Operational Semantics of Resolution in Horn Clause Logic

Peng Fu, Ekaterina Komendantskaya

Abstract. This paper presents a study of operational and type-theoretic properties of different resolution strategies in Horn clause logic. We distinguish four different kinds of resolution: resolution by unification (SLD-resolution), resolution by term-matching, the recently introduced structural resolution, and partial (or lazy) resolution. We express them all uniformly as abstract reduction systems, which allows us to undertake a thorough comparative analysis of their properties. To match this small-step semantics, we propose to take Howard’s system $H$ as a type-theoretic semantic counterpart. Using system $H$, we interpret Horn formulas as types, and a derivation for a given formula as the proof term inhabiting the type given by the formula. We prove soundness of these abstract reduction systems relative to system $H$, and we show completeness of SLD-resolution and structural resolution relative to the system $H$. We identify conditions under which structural resolution is operationally equivalent to SLD-resolution. We show a one-to-one correspondence between term rewriting and term-matching resolution for Horn clause programs without existential variables.

Keywords: Logic Programming, Typed lambda calculus, Reduction Systems, Structural Resolution, Termination, Productivity.

1. Introduction

Horn clause logic is a fragment of first-order logic that gives theoretical foundations to logic programming (LP). A set of Horn clauses is called a logic program. SLD-resolution is the most common LP algorithm for automatically inferring whether, given a logic program $\Phi$ and a first-order formula $A$, $\Phi \vdash \sigma A$ holds for some substitution $\sigma$. SLD-resolution is semi-decidable as not all derivations by SLD-resolution terminate. The meaning of terminating SLD-resolution is quite well understood [18], and SLD-resolution has been successfully incorporated into a number of LP implementations. However, nontermination has made it difficult for such systems to implement computations on infinite data structures.

Example 1. Consider the following logic program.

$$\kappa_1 : \text{Stream}(y) \Rightarrow \text{Stream}(\text{Cons}(0,y))$$

For query $\text{Stream}(x)$, it renders the following SLD-derivation:

$$\Phi \vdash \{\text{Stream}(x)\} \leadsto_{\kappa_1 \cdot \{\text{Cons}(0,y_1)/x\}} \{\text{Stream}(y_1)\} \leadsto_{\kappa_1 \cdot \{\text{Cons}(0,y_2)/y_1, \text{Cons}(0,\text{Cons}(0,y_2))/x\}} \{\text{Stream}(y_2)\} \leadsto_{\kappa_1 \cdot \{\text{Cons}(0,y_3)/y_2, \text{Cons}(0,\text{Cons}(0,y_3))/y_1, \text{Cons}(0,\text{Cons}(0,\text{Cons}(0,y_3)))/x\}} \{\text{Stream}(y_3)\} \leadsto \ldots$$

In the above derivation, we record the clauses that are used to make a resolution step and the computed substitutions as subscripts. For this derivation, it is impossible to find a finite substitution $\sigma$ such that $\Phi \vdash \sigma(\text{Stream}(x))$. Nevertheless, the query $\text{Stream}(x)$ is computationally meaningful, since it computes an infinite stream of zeros for $x$. 
The importance of developing infrastructures for computing with infinite data structures in LP has been argued in [6, 18, 22]. In the classical approach [18], the semantic view was taken: if a nonterminating SLD-resolution derivation for \( \Phi \) and \( A \) accumulates computed substitutions \( \sigma_0, \sigma_2, \ldots \) in such a way that \( \ldots (\sigma_2(\sigma_0(A))) \) is an infinite ground formula, then \( \ldots (\sigma_2(\sigma_0(A))) \) is said to be *computable at infinity*. Computations at infinity were proven to be sound with respect to the greatest Herbrand model construction for logic programs, i.e. if a formula is computable at infinity, it is also in the greatest Herbrand model of the program. Importantly for us, the notion of *infinite formula computed at infinity* captures the modern-day notion of producing an infinite data structure. We will use the term *global productivity* to describe computations at infinity. For example, the derivation shown in Example 1 is globally productive, as it computes an infinite stream of zeros at infinity. However, this approach did not result in implementation as no procedure for (semi-)deciding this property was given.

An alternative solution was proposed in [6, 22]: subgoals produced in the course of an infinite SLD-derivation are memorized, and if any two subgoals are unifiable, then the derivation is said to be closed coinductively. This approach was implemented as CoLP extension to Prolog, but it suffers from two main problems. Firstly, it does not work well for computations that produce infinite formulas with irrational tree structure, as in this case the derivations do not feature unifiable subgoals. Secondly, CoLP is neither sound nor complete relative to computations at infinity; and thus is not suitable for studies of productivity of infinite data computed by resolution. For example, the query \( P(x) \) for the clause \( P(y) \Rightarrow P(y) \) will exhibit a cycle and will be coinductively proven by CoLP, but it will not compute an infinite term at infinity. In other words, the derivation for \( P(x) \) is not globally productive despite being coinductively provable by CoLP.

In this paper, we distinguish further two variants of the classical SLD-resolution, both have distinct operational properties and play a role in productivity studies. If we replace the unification in SLD-resolution to term-matching, we obtain term-matching resolution. This kind of resolution is used in e.g. type class resolution [12] in functional programming. It has different operational properties compared to SLD-resolution. For example, taking the program in Example 1 and the query \( \text{Stream}(x) \), term-matching resolution will not loop but will terminate after one step, as it will not be able to match \( \text{Stream}(\text{Cons}(0, y)) \) to \( \text{Stream}(x) \).

Another version of resolution combines terminating term-matching steps with potentially nonterminating unification steps. It is called *structural resolution* in [10]. For example, consider the following derivation (where \( \rightarrow \) denotes a term-matching step, and \( \rightarrow^* \) applies substitution obtained by unification to the current queries):

\[
\begin{align*}
\Phi \vdash \{\text{Stream}(x)\} & \rightarrow_{\kappa_1} \{\text{Stream}(\text{Cons}(0, y_1))/x\} \{\text{Stream}(\text{Cons}(0, y_1))\} \rightarrow_{\kappa_1} \\
\{\text{Stream}(y_1)\} & \rightarrow_{\kappa_1} \{\text{Cons}(0, y_2)/y_1, \text{Cons}(0, \text{Cons}(0, y_2))/x\} \{\text{Stream}(\text{Cons}(0, y_2))\} \rightarrow_{\kappa_1} \\
\{\text{Stream}(y_2)\} & \rightarrow_{\kappa_1} \{\text{Cons}(0, y_3)/y_2, \text{Cons}(0, \text{Cons}(0, y_3))/y_1, \text{Cons}(0, \text{Cons}(0, \text{Cons}(0, y_3)))/x\} \{\text{Stream}(\text{Cons}(0, y_3))\} \rightarrow_{\kappa_1} \\
\{\text{Stream}(y_3)\} & \rightarrow^* \ldots
\end{align*}
\]

Note that the overall derivation is nonterminating, but the term-matching derivations are always finite in the above resolution trace. This separation of term-matching and unification enabled us to formulate an alternative notion of *observational productivity*: given a program \( \Phi \) and a query \( A \), if a derivation for \( A \) is infinite, but features only terminating term-matching resolution steps, it is called *observationally productive*. Under certain conditions, observational productivity implies global productivity. See [14] for further discussion of the relation between observational productivity and global productivity.

In this paper, we introduce another notion of productivity for LP. When SLD-resolution produces a finite or infinite ground answer for a variable in the query, we say the query is *locally productive* at that variable (see Definition 24). In order to formally define this notion of productivity, we introduce a lazy version of resolution (called *partial resolution*). Firstly, we label those variables in queries for which we want to compute substitutions. Partial resolution then takes the labels into account when performing the resolution, by giving priority to subgoals with labelled variables. The resolution will stop when all the labels in the queries are eliminated, or in other words, when all required substitutions have been computed.

To illustrate these notions of productivity, we consider the following three logic programs in the table below.
In this paper, we establish a framework for a comparative analysis of different kinds of resolution and different notions of productivity. Firstly, we use a uniform style of small-step semantics for all these kinds of resolution and formulate them as abstract reduction systems. We call the resulting abstract reduction systems \( \text{LP-Unif} \), \( \text{LP-TM} \), \( \text{LP-Struct} \), and \( \text{partial LP-Unif} \), respectively. Using this framework, we ask and answer several research questions about operational properties and relations of the these reduction systems. Are \( \text{LP-Unif} \) and \( \text{LP-Struct} \) equivalent for terminating derivations, and under which conditions? Are \( \text{LP-Unif} \) and \( \text{LP-Struct} \) equivalent for observationally productive programs? Since termination of \( \text{LP-TM} \) is essential for observational productivity, are there any suitable program transformation methods to ensure \( \text{LP-TM} \) termination?

We give a type-theoretic semantics to interpret all these reduction systems. Notably, we take system \( \mathbf{H} \) (based on Howard’s work [8]) as a calculus to capture the type-theoretic meaning of \( \Phi \). We show that \( \text{LP-Unif} \), \( \text{LP-TM} \), \( \text{LP-Struct} \) and partial \( \text{LP-Unif} \) are sound relative to the system \( \mathbf{H} \). Moreover, \( \text{LP-Unif} \) is complete relative to system \( \mathbf{H} \), and under a meaning preserving transformation, \( \text{LP-Struct} \) is also complete relative to system \( \mathbf{H} \).

We discover that, given a program \( \Phi \) and a formula \( P \), \( \text{LP-Struct} \) is operationally equivalent to \( \text{LP-Unif} \) under two conditions; the termination of all \( \text{LP-TM} \) reductions for \( \Phi \) and clauses being non-overlapping in \( \Phi \). Thus the termination of \( \text{LP-TM} \) plays a crucial role not only in ensuring observational productivity, but also in ensuring the operational equivalence of \( \text{LP-Struct} \) and \( \text{LP-Unif} \), which in its turn is crucial for our proofs of soundness and completeness of \( \text{LP-Struct} \) with respect to \( \mathbf{H} \).

