_terminate*, *succeeds*, *fails*, *finitely_succeeds*, *infinitely_succeeds*, *finitely_fails*, *infinitely_fails*), etc.

4 Meaning of the Static Analysis Information

We now explain the meaning of the static analysis results, whose representation as assertions with status `true` was introduced in Section 2. For some abstract domains, CiaoPP's static analysis infers properties that refer to the stores of the derivations (e.g., types/shapes, sharing, freeness, groundness, etc.). We refer to such properties as *state properties*. For other analysis domains, the analysis infers properties that refer to (sub)sets of whole derivations, that we call *computational* or *global* properties (e.g., success, failure, termination, determinism, (type) covering, mutual exclusion, cost, etc.).

Given a set of queries Q for a program P, the analysis results (abstract semantics), denoted by $[\![P]\!]^{\alpha}_{\mathsf{Q}}$, can conceptually be seen as a set of triples: $\{\langle L_1,\lambda_1^c,\lambda_1^s\rangle,\ldots,\langle L_n,\lambda_n^c,\lambda_n^s\rangle\}$. In each $\langle L_i,\lambda_i^c,\lambda_i^s\rangle$ triple, L_i is a predicate descriptor (an atom whose arguments are distinct variables), and λ_i^c and λ_i^s are, respectively, the abstract call and success substitutions, elements of an abstract domain D_α. Each pair (L_i,λ_i^c) defines a calling pattern, which corresponds to a node in the graph constructed by the analysis, and represents a set of concrete queries. There is one tuple per calling pattern. We also consider tuples of the form $\langle L_i,\lambda_i^c,\lambda_i^s,\beta\rangle$, where β represents global computational properties. However, for simplicity, we assume that β is represented in λ_i^s, which includes both success-state and computational properties.

For each predicate p in a program P, the predicate descriptor $p(\bar{v})$ denotes a representative of the class of all normalized atoms for p. Let $Ren(p(\bar{v}),[\![P]\!]^{\alpha}_{\mathsf{Q}})$ denote the set of tuples resulting from the static analysis for predicate $p\in P$ w.r.t. Q renamed to variables $\bar{v}$, i.e.: $Ren(p(\bar{v}),[\![P]\!]^{\alpha}_{\mathsf{Q}}) = \{\langle p(\bar{v}),\lambda^c\sigma,\lambda^s\sigma\rangle \mid \exists\langle p(\bar{v}'),\lambda^c,\lambda^s\rangle \in [\![P]\!]^{\alpha}_{\mathsf{Q}}$ and σ is a renaming substitution s.t. $p(\bar{v}) = p(\bar{v}')\sigma\}$. This renaming is useful, for example, for comparison with information in (check) assertions, among other tasks.

To this end, before presenting correctness results of the static analysis by abstract interpretation, we provide instrumental definitions and notation.

Definition 5 (Calling Context) *Consider a program P, a predicate p and a set of queries Q. The* calling context *of p for P and Q is:*
$C(p,P,\mathsf{Q}) = \{\overline{\exists}_{p(\bar{v})}\theta \mid \exists d\in derivations(P,\mathsf{Q})\ (\exists G\ curr\_state(d) = \langle p(\bar{v})::G,\theta,\_\rangle)\}$.

Definition 6 (Success Context for a Store) *Consider a program P, a predicate p, a constraint store θ, and a set of queries Q. The* success context *of p and θ for P and Q is:*
$S(p,\theta,P,\mathsf{Q}) = \{\overline{\exists}_{p(\bar{v})}\theta' \mid \exists d\in derivations(P,\mathsf{Q})\ \exists G\ (\langle p(\bar{v})::G,\theta,\_\rangle \in d \wedge curr\_state(d) = \langle G,\theta',\_\rangle)\}$.

Definition 7 (Success Context) *Consider a program P, a predicate p, and a set of queries Q. The* success context *of p for P and Q is:*
$S(p,P,\mathsf{Q}) = \bigcup_{\theta\in C(p,P,\mathsf{Q})} S(p,\theta,P,\mathsf{Q})$.

We can restrict the constraints in the calling and success contexts to the variables in $p(\bar{v})$ since this does not affect the evaluation of calls and success assertions.