We give a formal analysis of properties of \( \text{LP-TM} \) resolution. We show how \( \text{LP-TM} \) relates to term rewriting systems by introducing a transformation method that translates any logic program without existential variables into a term rewriting system (we call this process functionalisation). After functionalisation, standard term rewriting methods for detecting termination can be applied. We also give an alternative transformational method that renders all logic programs \( \text{LP-TM} \) terminating and non-overlapping. The method has connection to Kleene’s realizability method [13], and we therefore call it realizability transformation.

The technical content of this paper is organized as follows.

- In Section 2 we prove soundness and completeness of \( \text{LP-Unif} \) with respect to the type system \( \mathbf{H} \). This means \( \mathbf{H} \) can be used to understand operational semantics of logic programming.
- In Section 3 we extend the results to \( \text{LP-Struct} \). We formally define \( \text{LP-Struct} \), and identify two conditions that ensure that \( \text{LP-Struct} \) is operationally equivalent to \( \text{LP-Unif} \).
- In Section 4 we define functionalisation and show the exact relation of \( \text{LP-TM} \) to term rewriting systems, and use existing termination detection techniques of term rewriting to detect termination of \( \text{LP-TM} \).
- In Section 5 we define realizability transformation and show that this transformation preserves operational meaning of logic programs. We use it to show the equivalence of \( \text{LP-Struct} \) and \( \text{LP-Unif} \) for the

| Name | \( \Phi_1 \) | \( \Phi_2 \) | \( \Phi_3 \) |
|------|-------------|-------------|-------------|
| Program | \( P(x) \Rightarrow P(x) \) | \( P(x) \Rightarrow P(f(x)) \) | \( P(x, y) \Rightarrow P(x, g(y)) \) |
| Query | \( P(x) \) | \( P(x) \) | \( P(x, y) \) |
| Productivity | None | Global, Observational, Local | Observational, Local at \( y \) at \( x \) |
transformed programs. As a corollary, we obtain soundness and completeness of LP-Struct relative to system H for transformed programs.

- In Section 4 we formally define partial resolution as an abstract reduction system and call it partial LP-Unif. Based on this formalism we define local productivity. Partial LP-Unif provides a possibility of shifting the focus from deciding entailment for a given query to programming with LP and computing substitutions.

Finally, in Sections 7 and 8 we survey related work and conclude the paper.

2. Horn-Formulas as Types

In this section, we use Howard’s type system H to model logic programming. We use an abstract reduction system (called LP-Unif) to model the small-step semantics of SLD-resolution. The purpose of this section is to set up a type-theoretic framework for the rest of the paper, where Horn formulas are viewed as types in a type system and resolution corresponds to the proof construction. We show the correspondence between the small-step semantics of resolution and the system H. This result can be viewed as an alternative to the classical-style soundness and completeness results for SLD-resolution relative to Herbrand models.

Definition 1 (Syntax).

Term $t ::= x \mid f(t_1, \ldots, t_n)$

Atomic Formula $A, B, C, D ::= P(t_1, \ldots, t_n)$

Formula $F ::= A \mid F \Rightarrow F' \mid \forall x.F$

Horn Formula $H ::= \forall x.A_1, \ldots, A_n \Rightarrow B$

Proof Evidence $p, e ::= \kappa \mid a \mid e'(\lambda a.e)$

Axioms/LP Programs $\Phi ::= \cdot \mid \kappa : H, \Phi \mid a : F, \Phi$

Proof evidence are lambda terms with the constant evidence denoted by $\kappa$. We write $A_1, \ldots, A_n \Rightarrow B$ as a short hand for $A_1 \Rightarrow \cdots \Rightarrow A_n \Rightarrow B$. We write $\forall x.F$ for quantifying over all free term variables in $F$, and $\forall x.F$ denotes $F$ or $\forall x.F$. We use $\overline{A}$ to denote $A_1, \ldots, A_n$, when the number $n$ is unimportant. If $n$ is zero for $\overline{A} \Rightarrow B$, then we write $\Rightarrow B$. Horn clause formulas have the form $\forall x.\overline{A} \Rightarrow B$, and queries are given by atomic formulas. We use FV($t$) to denote the set of all free term variables in $t$.

The following is a type system based on Howard’s work [8], intended to provide a type theoretic interpretation for LP.

Definition 2 (Howard’s System H for LP).

\[
\begin{align*}
(k : H) &\in \Phi & \text{Axiom} \\
\Phi \vdash k : H & \\
(a : F) &\in \Phi & \text{VAR} \\
\Phi \vdash a : F \\
\Phi \vdash e : \forall x.F & & \text{INST} \\
\Phi \vdash e : F & & \text{GEN} \\
\Phi \vdash \overline{A} & & \text{Cut} \\
\Phi \vdash e : \forall x.F & & \text{App} \\
\Phi \vdash e \Rightarrow F_1 & & \text{Cut} \\
\Phi \vdash e \Rightarrow F_2 & & \text{Cut} \\
\Phi \vdash \lambda x.e : F_1 & & \text{Cut} \\
\end{align*}
\]

Note that the type for the constant in the rule Axiom is required to be Horn formula. It has been observed that the CUT rule and proper axioms in intuitionistic sequent calculus can emulate LP [3](§13.4).

The following rule is a version of CUT rule, working only with Horn formulas.

\[
\begin{align*}
\Phi \vdash e_1 : \overline{A} &\Rightarrow D & \Phi \vdash e_2 : \overline{B} &\Rightarrow D \Rightarrow C \\
\Phi \vdash \lambda x.\lambda y.((e_2 y)(e_1 y)) : \overline{A} &\Rightarrow \overline{B} &\Rightarrow C & & \text{CUT}
\end{align*}
\]

We can use rules ABS and APP to emulate CUT rule, thus the CUT rule is admissible in Howard’s system H. We will use C to denote the deduction system that consists of rules Axiom, Cut, INST, and GEN. The subsystem C provides a natural framework to work with Horn formulas, but H is more expressive, since it allows full intuitionistic formulas, e.g. H would allow a formula of the form $(F_1 \Rightarrow F_2) \Rightarrow F_3$. Note that system C itself as a type system is not type preserving. Thus we often work with C through its embedding in H, which is type preserving and strongly normalizing.
Definition 3. Beta-reduction on proof evidences is defined as the congruence closure of the following relation: \((\lambda a.e) e' \rightarrow_{\beta} [e'/a]e\).

The following three theorems are standard for a type system such as H. For proofs we refer reader to Barendregt’s excellent book [3].

Theorem 1 (Strong Normalization). If \(\Phi \vdash e : F\) in H, then \(e\) is strongly normalizable with respect to beta-reduction on proof evidences.

Theorem 2 (Inversion). \(\Phi \vdash a : F\), then \((a : F') \in \Phi \) and \(\sigma F' \equiv F\) for some substitution \(\sigma\).

Theorem 3 (Type Preservation). \(\Phi \vdash e : F\) in H and \(e \rightarrow_{\beta} e'\), then \(\Phi \vdash e' : F\).

Definition 4 (Unification). We define \(t \sim_{\gamma} t', t\) is unifiable with \(t'\) with substitution \(\gamma\), if \(\{t = t'\} \rightarrow^{*} \gamma\). Note that the relation \(\rightarrow\) is defined as below.

\[
\begin{align*}
\{f(t_1, \ldots, t_n) = f(s_1, \ldots, s_m)\} \cup E & \rightarrow \{t_1 = s_1, \ldots, t_n = s_m\} \cup E \\
\{f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)\} \cup E & \rightarrow \perp \\
\{t = t\} \cup E & \rightarrow E \\
\{f(t_1, \ldots, t_n) = x\} \cup E & \rightarrow \{x = f(t_1, \ldots, t_n)\} \cup E \\
\{x = f(t_1, \ldots, t_n)\} \cup E & \rightarrow \perp \text{ if } x \in \text{FV}(f(t_1, \ldots, t_n)) \\
\{x = t\} \cup E & \rightarrow \{x = t\} \cup \{t/x\}E \text{ if } x \notin \text{FV}(t)
\end{align*}
\]

Unification can be routinely extended to atomic formulas. The symbol \(\perp\) denotes failure of unification. Below is a formulation of SLD-resolution as a reduction system, this formalism is following Nilsson and Maluszyński [20].

Definition 5 (LP-Unif reduction). Given axioms \(\Phi\), we define a reduction relation on the multiset of atomic formulas:

\(\Phi \vdash \{A_1, \ldots, A_i, \ldots, A_n\} \rightsquigarrow_{\kappa,\gamma}^{K,\gamma'} \{\gamma A_1, \ldots, \gamma B_1, \ldots, \gamma B_m, \ldots, \gamma A_n\}\) for any substitution \(\gamma'\), if there exists \(\kappa : \forall A_1, B_1, \ldots, B_n \Rightarrow C \in \Phi\) such that \(C \rightsquigarrow_{\gamma} A_i\).

The second subscript in the reduction is intended as a state, it will be updated by composition along with reductions. Notation \(\gamma \cdot \gamma'\) should be read as follows: the old state \(\gamma'\) is updated, producing a new state \(\gamma \cdot \gamma'\). We assume fresh names in the form of new numeric indices for the quantified variables each time the above rule is applied. We write \(\rightsquigarrow\) when we leave the associated state implicit. We use \(\rightsquigarrow^{*}\) to denote the reflexive and transitive closure of \(\rightsquigarrow\). Notation \(\rightsquigarrow_{\gamma}^{*}\) is used when the final state along the reduction path is \(\gamma\).

Given a program \(\Phi\) and a set of queries \(\{B_1, \ldots, B_n\}\), SLD-resolution uses LP-Unif reduction to reduce \(\{B_1, \ldots, B_n\}\).

Definition 6 (LP-Unif). Given a logic program \(\Phi\), LP-Unif is given by an abstract reduction system \((\Phi, \rightsquigarrow)\).

Example 2. Consider the following logic program \(\Phi\), consisting of Horn formulas labelled by \(\kappa_1, \kappa_2, \kappa_3\), defining connectivity for a graph with three nodes:

\[
\begin{align*}
\kappa_1 : & \forall x.\forall y.\forall z. \text{Connect}(x, y), \text{Connect}(y, z) \Rightarrow \text{Connect}(x, z) \\
\kappa_2 : & \Rightarrow \text{Connect}(Node_1, Node_2) \\
\kappa_3 : & \Rightarrow \text{Connect}(Node_2, Node_3)
\end{align*}
\]

The usual SLD-resolution for the query \(\text{Connect}(x, y)\) can be represented as the following LP-Unif reduction:
\[ \Phi \vdash \{ \text{Connect}(x, y) \} \leadsto_{\kappa_1, [x/x_1, y/z_1]} \{ \text{Connect}(x, y), \text{Connect}(y_1, y) \} \leadsto_{\kappa_2, [\text{Node}_3/x, \text{Node}_2/y_1, \text{Node}_1/x_1, y/z_1]} \{ \text{Connect}(\text{Node}_2, y) \} \leadsto_{\kappa_3, [\text{Node}_3/y, \text{Node}_2/x, \text{Node}_2/y_1, \text{Node}_1/x_1, \text{Node}_3/z_1]} \emptyset \]

The first reduction \( \leadsto_{\kappa_1, [x/x_1, y/z_1]} \) unifies the query \( \text{Connect}(x, y) \) with the head of the rule \( \kappa_1 \) (which is \( \text{Connect}(x_1, z_1) \) after renaming) with the substitution \( [x/x_1, y/z_1] \) (\( x_1 \) is replaced by \( x \) and \( z_1 \) is replaced by \( y \)). So the query is resolved with \( \kappa_1 \), producing the next queries: \( \text{Connect}(x, y_1), \text{Connect}(y_1, y) \). Note that the substitution in the subscript of \( \leadsto \) is a state that will be updated alongside the derivation. Thus we have an answer \([\text{Node}_3/y, \text{Node}_1/x] \) for the query \( \text{Connect}(x, y) \).

2.1. Soundness and Completeness of LP-Unif

We have introduced Howard’s system \( H \) and LP-Unif. Now we will show the soundness of LP-Unif, namely, that a reduction of LP-Unif gives rise to an intuitionistic proof. On the other hand, any first order ground evidence of type \( A \) in \( H \) corresponds to a successful LP-Unif reduction (which is the essence of completeness result).

**Lemma 1 (Soundness Lemma).** If \( \Phi \vdash \{ A \} \leadsto \{ B_1, ..., B_n \} \), then \( \Phi \vdash e : B_1, ..., B_n \Rightarrow \gamma A \) in \( C \).

**Proof.** By induction on the length of the reduction.

- **Base Case.** Suppose the length is one, namely, \( \Phi \vdash \{ A \} \leadsto_{\kappa, \gamma} \{ B_1, ..., B_n \} \). It implies \( (\kappa : \forall x B_1, ..., B_n \Rightarrow C) \in \Phi \), \( \gamma B_1 \equiv B_1 \) and \( \gamma \leadsto A \). So we have \( \Phi \vdash \kappa : \gamma B_1, ..., \gamma B_n \Rightarrow \gamma C \) by the rules Axiom and Inst.

- **Step Case.** Suppose \( \Phi \vdash \{ A \} \leadsto_{\kappa_1} \{ A_1, ..., A_i, ..., A_n \} \leadsto_{\kappa_2, \gamma, \gamma_i} \{ \gamma_2 A_1, ..., \gamma_2 B_1, ..., \gamma_2 B_m, ..., \gamma_2 A_n \} \), where \( \kappa : \forall \bar{x} B_1, ..., B_m \Rightarrow C \) and \( \gamma \sim \gamma_i A_i \). By inductive hypothesis (IH), we have \( \Phi \vdash e_1 : A_1, ..., A_n \Rightarrow \gamma_1 A \).

Since \( \gamma_2 \) is idempotent, we have \( \Phi \vdash \kappa : \gamma_2 B_1, ..., \gamma_2 B_m \Rightarrow \gamma_2 A_i \). Thus by Cut rule, we have \( \Phi \vdash e' : \gamma_2 A_1, ..., \gamma_2 B_1, ..., \gamma_2 B_m, ..., \gamma_2 A_n \Rightarrow \gamma_2 \gamma_1 A \) for some proof evidence \( e' \).

□

The soundness lemma above ensures that every LP-Unif reduction and its answer are meaningful. The usual notion of failure in LP can be understood as proving an implicational formula and success in reduction corresponds to a proof of an atomic formula. This interpretation is useful since some LP-Unif reductions can be nonterminating. In Section 4 we will use LP-Unif in a way that it does not have to resolve all the queries, but it still computes useful answers for the variable that we care about.

**Theorem 4 (Soundness of LP-Unif).** If \( \Phi \vdash \{ A \} \leadsto \emptyset \), then \( \Phi \vdash e : \forall \bar{x}. A \Rightarrow \gamma A \) in \( C \).

An evidence is ground if it does not contain free evidence variables. The proof of completeness relies on the strong normalization and the type preservation property of \( H \). We first show the normal form of the proof given by a successful LP-Unif reduction is first-order. We then show that, if an atomic formula is inhabited by a ground evidence, there exists a successful LP-Unif reduction.

**Definition 7 (First-Order Proof Evidence).** We define first-order proof evidence as follows.

- A proof evidence variable \( a \) and a constant \( \kappa \) are first-order.
- If \( n, n' \) are first-order, then \( n, n' \) is first-order.

**Lemma 2.** If \( n, n' \) are first-order, then \([n'/a]n \) is first-order.

**Lemma 3.** If \( \Phi \vdash e : [\forall \bar{x}. A] \Rightarrow B \) in \( C \), then either \( e \) is a proof evidence constant, variable or it is normalizable to the form \( \lambda \bar{a}. n \), where \( n \) is first-order normal proof evidence.

**Proof.** By induction on the derivation of \( \Phi \vdash e : [\forall \bar{x}. A] \Rightarrow B \).

- **Base Case.** Rule Axiom. Obvious.

- **Step Case.**

\[
\frac{\Phi \vdash e_1 : A \Rightarrow D \quad \Phi \vdash e_2 : B \Rightarrow D \quad \Phi \vdash \lambda \bar{a}. \overline{ab}(e_2 \overline{a}) : A \Rightarrow B \Rightarrow C}{\Phi \vdash \lambda \bar{a}. \overline{ab}(e_2 \overline{a}) : A \Rightarrow B \Rightarrow C \quad \text{CUT}}
\]
By IH, we know that $e_1 = \kappa$ or $e_1 = \lambda a.n_1$; $e_2 = \kappa'$ or $e_2 = \lambda b.\lambda d.n_2$, where $n_1, n_2$ are first-order. We know that $e_1, a$ will be normalizable to a first-order proof evidence. And $e_2, b$ will be normalized to either $\kappa'$ or $\lambda d.n_2$. So by Lemma 2 we conclude that $\lambda a.\lambda b.(e_2 b) (e_1 a)$ is normalizable to $\lambda a.\lambda b.n$ for some first-order normal term $n$.

- The GEN and INST cases are straightforward.

**Theorem 5.** If $\Phi \vdash e : \forall z. \Rightarrow B$ in $C$, then $e$ is normalizable to a first-order proof evidence.

Now let us prove the completeness theorem.

**Lemma 4.** If $e$ is a ground first-order evidence, then it is of the following form:

- $\kappa$
- $\kappa n_1 \ldots n_i$, where $n_i$ is ground first-order evidence for any $i$.

**Theorem 6 (Completeness of LP-Unif).** If $\Phi \vdash n : \Rightarrow A$ where $n$ is in ground first-order normal form in $H$, then $\Phi \vdash \{A\} \rightsquigarrow^* \emptyset$.

**Proof.** By induction on the structure of $n$.

- Base Case. $n = \kappa$. By inversion, we know $\kappa : \forall z. \Rightarrow A' \in \Phi$ and $\gamma A' \equiv A$ for some substitution $\gamma$. Thus $A' \rightsquigarrow_\gamma A$, which implies $\Phi \vdash \{A\} \rightsquigarrow^* \emptyset$.
- Step Case. $n = \kappa n_1 n_2 \ldots n_m$. By inversion, we have $\kappa : \forall z. C_1, \ldots, C_m \Rightarrow B \in \Phi$. To obtain $\Phi \vdash n : \Rightarrow A$, by inversion we have $\Phi \vdash \kappa : \forall z. C_1, \ldots, C_m \Rightarrow B$ with $\gamma_m \ldots \gamma_1(B) \equiv A$, and $\Phi \vdash n_1 : \Rightarrow C_1$, $\Phi \vdash n_2 : \Rightarrow \gamma_m C_2, \ldots, \Phi \vdash n_m : \Rightarrow \gamma_m C_1, \ldots, \gamma_1 C_m$. By the rule $\text{INST}$, we have $\Phi \vdash n_1 : \Rightarrow \gamma_m \ldots \gamma_1 C_1$, $\Phi \vdash n_2 : \Rightarrow \gamma_m \ldots \gamma_1 C_2, \ldots, \Phi \vdash n_m : \Rightarrow \gamma_m \ldots \gamma_1 C_m$. Thus we have $\Phi \vdash \{A\} \rightsquigarrow_{\kappa, \gamma_m \ldots \gamma_1} \{\gamma_m \ldots \gamma_1 C_1, \ldots, \gamma_m \ldots \gamma_1 C_m\}$. By IH, we have $\Phi \vdash \{\gamma_m \ldots \gamma_1 C_1\} \rightsquigarrow^* \emptyset$. So $\Phi \vdash \{A\} \rightsquigarrow_{\kappa, \gamma_m \ldots \gamma_1} \rightsquigarrow^* \{\gamma_m \ldots \gamma_1 C_2, \ldots, \gamma_m \ldots \gamma_1 C_m\}$.

Again, we have $\Phi \vdash n_2 : \Rightarrow \sigma_1 \gamma_m \ldots \gamma_1 C_2$ by rule $\text{INST}$. By applying IH repeatedly, we obtain $\Phi \vdash \{A\} \rightsquigarrow^* \emptyset$.