Definition 8 (Abstract Calling Context) *Consider a program P, a predicate p and a set of queries Q. The* abstract calling context *of p for P and Q is $C^{\alpha}(p,P,\mathsf{Q}) = \{\lambda^c \mid \exists\lambda^s\ (\langle p(\bar{v}),\lambda^c,\lambda^s\rangle \in Ren(p(\bar{v}),[\![P]\!]^{\alpha}_{\mathsf{Q}}))\}$*

Correctness of abstract interpretation guarantees that:

1. $\forall\langle p(\bar{v}),\lambda^c,\lambda^s\rangle \in Ren(p(\bar{v}),[\![P]\!]^{\alpha}_{\mathsf{Q}})\ (\bigcup_{\theta\in\gamma(\lambda^c)} S(p,\theta,P,\mathsf{Q})) \subseteq \gamma(\lambda^s)$.
2. $C(p,P,\mathsf{Q}) \subseteq \gamma(C^{\alpha}(p,P,\mathsf{Q}))$.

In order to ensure correctness of compile-time checking for a set of queries Q, the analyzer must be provided with a suitable Q_α such that $\mathsf{Q}\subseteq\gamma(\mathsf{Q}_\alpha)$. In CiaoPP, Q_α is expressed by means of *entry* assertions.

5 Assertion Checking in CiaoPP

We start by giving some instrumental definitions and notation. First, $subderivation(d^s, d, p)$ expresses that d^s is a subderivation of derivation d for predicate p. More precisely, $\exists d_1, S, d_2$ s.t.:

- $d = d_1 :: S :: d_2$, where $S = \langle p(\bar{v}) :: G, \theta, \_ \rangle$.
- $d^t = \begin{cases} S :: d_3 :: S_2 & \text{if } d_2 = d_3 :: S_2 :: d_4 \wedge S_2 = \langle G, \theta_2, \_ \rangle \text{ and there is no state in } d_3 \text{ of the form } \langle G, \_, \_ \rangle \\ S :: d_2 & \text{otherwise} \end{cases}$
- d^s is the result of replacing any state of the form $\langle G^t :: G, \theta^t, \_ \rangle$ in d^t by $\langle G^t, \theta^s, \_ \rangle$, where θ^s is θ^t projected over the variables of G^t.

If d^s is a subderivation of d for p, then d^s can be obtained by applying the transitions corresponding to the sequence of clauses of d^s to the first state of d^s. Given that d can be represented as a sequence of AND-trees (or an AND-tree), d^s is a sequence of the corresponding AND-subtrees (or an AND-subtree) whose root has predicate p.
$holdscomp(p(\bar{v}), Comp, D)$ expresses that formula *Comp* holds for the derivation set D. r and $r(O)$ represent a variable renaming and the result of applying it to some syntactic object O, respectively.

Definition 9 (Checked Comp) *A comp assertion* `:- comp` $p(\bar{v}) : Pre$ `+` $Comp$ *is* checked *for a predicate* $p \in P$ *and query set* Q *iff* $holdscomp(p(\bar{v}), Comp, D)$, *where* $D = \{d \in subderivations(P, \mathsf{Q}, p) \mid$ *the first state of* d *meets* Pre, *i.e.,* $d = S :: d_1 \wedge S = \langle q :: G, \theta, \_ \rangle \wedge q = r(p(\bar{v})) \wedge \theta \models_M r(Pre)\}$

Definition 10 (Checked Succ) *A success assertion* `:- success` $p(\bar{v}) : Pre$ `=>` $Post$ *is* checked *for a predicate* $p \in P$ *and query set* Q *iff for all successful* $d \in subderivations(P, \mathsf{Q}, p)$, *if the first state of* d *meets* Pre, *then the last state of* d *meets* $Post$.

True Assertions. For success and comp assertions, we also define the concept of *true* assertion (in addition to *checked* and *false*). We say that an assertion is *true* iff it is *checked* for every set of queries Q. Note that an assertion condition $calls(p, Pre)$ can never be found to be *true*, as the calling context of p depends on the query. If we pose no restriction on the queries we can always find a calling state which violates the assertion, unless Pre is a tautology. Clearly, the concept of *true* assertions is strictly stronger than that of *checked* assertions. In general, an assertion that is *checked* (for a given set of queries) is not necessarily *true*.
Context-Independence. Given a success assertion `:- success` $p(\bar{v}) : Pre$ `=>` $Post$ or a comp assertion `:- comp` $p(\bar{v}) : Pre$ `+` $Comp$, the *context-independent* set of queries of it is: $Q((p(\bar{v}), Pre)) = \{(p(\bar{v}), \theta) \mid \theta \models_M Pre\}$. I.e., the set of all queries that meet precondition Pre.