3. Structural Resolution

In this section, we represent structural resolution using the abstract reduction formalism we have seen in Section 2. We first define structural resolution as $\text{LP-Struct}$ reduction, thereby also defining $\text{LP-TM}$ reduction, which replaces unification by term-matching. We then identify two conditions that will make LP-Unif and $\text{LP-Struct}$ operationally equivalent. These two conditions are termination of $\text{LP-TM}$ and non-overlapping of Horn clauses.

**Definition 8.**

- **Term-matching($\text{LP-TM}$) reduction:** $\Phi \vdash \{A_1, \ldots, A_i, \ldots, A_n\} \rightsquigarrow_{\kappa, \gamma'} \{A_1, \ldots, \sigma B_1, \ldots, \sigma B_m, \ldots, A_n\}$ for any substitution $\gamma'$, if there exists $\kappa : \forall z. B_1, \ldots, B_n \Rightarrow C \in \Phi$ such that $\sigma C \equiv A_i$.
- **Substitutional reduction:** $\Phi \vdash \{A_1, \ldots, A_i, \ldots, A_n\} \rightsquigarrow_{\kappa, \gamma, \gamma'} \{\gamma A_1, \ldots, \gamma A_i, \ldots, \gamma A_n\}$ for any substitution $\gamma'$, if there exists $\kappa : \forall z. B_1, \ldots, B_n \Rightarrow C \in \Phi$ such that $C \rightsquigarrow_\gamma A_i$.

The second subscript of term-matching reduction is used to store the substitutions obtained by unification. It is only used when we combine LP-TM reduction with substitutional reductions, we can ignore the substitution when we only talk about LP-TM reduction. The second subscript in the substitutional reduction is intended as a state (similar to LP-Unif), it will be updated along with reductions.

Given a program $\Phi$ and a set of queries $\{B_1, \ldots, B_n\}$, LP-TM uses only term-matching reduction to reduce $\{B_1, \ldots, B_n\}$:

**Definition 9 (LP-TM).** Given a logic program $\Phi$, LP-TM is given by an abstract reduction system $(\Phi, \rightarrow)$.

LP-TM is also sound w.r.t. the type system of Definition 2 which implies that we can obtain a proof for each reductoin of the query.
Theorem 7 (Soundness of LP-TM). If \( \Phi \vdash \{ A \} \rightarrow^* \{ B_1, \ldots, B_n \} \), then \( \Phi \vdash e : \forall x, B_1, \ldots, B_n \Rightarrow A \) in C.

Comparing the soundness lemma and Theorem 7, we see that for LP-TM, there is no need to accumulate substitutions, this is due to the use of term-matching instead of unification for the LP-TM reduction.

We use \( \rightarrow^\mu \) to denote a reduction path to a \( \rightarrow \)-normal form. If the \( \rightarrow \)-normal form does not exist, then \( \rightarrow^\mu \) denotes an infinite reduction path. We write \( \rightarrow^1 \) to denote at most one step of \( \rightarrow \).

We can now formally define structural resolution within our formal framework. Given a program \( \Phi \) and a set of queries \( \{ B_1, \ldots, B_n \} \), LP-Struct first uses term-matching reduction to reduce \( \{ B_1, \ldots, B_n \} \) to a normal form, then performs one step substitutional reduction, and then repeats this process.

Definition 10 (Structural Resolution (LP-Struct)). Given a logic program \( \Phi \), LP-Struct is given by an abstract reduction system \( (\Phi, \rightarrow^\mu \cdot \rightarrow^1) \).

If a finite term-matching reduction path does not exist, then \( \rightarrow^\mu \cdot \rightarrow^1 \) denotes an infinite path. When we write \( \Phi \vdash \{ A \} (\rightarrow^\mu \cdot \rightarrow^1)^* \{ C \} \), it means a nontrivial finite path will be of the shape \( \Phi \vdash \{ A \} \rightarrow^\mu \cdot \rightarrow^1 \cdots \rightarrow^\mu \cdot \rightarrow^1 \{ C \} \).

Now let us recall the execution trace of Stream \( \kappa_1 : \forall x. \forall y. \text{Stream}(y) \Rightarrow \text{Stream}((\text{Cons}(0, y))) \) on the query \( \text{Stream}(x) \) using LP-Struct:

\[
\begin{align*}
\Phi \vdash \{ \text{Stream}(x) \} & \rightarrow_{\kappa_1, ([\text{Cons}(0, y_1)/x]} \{ \text{Stream}((\text{Cons}(0, y_1))) \} \rightarrow_{\kappa_1} \\
\{ \text{Stream}(y_1) \} & \rightarrow_{\kappa_1, ([\text{Cons}(0, y_1)/y_1, \text{Cons}(0, \text{Cons}(0, y_2))/y_1]} \{ \text{Stream}((\text{Cons}(0, y_2))) \} \rightarrow_{\kappa_1} \\
\{ \text{Stream}(y_2) \} & \rightarrow_{\kappa_1, ([\text{Cons}(0, y_2)/y_2, \text{Cons}(0, \text{Cons}(0, y_3))/y_2, \text{Cons}(0, \text{Cons}(0, y_3))/y_1, \text{Cons}(0, \text{Cons}(0, \text{Cons}(0, y_3))/y_1]} \{ \text{Stream}((\text{Cons}(0, y_3))) \} \rightarrow_{\kappa_1} \\
\{ \text{Stream}(y_3) \} & \rightarrow \cdots
\end{align*}
\]

3.1. LP-Struct and LP-Unif

LP-Struct exhibits different execution behavior compared to LP-Unif. In general, they are not equivalent. Consider the program and the finite LP-Unif derivation of Example 2. LP-Unif has a finite successful derivation for the query Connect\((x, y)\), but we have the following non-terminating reduction by LP-Struct:

\[
\Phi \vdash \{ \text{Connect}(x, y) \} \rightarrow_{\kappa_1} \{ \text{Connect}(x, y_1), \text{Connect}(y_1, y) \} \\
\rightarrow_{\kappa_1} \{ \text{Connect}(x, y_2), \text{Connect}(y_2, y_1), \text{Connect}(y_1, y) \} \rightarrow_{\kappa_1} \cdots
\]

The diverging behavior above is due to the divergence of LP-TM reduction.

Definition 11 (LP-TM Termination). We say a program \( \Phi \) is LP-TM terminating iff it admits no infinite \( \rightarrow \)-reduction.

LP-TM termination is important for LP-Struct in two aspects: 1. It is one of the conditions that ensure the operational equivalence of LP-Struct and LP-Unif. 2. The finiteness of LP-TM reductions is one of the two conditions defining observational productivity in LP.

The following example shows that LP-TM termination alone is not sufficient to establish that LP-Unif and LP-Struct are operationally equivalent.

Example 3.

\[
\kappa_1 : \Rightarrow P(C) \\
\kappa_2 : \forall x. Q(x) \Rightarrow P(x)
\]

Here \( C \) is a constant. The program is LP-TM terminating. For query \( P(x) \), we have \( \Phi \vdash \{ P(x) \} \rightarrow_{\kappa_1, [C/x]} \emptyset \) with LP-Unif, but there is only one reduction path \( \Phi \vdash \{ P(x) \} \rightarrow_{\kappa_2} \{ Q(x) \} \not\rightarrow \) for LP-Struct.

Thus the termination of LP-TM is insufficient for establishing the relation between LP-Struct and LP-Unif. In Example 3, the problem is caused by the overlapping heads \( P(C) \) and \( P(x) \). Motivated by the notion of non-overlapping rules in term rewriting systems ([2] [23]), we introduce the following definition.

Definition 12 (Non-overlapping Condition). Axioms \( \Phi \) are non-overlapping if for any \( \kappa_1 : \forall x. B \Rightarrow C, \kappa_2 : \forall x. D \Rightarrow E \in \Phi \), there are no substitution \( \sigma, \delta \) such that \( \sigma C \equiv \delta E \).

The following lemma shows that an LP-TM step can be viewed as an LP-Unif step without affecting the accumulated substitution.
Lemma 5. If $\Phi \vdash \{ D_1, ..., D_i, ..., D_n \} \rightarrow_{\kappa, \gamma} \{ D_1, ..., \sigma E_1, ..., \sigma E_m, ..., D_n \}$, with $\kappa : \forall x. E \Rightarrow C \in \Phi$ and $\sigma C \equiv D_i$ for any $\gamma$, then $\Phi \vdash \{ D_1, ..., D_i, ..., D_n \} \rightarrow_{\kappa, \gamma} \{ D_1, ..., \sigma E_1, ..., \sigma E_m, ..., D_n \}$.

Proof. Since for $\Phi \vdash \{ D_1, ..., D_i, ..., D_n \} \rightarrow_{\kappa, \gamma} \{ D_1, ..., \sigma E_1, ..., \sigma E_m, ..., D_n \}$, with $\kappa : \forall x. E \Rightarrow C \in \Phi$ and $\sigma C \equiv D_i$, we have $\Phi \vdash \{ D_1, ..., D_i, ..., D_n \} \rightarrow_{\kappa, \sigma, \gamma} \{ \sigma D_1, ..., \sigma E_1, ..., \sigma E_m, ..., \sigma D_n \}$. But dom($\sigma$) $\in$ FV($C$), thus we have $\Phi \vdash \{ D_1, ..., D_i, ..., D_n \} \rightarrow_{\kappa, \gamma} \{ D_1, ..., D_i, ..., D_n \}$.

The following lemma shows that several LP-Struct steps corresponds to one LP-Unif step, given the non-overlapping requirement.

Lemma 6. Suppose $\Phi$ is non-overlapping and $\{ A_1, ..., A_n \}$ are $\rightarrow$-normal. If $\Phi \vdash \{ A_1, ..., A_n \} (\rightarrow_{\kappa, \gamma} \rightarrow_{\mu} \rightarrow_{\gamma}) \{ C_1, ..., C_m \}$, then $\Phi \vdash \{ A_1, ..., A_n \} \rightarrow_{\gamma} \{ C_1, ..., C_m \}$.

Proof. Given $\Phi \vdash \{ A_1, ..., A_n \} (\rightarrow_{\kappa, \gamma} \rightarrow_{\mu} \rightarrow_{\gamma}) \{ C_1, ..., C_m \}$, we know the actual reduction path can be rearrange to the form $\Phi \vdash \{ A_1, ..., A_n \} \rightarrow_{\kappa, \gamma} \{ \gamma A_1, ..., \gamma A_n \} \rightarrow_{\kappa, \gamma} \{ \gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n \} \rightarrow_{\mu} \{ C_1, ..., C_m \}$, where $\gamma A_i \equiv \gamma C$ with $\kappa : \forall x. D \Rightarrow C \in \Phi$ and $A_i \equiv \sigma B$. Note that $\gamma$ is unchanged along the term-matching reduction. We rearrange the $\rightarrow$ following right after $\rightarrow$ using $\kappa$ due to the property of LP-TM. Note that to LP-TM reduce $A_i$ we only use $\kappa$, since otherwise it would mean with $\kappa' : \forall x. D \Rightarrow B \in \Phi$. This implies $\gamma C \equiv \gamma \sigma B$, contradicting the non-overlapping restriction. Thus we have $\Phi \vdash \{ A_1, ..., A_n \} \rightarrow_{\kappa, \gamma} \{ \gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n \} \rightarrow_{\gamma} \{ C_1, ..., C_m \}$.

The Lemma and Theorem below show that for non-overlapping program, LP-Unif is equivalent to LP-Struct if the overall reduction terminates.

Lemma 7. Given $\Phi$ is non-overlapping, if $\Phi \vdash \{ A_1, ..., A_n \} (\rightarrow_{\mu} \rightarrow_{\gamma}) \{ C_1, ..., C_m \}$ with $\{ C_1, ..., C_m \}$ in $\rightarrow_{\mu} \rightarrow_{\gamma}$-normal form, then $\Phi \vdash \{ A_1, ..., A_n \} \rightarrow_{\gamma} \{ C_1, ..., C_m \}$ with $\{ C_1, ..., C_m \}$ in $\rightsquigarrow$-normal form.

Proof. Since $\Phi \vdash \{ A_1, ..., A_n \} (\rightarrow_{\mu} \rightarrow_{\gamma}) \{ C_1, ..., C_m \}$, this means the reduction path must be of the form $\Phi \vdash \{ A_1, ..., A_n \} \rightarrow_{\mu} \rightarrow_{\gamma} \rightarrow_{\mu} \rightarrow_{\gamma} \rightarrow_{\mu} \rightarrow_{\gamma} \rightarrow_{\gamma} \{ C_1, ..., C_m \}$. Thus $\Phi \vdash \{ A_1, ..., A_n \} \rightarrow_{\mu} (\rightarrow_{\gamma} \rightarrow_{\gamma}) (\rightarrow_{\gamma} \rightarrow_{\gamma}) \rightarrow_{\gamma} \{ C_1, ..., C_m \}$. By Lemma and Lemma we have $\Phi \vdash \{ A_1, ..., A_n \} \rightarrow_{\gamma} \{ C_1, ..., C_m \}$ with $\{ C_1, ..., C_m \}$ in $\rightsquigarrow$-normal form.

Lemma 8. Given $\Phi$ is a non-overlapping, if $\Phi \vdash \{ A_1, ..., A_n \} \rightarrow_{\gamma} \{ C_1, ..., C_m \}$ with $\{ C_1, ..., C_m \}$ in $\rightsquigarrow$-normal form, then $\Phi \vdash \{ A_1, ..., A_n \} (\rightarrow_{\mu} \rightarrow_{\gamma}) \{ C_1, ..., C_m \}$ with $\{ C_1, ..., C_m \}$ in $\rightarrow_{\mu} \rightarrow_{\gamma}$-normal form.

Proof. By induction on the length of $\rightarrow_{\gamma}$.

- Base Case. $\Phi \vdash \{ A_1, ..., A_i, ..., A_n \} \rightarrow_{\kappa, \gamma} \{ \gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n \}$ with $\kappa : \forall x. D \Rightarrow C \in \Phi$, $C \equiv \gamma A_i$ and $\{ A_1, ..., \gamma A_i, ..., A_n \} \rightarrow_{\kappa, \gamma} \{ \gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n \}$ in $\rightsquigarrow$-normal form. We have $\Phi \vdash \{ A_1, ..., \gamma A_i, ..., A_n \} \rightarrow_{\kappa, \gamma} \{ \gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n \}$ with $\{ \gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n \}$ in $\rightarrow_{\mu} \rightarrow_{\gamma}$-normal form. Note that there can not be another $\kappa' : \forall x. D \Rightarrow C' \in \Phi$ such that $\sigma C' \equiv A_i$, since this would means $\gamma C \equiv \gamma A_i \equiv \gamma \sigma C'$, violating the non-overlapping requirement.

- Step Case. $\Phi \vdash \{ A_1, ..., A_i, ..., A_n \} \rightarrow_{\kappa, \gamma} \{ \gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n \}$ with $\kappa : \forall x. D \Rightarrow C \in \Phi$ and $C \equiv \gamma A_i$. We have $\Phi \vdash \{ A_1, ..., A_i, ..., A_n \} \rightarrow_{\kappa, \gamma} \{ \gamma A_1, ..., \gamma A_i, ..., \gamma A_n \} \rightarrow_{\gamma} \{ \gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n \}$. By the non-overlapping requirement, there can not be another $\kappa' : \forall x. D \Rightarrow C' \in \Phi$ such that $\sigma C' \equiv A_i$. By IH, we have $\Phi \vdash \{ \gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n \} (\rightarrow_{\mu} \rightarrow_{\gamma}) \{ C_1, ..., C_m \}$. Thus we conclude that $\Phi \vdash \{ A_1, ..., A_i, ..., A_n \} (\rightarrow_{\mu} \rightarrow_{\gamma}) \{ C_1, ..., C_m \}$.

□

Theorem 8. Suppose $\Phi$ is non-overlapping. $\Phi \vdash \{ A_1, ..., A_n \} \rightarrow_{\gamma} \{ C_1, ..., C_m \}$ with $\{ C_1, ..., C_m \}$ in $\rightsquigarrow$-normal form iff $\Phi \vdash \{ A_1, ..., A_n \} \rightarrow_{\mu} \rightarrow_{\gamma} \{ C_1, ..., C_m \}$ with $\{ C_1, ..., C_m \}$ in $\rightarrow_{\mu} \rightarrow_{\gamma}$-normal form.

The theorem above implies that for terminating and non-overlapping program, LP-Unif is equivalent to LP-Struct. But the termination requirement can be relaxed by only requiring termination of the $\rightarrow$-reduction, i.e. by requiring termination of LP-TM.

Lemma 9. If $\Phi \vdash \{ A \} \rightarrow_{\kappa, \gamma} \{ B_1, ..., B_m \}$ and dom($\sigma$) $\cap (\bigcup \text{FV}(B_i)) \setminus \text{FV}(A) = \emptyset$, then $\Phi \vdash \{ \sigma A \} \rightarrow_{\kappa, \gamma} \{ \sigma B_1, ..., \sigma B_m \}$.
Note that the above lemma shows that the reduction → is closed under substitution only under the condition that dom(σ) \cap (\bigcup_i \text{FV}(B_i)) = \emptyset, i.e. the domain of the substitution must not contain any variable that are in B_i but not in A for any i, otherwise it will not be the case. If Φ ⊢ \{A\} →^\mu \{B_1, ..., B_m\}, we write [A] to mean the normal form of A, i.e. B_1, ..., B_m.

**Theorem 9 (Equivalence of LP-Struct and LP-Unif).** Suppose Φ is non-overlapping and LP-TM terminating.

1. If Φ ⊢ \{A_1, ..., A_n\} \equiv \{B_1, ..., B_m\}, then Φ ⊢ \{A_1, ..., A_n\} →^\mu \rightarrow^1 \ast \{C_1, ..., C_l\} and Φ ⊢ \{B_1, ..., B_m\} →^* \{C_1, ..., C_l\}.
2. If Φ ⊢ \{A_1, ..., A_n\} →^{μ \cdot \rightarrow^1 \ast} \{B_1, ..., B_m\}, then Φ ⊢ \{A_1, ..., A_n\} →^* \{B_1, ..., B_m\}.

**Proof.** 1. Suppose Φ ⊢ \{A_1, ..., A_n\} ∼_{κ,γ} \{γA_1, ..., γE_1, ..., γE_i, ..., γA_n\}, with κ : E \rightharpoonup D ∈ Φ and D ∼_{γ} A_i. Suppose γD ≡ A_i, we have Φ ⊢ \{A_1, ..., A_n\} →_{κ,γ} \{γA_1, ..., γE_1, ..., γE_i, ..., γA_n\} →^\mu \{C_1, ..., C_l\}. Suppose γD ≠ A_i. In this case, we have Φ ⊢ \{A_1, ..., A_n\} →^\mu \{[A_1], ..., A_i, ..., [A_n]\} →_{κ,γ} \{[γA_1], ..., γE_1, ..., γE_i, ..., γA_n\} →^\mu \{C_1, ..., C_l\}. By Lemma 5 we know that Φ ⊢ γA_i →^\mu \{[γA_i]\}, ..., Φ ⊢ γA_n →^\mu \{[γA_n]\}. Thus Φ ⊢ \{γA_1, ..., γE_1, ..., γE_i, ..., γA_n\} →^\mu \{C_1, ..., C_l\}.