We say that a success assertion (resp. comp assertion) is *context-independent checked* for predicate $p \in P$ iff it is *checked* for p and its corresponding *context-independent* set of queries. The concepts of *context-independent checked* and *true* assertions are equivalent.

6 LPTP Semantics

To compare LPTP semantics with the general Prolog semantics described above, it is important to take into account that LPTP distinguishes two separate sets of notions of *success*, *failure*, and *termination*. The first set is semantic in nature and is defined in terms of derivations, leading to concepts equivalent to those examined previously in Section 3. The names used to refer to these properties in LPTP papers and documents coincide with those used in Section 3, with the exception of the name *fails*, which corresponds to the property called *finitely fails* (or *finitely_fails*) in Section 3. To avoid confusion and to adopt a unified and as standard as possible nomenclature, in this paper we use the name *finitely fails* instead of the LPTP name *fails* when referring to this property, including in the context of LPTP. We reserve the name *fails* exclusively for the

BF (Universal) Property	Relation	LPTP Inference
$terminates(P,Q)$	$\Leftrightarrow$	$\mathrm{IND}(P) \vdash \mathbf{T}\,Q$
$does\_not\_terminate(P,Q)$	$\Leftrightarrow$	$\mathrm{IND}(P) \vdash \neg\mathbf{T}\,Q$
$succeeds(P,Q)$	$\Leftrightarrow$	$\mathrm{IND}(P) \vdash \mathbf{S}\,Q$
$fails(P,Q)$	$\Leftrightarrow$	$\mathrm{IND}(P) \vdash \mathbf{F}\,Q$
$finitely\_succeeds(P,Q)$	$\Leftrightarrow$	$\mathrm{IND}(P) \vdash \mathbf{T}\,Q \wedge \mathbf{S}\,Q$
$infinitely\_succeeds(P,Q)$	$\Leftrightarrow$	$\mathrm{IND}(P) \vdash \neg\mathbf{T}\,Q \wedge \mathbf{S}\,Q$
$finitely\_fails(P,Q)$	$\Leftrightarrow$	$\mathrm{IND}(P) \vdash \mathbf{T}\,Q \wedge \mathbf{F}\,Q$
$infinitely\_fails(P,Q)$	$\Leftrightarrow$	$\mathrm{IND}(P) \vdash \neg\mathbf{T}\,Q \wedge \mathbf{F}\,Q$
$not\_finitely\_fails(P,Q)$	$\Leftrightarrow$	$\mathrm{IND}(P) \vdash \mathbf{S}\,Q \vee \neg\mathbf{T}\,Q$

Table 1: Relating universal properties (breadth-first) with properties inferred by LPTP.

property defined in Section 3 as the negation of *succeeds*, which also includes the case of *non–termination*. Consequently, the names *terminates* and *succeeds* in LPTP documents refer to the same properties described under those names in Section 3.

The second set of notions is the most important in practice, as it is the one handled directly by the LPTP system. In the system, we have access to three syntactic operators acting on program goals, transforming them into formulas in the formal language of LPTP. Formally, we write **S**, **F**, **T** for the succeed, failure and termination operators, respectively. The complete inductive definition can be found in [7] along with the following theorems that relate the two notions:

Theorem 6.1 (Soundness)

1. *If G terminates, then* $IND(P) \vdash \mathbf{T}\,G$.
2. *If G succeeds with answer* σ, *then* $IND(P) \vdash \mathbf{S}\,G\sigma$.
3. *If G finitely fails, then* $IND(P) \vdash \mathbf{F}\,G$.

Theorem 6.2 (Adequacy)

1. *If* $IND(P) \vdash \mathbf{T}\,G$, *then G terminates.*
2. *If* $IND(P) \vdash \mathbf{T}\,G \wedge \mathbf{S}\,G\sigma$, *then G succeeds with answer including* σ.
3. *If* $IND(P) \vdash \mathbf{T}\,G \wedge \mathbf{F}\,G$, *then G finitely fails.*

Note that for a terminating goal we get equivalence between the two notions of success and failure but not in general.