2. We just need to show that if \{A_1, ..., A_n\} are → normal and Φ ⊢ \{A_1, ..., A_n\} ∼_{κ,γ} \{γA_1, ..., γA_n\} →^\mu \{B_1, ..., B_m\}, then Φ ⊢ \{A_1, ..., A_n\} ∼_{κ} \{B_1, ..., B_m\}. Suppose Φ ⊢ \{A_1, ..., A_n\} ∼_{κ,γ} \{γA_1, ..., γA_n\} →^\mu \{B_1, ..., B_m\}, we have Φ ⊢ \{A_1, ..., A_n\} →_{κ,γ} \{γA_1, ..., γA_n\} →_{κ} \{γA_1, ..., γC_1, ..., γC_l, ..., γA_n\} →^\mu \{B_1, ..., B_m\} with κ : C \rightharpoonup D ∈ Φ and D ∼_{γ} A_i. Thus we have Φ ⊢ \{A_1, ..., A_n\} ∼_{κ,γ} \{γA_1, ..., γC_1, ..., γC_l, ..., γA_n\}. By Lemma 5 we have Φ ⊢ \{A_1, ..., A_n\} ∼_{κ,γ} \{γA_1, ..., γC_1, ..., γC_l, ..., γA_n\} →^\mu \{B_1, ..., B_m\}.

Note that the above theorem does not rely on the whole program termination, therefore it establishes equivalence of LP-Unif and LP-Struct even for nonterminating programs like the Stream example, as long as they are LP-TM terminating and non-overlapping. This result has not been described in previous work.

### 3.2. Discussion

Structural resolution was first introduced in Komendantskaya and Power’s work [15, 16] under the name of coalgebraic logic programming. It was further developed into a resolution method based on resolution trees (called “rewriting trees”) generated by term-matching [10, 14]. The formulation of LP-Struct in this paper is based on the abstract reduction system framework, instead of the tree formalism in previous work. As a consequence, for overlapping logic programs, the reduction-based LP-Struct behaves differently compared to the tree-based formalism (see e.g. Example 5). The novelty of our development in this section is the articulation of the two conditions that ensure the operational equivalence of LP-Struct and LP-Unif (Theorem 9).

### 4. Functionalisation of LP-TM

One of the important features of LP-Struct is that it refines SLD-resolution by a combination of term-matching and unification. LP-TM itself is used in the type class context reduction [12]. The termination behavior of LP-TM is of practical interest. For example, detecting termination for the type class inference is essential to achieve decidability of the type inference in languages such as Haskell [17, Section 5].

Of course, detecting termination of LP-TM also means ensuring observational productivity in the context of LP-Struct. As explained Introduction and Section 3, LP-TM termination is not only essential to ensure the equivalence of LP-Struct and LP-Unif, but also is important to allow viewing the LP-TM reductions within nonterminating LP-Struct reduction as finite observations.

On the other hand, termination and nontermination detection are well-studied in the context of term rewriting. In this section we show a method that reuses the techniques developed in term rewriting to detect termination of LP-TM. We first define a process called functionalisation that transforms a set of Horn clauses into a set of rewrite rules, where execution of a query is seen as a process of rewriting the query to its proof.
As a result, termination and nontermination detection techniques from term rewriting can be applied to LP-TM, assuming logic programs contain no existential variables.

In this section we work only with the Horn formulas without existential variables, i.e. for any Horn formula \( \forall x_1, \ldots, x_n \Rightarrow B \), we have \( \bigcup FV(A_i) \subseteq FV(B) \). The restriction that Horn clauses should not contain existential variables comes directly from a similar requirement imposed in term-rewriting.

Since the idea of functionalisation is to view LP-TM resolution for a query as a rewriting process to its proof evidence, the rewriting is defined on mixed terms, i.e. a mixture of atomic formulas and proof evidence.

**Definition 13 (Mixed Terms).**

\[
\text{Mixed term context } \mathcal{C} := \bullet \mid \mathcal{C} q \mid q \mathcal{C}
\]

Mixed term context \( \mathcal{C} \) is defined as in Definition 14. 

**Definition 14 (Functionalisation).**

We can construct a set of rewrite rules \( K(\Phi) \) from a set of axioms \( \Phi \) as follows. For each \( \kappa : \forall x_1, \ldots, x_n \Rightarrow B \in \Phi \), we define a rewrite rule \( B \rightarrow \kappa \ A_1 \ldots A_n \in K(\Phi) \) on mixed terms. We call \( \kappa \) an axiom symbol.

**Lemma 10.** \( \Phi \vdash \{ A_1, \ldots, A_n \} \rightarrow \{ A_1, \ldots, \sigma B_1, \ldots, \sigma B_m, \ldots, A_n \} \), where \( \kappa : \forall x_1, \ldots, x_n \Rightarrow B \in \Phi \) and \( \sigma B \equiv A_i \) if \( \mathcal{C}[A_1, \ldots, A_i, \ldots, A_n] \rightarrow \mathcal{C}'[A_1, \ldots, \sigma B_1, \ldots, \sigma B_m, \ldots, A_n] \), where \( \mathcal{C}, \mathcal{C}' \) do not contain any atomic formulas and \( \mathcal{C} \rightarrow \mathcal{C}' \).

**Proof.** By Definition 14, \( \kappa : \forall x_1, \ldots, x_n \Rightarrow B \in \Phi \) implies \( A_i \rightarrow \kappa B_1 \ldots B_m \in K(\Phi) \), and vice versa. So \( \mathcal{C}' \) can be obtained by replacing the \( i \)-th \( \bullet \) in \( \mathcal{C} \) by \( \kappa \bullet_1 \ldots \bullet_m \).

**Lemma 11.** \( \Phi \vdash \{ A_1, \ldots, A_n \} \rightarrow \{ C_1, \ldots, C_l \} \) iff \( \mathcal{C}[A_1, \ldots, A_n] \rightarrow \mathcal{C}'[C_1, \ldots, C_l] \), where \( \mathcal{C} \rightarrow^* \mathcal{C}' \) and \( \mathcal{C}, \mathcal{C}' \) do not contain any atomic formulas.

**Proof.** We prove both directions together. By induction on the length of \( \rightarrow^* \).

**Base Case.** By Lemma 10.

**Step Case.**

**Left to Right:**

Suppose \( \Phi \vdash \{ A_1, \ldots, A_n \} \rightarrow \{ A_1, \ldots, \sigma B_1, \ldots, \sigma B_m, \ldots, A_n \} \rightarrow^* \{ C_1, \ldots, C_l \} \), with \( \kappa : \forall x_1, \ldots, x_n \Rightarrow B \in \Phi \) and \( \sigma B \equiv A_i \). Then we know \( \mathcal{C}[A_1, \ldots, A_n] \rightarrow \mathcal{C}'[A_1, \ldots, \sigma B_1, \ldots, \sigma B_m, \ldots, A_n] \). Also, \( \mathcal{C} \rightarrow \mathcal{C}' \), where \( \mathcal{C}' \) can be obtained from \( \mathcal{C} \) by replacing its \( i \)-th \( \bullet \) by \( \kappa \bullet_1 \ldots \bullet_m \). By IH, \( \mathcal{C}'[A_1, \ldots, \sigma B_1, \ldots, \sigma B_m, \ldots, A_n] \rightarrow^* \mathcal{C}'[C_1, \ldots, C_l] \) with \( \mathcal{C}' \rightarrow^* \mathcal{C}' \). So \( \mathcal{C}[A_1, \ldots, A_n] \rightarrow^* \mathcal{C}'[C_1, \ldots, C_l] \) with \( \mathcal{C} \rightarrow^* \mathcal{C}' \).

**Right to Left:**

Suppose \( \mathcal{C}[A_1, \ldots, A_n] \rightarrow \mathcal{C}'[A_1, \ldots, \sigma B_1, \ldots, \sigma B_m, \ldots, A_n] \rightarrow^* \mathcal{C}'[C_1, \ldots, C_l] \) with \( \mathcal{C} \rightarrow \mathcal{C}' \rightarrow^* \mathcal{C}' \), where \( B \rightarrow \kappa B_1 \ldots B_m \) and \( \sigma B \equiv A_i \). So \( \mathcal{C}[A_1, \ldots, A_n] \rightarrow \mathcal{C}'[C_1, \ldots, C_l] \) with \( \mathcal{C} \rightarrow \mathcal{C}' \rightarrow^* \mathcal{C}' \). Thus \( \Phi \vdash \{ A_1, \ldots, A_n \} \rightarrow \{ A_1, \ldots, \sigma B_1, \ldots, \sigma B_m, \ldots, A_n \} \rightarrow^* \{ C_1, \ldots, C_l \} \). Thus, we have \( \Phi \vdash \{ A_1, \ldots, A_n \} \rightarrow^* \{ C_1, \ldots, C_l \} \).

**Theorem 10.** \( \Phi \vdash \{ A \} \rightarrow^* \emptyset \) iff \( A \rightarrow^* e \), and \( e \) is a ground evidence. As a consequence, the query \( A \) is LP-TM (non)terminating iff \( A \) is (non)terminating for \( K(\Phi) \).

**Proof.** By Lemma 11.

In practice, functionalisation can also be used to implement LP-TM, especially if computing the proof evidence is the only goal. This is the case for type class inference [1], for example.

Now we demonstrate how to apply a convenient termination technique in term rewriting called dependency pair method [1] to analyze the termination behavior of LP-TM. The next definitions and a theorem follow [1].
**Example 4.** Consider the following program $\Phi$:

The dependency pairs of $\Phi$ are meaning preserving realizability transformation logic programs that are not LP-TM terminating but are still meaningful from the LP-Unif perspective (cf. Functionalisation provides a way to detect LP-TM termination for LP-Struct. But sometimes there are 5. Realizability Transformation and LP-Struct

**Theorem 11 (Arts-Giesel [1]).** $K(\Phi)$ is terminating iff no infinite $K(\Phi)$-chain exist.

Theorem [11] allows us to detect the termination of $K(A)$ by looking at the possible $K(\Phi)$-chain.

**Example 4.** Consider the following program $\Phi$:

$$
\begin{align*}
\kappa_1 : & \Rightarrow P(\text{Int}) \\
\kappa_2 : & \forall x. P(x), P(\text{List}(x)) \Rightarrow P(\text{List}(x))
\end{align*}
$$

The dependency pairs of $\Phi$ are $P(\text{List}(x)) \rightarrow P(\text{List}(x))$ and $P(\text{List}(x)) \rightarrow P(x)$. We can see $P(\text{List}(x)) \rightarrow P(\text{List}(x))$ can form an infinite $E(K(\Phi))$-chain, thus $K(\Phi)$ is not terminating. So $\Phi$ is not LP-TM terminating.