7 SLD Semantics and LPTP

In this section we discuss the relation between the properties declaratively defined in Section 3 and the LPTP operators. We first relate the properties that have been defined with respect to a breadth-first search strategy that generates the complete SLD-tree (Section 3.1), i.e., universal properties. Then, we carry out the same analysis for properties defined with respect to Prolog 's depth-first search strategy (Section 3.2).

Both kinds of properties coincide for terminating goals.

However, for non-terminating goals (i.e., goals for which there exist infinite branches), differences arise. These appear when the definition of the property in question may involve both *infinite* and *successful* branches of the search trees. In such cases, the tree reachable under depth-first search before looping may contain no successful branch, whereas the breadth-first tree may contain successful branches. The resulting relationships are summarized in Tables 1, 2 and 3.

Table 3 expresses relations between properties inferred by LPTP and properties inferred by CiaoPP. They allow the bidirectional interaction between the two complementary systems. For example, if a property is

DF (Prolog) Property	Relation	BF (Universal) Property
$terminates_{df}(P,Q)$	$\Leftrightarrow$	$terminates(P,Q)$
$does\_not\_terminate_{df}(P,Q)$	$\Leftrightarrow$	$does\_not\_terminate(P,Q)$
$succeeds_{df}(P,Q)$	$\Rightarrow$	$succeeds(P,Q)$
$fails_{df}(P,Q)$	$\Leftarrow$	$fails(P,Q)$
$finitely\_succeeds_{df}(P,Q)$	$\Leftrightarrow$	$finitely\_succeeds(P,Q)$
$infinitely\_succeeds_{df}(P,Q)$	$\Rightarrow$	$infinitely\_succeeds(P,Q)$
$finitely\_fails_{df}(P,Q)$	$\Leftrightarrow$	$finitely\_fails(P,Q)$
$infinitely\_fails_{df}(P,Q)$	$\Leftarrow$	$infinitely\_fails(P,Q)$
$not\_finitely\_fails_{df}(P,Q)$	$\Leftrightarrow$	$not\_finitely\_fails(P,Q)$

Table 2: Relating DF (Prolog) Properties with BF (Universal) Properties.

LPTP Inference	Relation	CiaoPP Inference
$\text{IND}(P) \vdash \mathbf{T}G$	$\Leftrightarrow$	$terminates_{ciaopp}(P,Q)$
$\text{IND}(P) \vdash \neg\mathbf{T}G$	$\Leftrightarrow$	$does\_not\_terminate_{ciaopp}(P,Q)$
$\text{IND}(P) \vdash \mathbf{S}G$	$\Leftrightarrow$	$succeeds_{ciaopp}(P,Q)$
	$\Leftarrow$	$not\_finitely\_fails_{ciaopp}(P,Q) \wedge terminates_{ciaopp}(P,Q)$
$\text{IND}(P) \vdash \mathbf{F}G$	$\Leftrightarrow$	$fails_{ciaopp}(P,Q)$
$\text{IND}(P) \vdash \mathbf{T}G \wedge \mathbf{S}G$	$\Leftrightarrow$	$finitely\_succeeds_{ciaopp}(P,Q)$
	$\Leftrightarrow$	$not\_finitely\_fails_{ciaopp}(P,Q) \wedge terminates_{ciaopp}(P,Q)$
$\text{IND}(P) \vdash \neg\mathbf{T}G \wedge \mathbf{S}G$	$\Leftrightarrow$	$infinitely\_succeeds_{ciaopp}(P,Q)$
$\text{IND}(P) \vdash \mathbf{T}G \wedge \mathbf{F}G$	$\Leftrightarrow$	$finitely\_fails_{ciaopp}(P,Q)$
$\text{IND}(P) \vdash \neg\mathbf{T}G \wedge \mathbf{F}G$	$\Leftrightarrow$	$infinitely\_fails_{ciaopp}(P,Q)$
$\text{IND}(P) \vdash \mathbf{S}G \vee \neg\mathbf{T}G$	$\Leftrightarrow$	$not\_finitely\_fails_{ciaopp}(P,Q)$

Table 3: Relating properties inferred by LPTP and properties inferred by CiaoPP.

inferred by CiaoPP, using such relations, we could assert a logical representation of such property as an axiom for LPTP or as a proof with an special tag "by CiaoPP". Conversely, properties inferred by LPTP could be expressed in some cases as Ciao assertions with status trust.

This concludes the formalization of the semantic framework. Building on these correspondences, we now address the practical translation of Ciao assertions into LPTP theorems.

8 From Ciao Assertions to Logical Formulae

First, we propose a suitable representation of Ciao assertions as first-order logical formulas. We focus for now on Ciao assertion schemas of the form:

```
1 :- pred p(X1, ..., Xn) : Pre => Post + Comp.
```

which, as described in Section 2, are internally translated into an equivalent set of three assertions:

```
1 :- calls p(X1, ..., Xn) : Pre.
2 :- success p(X1, ..., Xn) : Pre => Post.
3 :- comp p(X1, ..., Xn) : Pre + Comp.
```

The calls assertion presents problems due to its essentially operational nature but both success and comp assertions can be translated to first order formulas which are easily interpreted in the LPTP language. The properties in the Pre and Post fields are built from conjunctions of predicates over the variables of p/n,

Ciao property	**Corresponding LPTP formula**
Type atom $q(V)$ in **Pre** or **Post**	$\mathbf{S}\ \mathsf{q}(V)$
`ground(V)`	$\mathsf{gr}(V)$
`terminates`	$\mathbf{T}\ \mathsf{p}(X)$
`not_fails`	$\exists X_2 :\ \neg\mathbf{T}\ \mathsf{p}(X_1,X_2)\ \vee\ \mathbf{S}\ \mathsf{p}(X_1,X_2)$
`num_solutions(X,N)`	$\exists_{=N} x :\ \mathsf{p}(x)$
`det(X)`	$\exists_{=N}(x,t) :\ \mathsf{p}^t(x,t)$, using an explicit trace argument; equivalently, one successful trace or divergence

Table 4: Correspondence between Ciao properties and LPTP formulas.

or from *star terms*, a shorthand notation that ultimately reduces to such conjunctions. Because of this, they can be expressed directly in the language of P. The situation is different for `Comp`. Its properties describe the execution behavior of a program, which falls outside what the base language of P can express. Nevertheless, many of these properties can be captured in the extended language $\mathrm{IND}(P)$.

We will later make explicit this representation, for now we assume that we are working in an appropriate extension of the language in which we can express a formula Comp representing this property.

Finally, we can represent the `success` assertion via the formula:

$$\forall \bar{X} : \mathsf{Pre}(\bar{X}) \rightarrow (\mathsf{succeeds}(p(\bar{X})) \rightarrow \mathsf{Post}(\bar{X})) \tag{2}$$

where $\bar{X} = (X_1, \ldots, X_n)$ is an n-tuple of variables and succeeds is the LPTP operator. And the `comp` assertion via:

$$\forall \bar{X} : \mathsf{Pre}(\bar{X}) \rightarrow \mathsf{Comp}(p(\bar{X})). \tag{3}$$

Notice that both expressions are of the form:

$$\forall \bar{X} : \mathsf{Pre}(\bar{X}) \rightarrow \Psi(\bar{X}). \tag{4}$$

where Ψ is a predicate in the adequate language. Moreover, we can put both formulas together into a single formula for a direct translation of the original `pred` assertion:

$$\forall \bar{X} : \mathsf{Pre}(\bar{X}) \rightarrow ((\mathsf{succeeds}(p(\bar{X})) \rightarrow \mathsf{Post}(\bar{X})) \wedge \mathsf{Comp}(p(\bar{X}))). \tag{5}$$

We now apply the general translation scheme (Eq. 5) to concrete Ciao properties. The sections below demonstrate how both simple properties (types) and complex computational properties (termination, non-failure) fit within our framework, revealing which assertions are naturally encodable and which require approximation.

9 Property Correspondence

As previously explained, both Ciao and LPTP assertions are formed from a collection of atomic predicate symbols that state properties about the program's predicates. In LPTP these atomic properties are three: succeeds, fails and terminates. In Ciao we have more complex atomic properties such as determinacy or not finitely failing but a lot of them can be represented in LPTP using its three atomic properties and its complete assertion language. In this section we provide a correspondence between Ciao properties and LPTP assertions. Table 4 summarizes the correspondence between the Ciao properties used in assertions and their representation in LPTP. Throughout this section we restrict our attention to *instantiation types*.