**5. Realizability Transformation and LP-Struct**

Functionalisation provides a way to detect LP-TM termination for LP-Struct. But sometimes there are logic programs that are not LP-TM terminating but are still meaningful from the LP-Unif perspective (cf. Example [2]. For these programs, we still want to be able to use LP-Struct. To solve this problem, we define a meaning preserving realizability transformation that transforms any logic program into LP-TM terminating one.

Realizability [13] ([82]) is a technique that uses a number representing the proof of a number-theoretic formula. The transformation described here is similar in the sense that we use a first-order term to represent the proof of a Horn formula. More specifically, we use a first-order term as an extra argument for Horn formula to represent a proof of that formula.

Lemma [3] and Theorem [5] show that we can use first-order terms to represent normalized proof evidences, and thus justify this method.

**Definition 18 (Representing First-Order Proof Evidences).** Let $\phi$ be a mapping from proof evidence variables to first-order terms. We define a representation function $[[\cdot]]_\phi$ from first-order normal proof evidences to first-order terms.

- $[[a]]_\phi = \phi(a)$.
- $[[\kappa p_1...p_n]]_\phi = f_\kappa([[p_1]]_\phi,...,[[p_n]]_\phi)$, where $f_\kappa$ is a function symbol.

Let $A \equiv P(t_1,...,t_n)$ be an atomic formula and $t'$ be a term such that $(\bigcup_i \text{FV}(t_i)) \cap \text{FV}(t') = \emptyset$, we write $A[t']$ to abbreviate a new atomic formula $P(t_1,...,t_n,t')$.

**Definition 19 (Realizability Transformation).** We define a transformation $F$ on Horn formula and its normalized proof evidence:

- $F(\kappa : \forall x.A_1,...,A_m \Rightarrow B) = \kappa : \forall x.\forall y.A_1[y_1],...,A_m[y_m] \Rightarrow B[f_\kappa(y_1,...,y_m)]$, where $y_1,...,y_m$ are all fresh and distinct.
- $F(\lambda x.n : [\forall x.A_1,...,A_m \Rightarrow B]) = \lambda x.n : [\forall x.\forall y.A_1[y_1],...,A_m[y_m] \Rightarrow B[[n]]_{y/x}]$, where $y_1,...,y_m$ are all fresh and distinct.

The realizability transformation systematically associates a proof to each predicate, so that the proof can be recorded alongside with reductions. Let $F(\Phi)$ mean applying the realizability transformation to every axiom in $\Phi$.

12
Example 5. The following logic program \( F(\Phi) \) is the result of applying realizability transformation on the program \( \Phi \) in Example 2.

\[
\begin{align*}
\kappa_1 : \forall x.\forall y.\forall u_1.\forall u_2. & \text{Connect}(x, y, u_1), \text{Connect}(y, z, u_2) \Rightarrow \text{Connect}(x, z, f_{\kappa_1}(u_1, u_2)) \\
\kappa_2 : & \Rightarrow \text{Connect}(\text{Node}_1, \text{Node}_2, C_{\kappa_2}) \\
\kappa_3 : & \Rightarrow \text{Connect}(\text{Node}_2, \text{Node}_3, C_{\kappa_3})
\end{align*}
\]

Before the realizability transformation, we have the following judgement in \( H \):

\[
\Phi \vdash \lambda b.(\kappa_1 \ b) \ \kappa_2 : \text{Connect}(\text{Node}_2, z) \Rightarrow \text{Connect}(\text{Node}_1, z)
\]

We can apply the transformation, we get:

\[
F(\Phi) \vdash \lambda b.(\kappa_1 \ b) \ \kappa_2 : \text{Connect}(\text{Node}_2, z, u_1) \Rightarrow \text{Connect}(\text{Node}_1, z, [(\kappa_1 \ b) \ \kappa_2]_{u_1/b})
\]

which is the same as

\[
F(\Phi) \vdash \lambda b.(\kappa_1 \ b) \ \kappa_2 : \text{Connect}(\text{Node}_2, z, u_1) \Rightarrow \text{Connect}(\text{Node}_1, z, f_{\kappa_1}(u_1, C_{\kappa_3}))
\]

We write \((F(\Phi), \Rightarrow)\), to mean given axioms \( F(\Phi) \), use LP-Unif to reduce a given query. Note that for query \( A \) in \((\Phi, \Rightarrow)\), it becomes query \( A[t] \) for some \( t \) such that \( \text{FV}(A) \cap \text{FV}(t) = \emptyset \) in \((F(\Phi), \Rightarrow)\).

The following theorem shows that realizability transformation does not change the type-theoretic meaning of a program. This is important because it means we can apply different resolution strategies to resolve the query on the transformed program without worrying about the change of meaning. Later we will see that the behavior of LP-Struct is different for the original program and the transformed program.

Theorem 12. If \( \Phi \vdash e : [\forall x.\forall y. A] \Rightarrow B \) in \( C \) and \( e \) normalized to \( n \), then \( F(\Phi) \vdash F(n : [\forall x.\forall y. A] \Rightarrow B) \) in \( H \).

Proof. By induction on the derivation of \( \Phi \vdash e : [\forall x.\forall y. A] \Rightarrow B \).

- Base Case.

\[
(\kappa : \forall x.\forall y. A \Rightarrow B) \in \Phi \quad \Rightarrow \quad \Phi \vdash \kappa : \forall x.\forall y. A \Rightarrow B
\]

In this case, we know that \( F(\kappa : \forall x.\forall y. A \Rightarrow B) = \kappa : \forall x.\forall y. A_1[y_1], ..., A_n[y_n] \Rightarrow B[f_n(y_1, ..., y_n)] \in F(\Phi) \).

- Step Case.

\[
\Phi \vdash e_1 : A \Rightarrow D \\
\Phi \vdash e_2 : B, D \Rightarrow C
\]

then by the \( \Rightarrow \) rule, we have

\[
\Phi \vdash \lambda a.A(e_2 \ b) (e_1 \ a) : A, B \Rightarrow C
\]

By Lemma 3 we know that the normal form of \( e_1 \) is \( \kappa_1 \) or \( \lambda a.n_1 \), and the normal form of \( e_1 \) is \( \kappa_2 \) or \( \lambda b.d.n_2 \), with \( n_1, n_2 \) are first-order.

- \( e_1 \equiv \kappa_1, e_2 \equiv \kappa_2 \). By IH, we know that \( \Phi \vdash \kappa_1 : A_1[y_1], ..., A_n[y_n] \Rightarrow D[f_n(y_1, ..., y_n)] \) and \( \Phi \vdash \kappa_2 : B_1[z_1], ..., B_m[z_m], D[y] \Rightarrow C[f_{\kappa_2}(z_1, ..., z_m, y)] \). So by GEN and INST, we have

\[
\Phi \vdash \lambda a.\lambda b.(\kappa_2 \ l) : A_1[y_1], ..., A_n[y_n], B_1[z_1], ..., B_m[z_m] \Rightarrow C[f_{\kappa_2}(z, f_{\kappa_2}(l))] \]

Then by the \( \Rightarrow \) rule, we have

\[
\Phi \vdash \lambda a.\lambda b.\lambda c. \lbrack (\kappa_1 \ u) \rbrack : A_1[y_1], ..., A_n[y_n], B_1[z_1], ..., B_m[z_m] \Rightarrow C[f_{\kappa_2}(z, f_{\kappa_2}(l))] \]

We can see that \( \lbrack \kappa_2 \ l \rbrack_{[y/a, z/b]} = f_{\kappa_2}(z, f_{\kappa_2}(l)) \).

- \( e_1 \equiv \lambda a.n_1, e_2 \equiv \lambda b.d.n_2 \). By IH, we know that \( \Phi \vdash \lambda a.n_1 : A_1[y_1], ..., A_n[y_n] \Rightarrow D[[n_1]\lbrack y/a \rbrack] \) and \( \Phi \vdash \lambda b.d.n_2 : B_1[z_1], ..., B_m[z_m], D[y] \Rightarrow C[[n_2]_{[z/b, y/d]}] \). So by GEN and INST, we have

\[
\Phi \vdash \lambda a.\lambda b.\lambda c. \lbrack (\kappa_2 \ l) \rbrack : A_1[y_1], ..., A_n[y_n], B_1[z_1], ..., B_m[z_m] \Rightarrow C[[n_2]_{[z/b, n_1]\lbrack y/d \rbrack}] \]

Then by the \( \Rightarrow \) rule and beta reduction, we have

\[
\Phi \vdash \lambda a.\lambda b.\lambda c. \lbrack (\kappa_2 \ l) \rbrack_{[y/a, z/b]} = [n_2]_{[z/b, n_1]\lbrack y/d \rbrack} \in \text{IH} \). We know that

\[
[[n_1/d]_{[y/a, z/b]}; [n_2]_{[z/b, n_1]\lbrack y/d \rbrack}]
\]

- The other cases are handle similarly.

• Step Case.
\[ \Phi \vdash \lambda \alpha \cdot n : \forall \gamma. A \Rightarrow B \]
\[ \Phi \vdash \lambda \alpha \cdot n : \forall \gamma. A \Rightarrow \forall \gamma. B \]

By IH, we know that \( \Phi \vdash \lambda \alpha \cdot n : \forall \gamma. A[\gamma[1]] \Rightarrow B[[n]][\gamma/\alpha] \). By INST rule, we have \( \Phi \vdash \lambda \alpha \cdot n : \forall \gamma. A[\gamma[1]] \Rightarrow \forall \gamma. B[[n]][\gamma/\alpha] \).

- Step Case.

\[ \Phi \vdash e : F \]
\[ \Phi \vdash e : \forall \gamma. F \]

This case is straightforwardly by IH.

\[ \square \]

The following lemma and a theorem show that the extra argument can be used to record the term representation of the corresponding proof.