Type and Mode Properties. The first two rows of Table 4 are direct translations. A type property appearing in the **Pre** or **Post** part of a Ciao assertion is represented in LPTP by a succeeds atom for the corresponding predicate, while `ground/1` is translated as $\mathsf{gr}/1$. For example, consider the following program:

```
nat(0).
nat(s(X)) :- nat(X).

plus(0,Y,Y).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).

times(0,_Y,0).
times(s(X),Y,Z) :- times(X,Y,A), plus(Y,A,Z).
```

The following Ciao assertion:

```
:- success plus(X, Y, Z) : (nat(X), nat(Y)) => nat(Z).
```

expresses that if any call to plus(X, Y, Z) with X and Y bound to natural numbers (nat) succeeds, then Z must be bound to a nat upon success. It corresponds, by the type row of Table 4, to: $\forall(X,Y,Z).\ (\mathbf{S}\ \mathsf{plus}(X,Y,Z) \wedge \mathbf{S}\ \mathsf{nat}(X) \wedge \mathbf{S}\ \mathsf{nat}(Y)) \rightarrow \mathbf{S}\ \mathsf{nat}(Z)$.
In LPTP this can be stated as:

```
:- lemma(plus:type,
all [x,y,z]: succeeds plus(?x,?y,?z) & succeeds nat(?x) & succeeds nat(?y) =>
 succeeds nat(?z), [...]).
```

This assertion can be proven automatically by CiaoPP provided we add an extra assertion:

```
:- entry plus(X, Y, Z) : (nat(X), nat(Y)).
```

This explicitly tells the preprocessor about the way this predicate is going to be called, which is necessary due to the top-down and goal-driven nature of the CiaoPP system.

The same direct translation applies to mode properties. For instance, the following Ciao assertion:

```
:- success plus(X, Y, Z) : (ground(X), ground(Y)) => ground(Z).
```

corresponds, by the groundness row of Table 4, to:

```
:- lemma(plus:ground,
all [x,y,z]: succeeds plus(?x,?y,?z) & gr(?x) & gr(?y) => gr(?z), [...]).
```

It is also worth noting why the succeeds atom appears in the antecedent of the translation. If we replace plus by the failing predicate:

```
plusB(0,Y,Y) :- fail.
plusB(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

then the translated implication may still hold vacuously, since plusB never succeeds. Indeed, the following lemma is provable in LPTP:

```
:- lemma(plusB:fail, all [x,y,z]: succeeds nat(?x) => fails plusB(?x,?y,?z), [...]).
```

Computational Properties. The remaining rows of Table 4 concern computational properties. For termination, the translation is immediate: a Ciao assertion of the form:

```
:- comp plus(X, Y, Z) : (nat(X), nat(Y)) + terminates.
```

becomes:

```
:- lemma(plus:termination:1,
all [x,y,z]: succeeds nat(?x) => terminates plus(?x,?y,?z), [...]).
```

as indicated by the terminates row. For not_fails, the corresponding row in Table 4 expresses that a call either succeeds for some output or does not terminate. Writing $X = (X_1, X_2)$, with X_1 the input variables and X_2 the output variables, the following Ciao assertion:

```
:- comp plus(X, Y, Z) : (nat(X), nat(Y)) + not_fails.
```

is translated as:

```
:- lemma(plus:not_fails,
all [x,y]: succeeds nat(?x) =>
  (ex z:  terminates plus(?x,?y,?z) \/ succeeds plus(?x,?y,?z)), [...]).
```

For `num_solutions(X,N)`, Table 4 uses the counting quantifier $\exists_{=N}$. Since LPTP is first-order, this should be understood as syntactic sugar for a first-order formula expressing that there are exactly N distinct solutions. In the special case $N = 1$, this yields the usual uniqueness statement; for example, uniqueness of Peano addition can be written as:

```
:- lemma(plus:uniqueness,
all [x,y,z1,z2]:
  succeeds plus(?x,?y,?z1) & succeeds plus(?x,?y,?z2) => ?z1 = ?z2, [...]).
```

Finally, the last row of Table 4 explains `det(X)`. Since Ciao determinacy distinguishes solutions not only by answer substitutions but also by derivation trace, the translation is not stated directly on the original predicate; instead, we use a traced version $\mathsf{p}^t(X,t)$ with an additional trace argument and then require exactly one successful trace. For example, consider:

```
edge(a, b).    edge(b, d).    edge(a, c).    edge(c, d).

path(X, Y) :- edge(X, Y).
path(X, Y) :- edge(X, Z), path(Z, Y).
```

The query `path(a,d)` yields the same output binding along two different derivations, so to capture determinacy we introduce traces explicitly:

```
path_trace(X,Y, c1(T)) :- edge(X,Y, T).
path_trace(X,Y, c2(T1,T2)) :- edge(X,Z,T1), path_trace(Z,Y, T2).
```

Determinacy is then expressed applying the `num_solutions` translation with $N = 1$ to the traced predicate.

Finally, using the property correspondence between Ciao and LPTP above, and the translation schemas given in Section 8, we conclude that if we can find a proof for the proposed translation of a Ciao assertion into LPTP, then such assertion is *true* (w.r.t. definitions given in Section 5). Conversely, *true* assertions inferred by CiaoPP could be stated as theorems in LPTP.

10 Conclusions

We have addressed the problem of relating Abstract Interpretation–based verification and Theorem Proving by studying the translation of Ciao assertions into LPTP formulae and identifying a partial correspondence between assertion-based and logic-based specifications, showing that while many assertions translate faithfully, others resist direct encoding due to intrinsic semantic mismatches. We have introduced a partial translation scheme, characterized assertion classes according to their logical encodability, and proposed approximation strategies and auxiliary constructs for non-translatable cases. We have also analyzed the resulting soundness and completeness trade-offs. We argue that the proposed techniques enables a tight integration of Ciao 's assertion checking with LPTP-based deductive verification, leveraging their complementary capabilities.

References

[1] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M.V. Hermenegildo, J. Maluszynski & G. Puebla (1997): *On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs*. In: *Proc. of the 3rd Int'l. Workshop on Automated Debugging–AADEBUG'97*, U. of Linköping Press, Linköping, Sweden, pp. 155–170. Available at `https://cliplab.org/papers/aadebug97-informal_bitmap.pdf`.

[2] I. Garcia-Contreras, J. F. Morales & M.V. Hermenegildo (2021): *Incremental and Modular Context-sensitive Analysis*. *Theory and Practice of Logic Programming* 21(2), pp. 196–243, doi:10.1017/S1471068420000496. Available at `https://arxiv.org/abs/1804.01839`.

[3] M.V. Hermenegildo, G. Puebla & F. Bueno (1999): *Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging*. In K. R. Apt, V. Marek, M. Truszczynski & D. S. Warren, editors: *The Logic Programming Paradigm: a 25–Year Perspective*, Springer-Verlag, pp. 161–192.

[4] M.V. Hermenegildo, G. Puebla, F. Bueno & P. Lopez-Garcia (2005): *Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor)*. *Science of Computer Programming* 58(1–2), pp. 115–140, doi:10.1016/j.scico.2005.02.006.

[5] Fred Mesnard, Thierry Marianne & Étienne Payet (2025): *Automated Theorem Proving for Prolog Verification*. In Martin Gebser, Daniela Inclezan, Francesco Ricca, Manuel Carro & Miroslaw Truszczynski, editors: *Proceedings 41st International Conference on Logic Programming, ICLP 2025, Rende, Italy, 12-19th September 2025*, EPTCS, pp. 469–481, doi:10.4204/EPTCS.439.32. Available at `https://doi.org/10.4204/EPTCS.439.32`.

[6] G. Puebla, F. Bueno & M.V. Hermenegildo (2000): *Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs*. In: *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, *LNCS* 1817, Springer-Verlag, pp. 273–292, doi:10.1007/10720327_16.

[7] Robert F. Stärk (1997): *Formal Verification of Logic Programs: Foundations and Implementation*. In: *Logical Foundations of Computer Science*, 1234, pp. 354–368, doi:10.1007/3-540-63045-7_36. Available at `http://link.springer.com/10.1007/3-540-63045-7_36`.

[8] Robert F. Stärk (1998): *The Theoretical Foundations of LPTP (A Logic Program Theorem Prover)*. *J. Log. Program.* 36(3), pp. 241–269, doi:10.1016/S0743-1066(97)10013-9. Available at `https://doi.org/10.1016/S0743-1066(97)10013-9`.

[9] Robert F. Stärk (2001): *LPTP: A Logic Program Theorem Prover*.