**Lemma 12.** If \( F(\Phi) \vdash \{ A[1[y_1]], ..., A[n[y_n]] \} \sim_{\gamma} 0 \), and \( y_1, ..., y_n \) are fresh, then \( F(\Phi) \vdash e_i : A[\gamma_i] \) in \( H \) with \( [e_i]_0 = \gamma y_i \) for all \( i \).

**Proof.** By induction on the length of the reduction \( F(\Phi) \vdash \{ A[1[y_1]], ..., A[n[y_n]] \} \sim_{\gamma} 0 \).

- Base Case. Suppose the length is one, namely, \( F(\Phi) \vdash \{ A[1[y]] \} \sim_{\gamma_1} 0 \). Thus there exists \( \gamma \vdash \forall \gamma. A \Rightarrow C[f(x)] \in F(\Phi) \) (here \( x \) is a constant), such that \( C[f(x)] \sim_{\gamma_1} A[\gamma] \). Thus \( \gamma_1 (C[f(x)]) \equiv \gamma_1 A[\gamma y] \). So \( \gamma_1 y \equiv f_1 \) and \( \gamma_1 C \equiv \gamma_1 A \). We have \( F(\Phi) \vdash \gamma \vdash \gamma_1 C[f(x)] \) by the INST rule, thus \( F(\Phi) \vdash \gamma \vdash \gamma_1 A[\gamma y] \), hence \( F(\Phi) \vdash \gamma : \forall \gamma. \Rightarrow \gamma_1 A[\gamma_1 y] \) by the GEN rule and \( [\gamma]_0 = f_1 \).

- Step Case. Suppose \( F(\Phi) \vdash \{ A[1[y_1]], ..., A[y_i], ..., A[n[y_n]] \} \sim_{\gamma} 0 \). 

  where \( \gamma \vdash \forall \gamma. \forall \gamma. A[1[y_1]], ..., A[n[y_n]] \Rightarrow C[f_n(x)] \in F(\Phi) \), and \( C[f_n(x)] \sim_{\gamma} A[\gamma_i] \). So we know \( \gamma_1 C[f_n(x)] \equiv \gamma_1 A[\gamma_y] \), \( \gamma_n \equiv \gamma_n \), \( C[f_n(x)] \equiv \gamma_1 A \), and \( \operatorname{dom}(\gamma_n) \cap \{ y_1, ..., y_n \} = \emptyset \). By IH, we know that \( F(\Phi) \vdash e_i : \forall \gamma. \Rightarrow \gamma_1 A[\gamma y] \), \( F(\Phi) \vdash p_i : \forall \gamma. \Rightarrow \gamma_1 B_1[\gamma y_1] \), ..., \( F(\Phi) \vdash p_m : \forall \gamma. \Rightarrow \gamma_1 B_m[\gamma y_m] \), ..., \( F(\Phi) \vdash e_m : \forall \gamma. \Rightarrow \gamma_1 A[n[y_n]] \) and \( [e_i]_0 = \gamma y_i, ..., [p_m]_0 = \gamma y_m, ..., [e_m]_0 = \gamma y_n \). We can construct a proof \( e_i : \forall \gamma. \Rightarrow \gamma_1 A[\gamma y_i] \), \( e_m : \forall \gamma. \Rightarrow \gamma_1 A[n[y_n]] \), by first apply the INST to instantiate the quantifiers of \( \gamma \), then applying the CUT rule \( m \) times. Moreover, we have \( [\gamma]_0 = f_1, ..., [p_m]_0 = \gamma y_m, ..., [e_m]_0 = \gamma y_n \).

\[ \square \]

**Theorem 13.** Suppose \( F(\Phi) \vdash \{ A[1[y]] \} \sim_{\gamma} 0 \). We have \( F(\Phi) \vdash p : \forall \gamma. \Rightarrow \gamma A[\gamma y] \) in \( H \), where \( p \) is in normal form and \( [p]_0 = \gamma y \).

Now we are able to show that realizability transformation will not change the unification reduction behaviour.

**Lemma 13.** If \( \Phi \vdash \{ A[1[y]], ..., A[n[y_n]] \} \sim^* 0 \), then \( F(\Phi) \vdash \{ A[1[y]], ..., A[n[y_n]] \} \sim^* 0 \) with \( y_i \) fresh for all \( i \).

**Proof.** By induction on the length of \( \Phi \vdash \{ A[1[y]], ..., A[n[y_n]] \} \sim^* 0 \).

- Base Case. Suppose the length is one, namely, \( \Phi \vdash \{ A[1[y]] \} \sim^* 0 \). There exists \( \gamma \vdash \forall \gamma. \Rightarrow C \in F(\Phi) \) such that \( C \sim_{\gamma} A \). Thus \( \gamma \vdash \forall \gamma. \Rightarrow C[f_n(x)] \in F(\Phi) \), and \( (C[f_n(x)]) \sim_{\gamma} A[\gamma y] \). So \( F(\Phi) \vdash \{ A[y] \} \sim_{\gamma} \forall \gamma. \Rightarrow A[y] \).

- Step Case. Suppose \( \Phi \vdash \{ A[1[y]], ..., A[y_i], ..., A[n[y_n]] \} \sim_{\gamma} 0 \).

  where \( \gamma \vdash \forall \gamma. \forall \gamma. A[1[y_1]], ..., A[n[y_n]] \Rightarrow C[f_n(x)] \in F(\Phi) \), and \( C[f_n(x)] \sim_{\gamma} A[\gamma_i] \). So we know that \( \gamma_1 C[f_n(x)] \equiv \gamma_1 A[\gamma y] \), \( \gamma_n \equiv \gamma_n \), \( C[f_n(x)] \equiv \gamma_1 A \), and \( \operatorname{dom}(\gamma_n) \cap \{ y_1, ..., y_n \} = \emptyset \). By IH, we know that \( F(\Phi) \vdash e_i : \forall \gamma. \Rightarrow \gamma_1 A[\gamma y_i] \), \( F(\Phi) \vdash p_i : \forall \gamma. \Rightarrow \gamma_1 B_1[\gamma y_1] \), ..., \( F(\Phi) \vdash p_m : \forall \gamma. \Rightarrow \gamma_1 B_m[\gamma y_m] \), ..., \( F(\Phi) \vdash e_m : \forall \gamma. \Rightarrow \gamma_1 A[n[y_n]] \) and \( [e_i]_0 = \gamma y_i, ..., [p_m]_0 = \gamma y_m, ..., [e_m]_0 = \gamma y_n \). We can construct a proof \( e_i : \forall \gamma. \Rightarrow \gamma_1 A[\gamma y_i] \), \( e_m : \forall \gamma. \Rightarrow \gamma_1 A[n[y_n]] \), by first apply the INST to instantiate the quantifiers of \( \gamma \), then applying the CUT rule \( m \) times. Moreover, we have \( [\gamma]_0 = f_1, ..., [p_m]_0 = \gamma y_m, ..., [e_m]_0 = \gamma y_n \).
Lemma 14. If \( F(\Phi) \vdash \{ A_1[y_1], \ldots, A_n[y_n] \} \leadsto^* \emptyset \) with \( y_i \) fresh for all \( i \), then \( \Phi \vdash \{ A_1, \ldots, A_n \} \leadsto^* \emptyset \).

Proof. By induction on the length of \( F(\Phi) \vdash \{ A_1[y_1], \ldots, A_n[y_n] \} \leadsto^* \emptyset \).

- Base Case. Suppose the length is one, namely, \( F(\Phi) \vdash \{ A[y] \} \leadsto^* \emptyset \).

  We know that \( (\kappa : \forall \underline{x} \rightarrow C[\{f_n\}] \in F(\Phi)) \) with \( C[\{f_n\}] \leadsto^* A[y] \). Thus \( C \leadsto^* (\gamma_1)/y A \). So \( \Phi \vdash \{ A \} \leadsto^* \emptyset \).

- Step Case. Suppose we have the following reduction:

  \[ F(\Phi) \vdash \{ A_1[y_1], \ldots, A_i[y_i], \ldots, A_n[y_n] \} \leadsto^* \gamma_1 \{ A_1'[y_1], \ldots, \gamma_1 B_1[z_1], \ldots, \gamma_1 B_m[z_m], \ldots, \gamma_1 A_n[y_n] \} \leadsto^* 0 \]

  Note that \( (\kappa : \forall \underline{x} \Rightarrow C[\{f_n\}] \in F(\Phi)) \) and \( C[\{f_n(z_1, \ldots, z_m)\}] \leadsto^* A_i[y_i] \). So we have \( \Phi \vdash \{ A_1, \ldots, A_i, \ldots, A_n \} \leadsto^* (\gamma_1 A_1, \ldots, \gamma_1 B_1, \ldots, \gamma_1 B_m, \ldots, \gamma_1 A_n) \equiv \{ \gamma_1 A_1, \ldots, \gamma_1 B_1, \ldots, \gamma_1 B_m, \ldots, \gamma_1 A_n \} \equiv \{ \gamma_1 A_1, \ldots, \gamma_1 B_1, \ldots, \gamma_1 B_m, \ldots, \gamma_1 A_n \} \leadsto^* \emptyset \).

\[ \square \]

Theorem 14. \( \Phi \vdash \{ A \} \leadsto^* \emptyset \) iff \( F(\Phi) \vdash \{ A[y] \} \leadsto^* \emptyset \).

Example 6. Consider the logic program in Example 5. Realizability transformation does not change the behaviour of LP-Unif, we still have the following successful unification reduction path for query \( \text{Connect}(x, y, u) \):

\[
F(\Phi) \vdash \{ \text{Connect}(x, y, u) \} \leadsto_{\kappa_1, \{x/x_1, y/z_1, f_1(u_3, u_4)/u\}/u} \{ \text{Connect}(x, y, u_3), \text{Connect}(y, y, u_4) \} \leadsto_{\kappa_2, \{ C_{\kappa_2}/u_3, \text{Node}/x, \text{Node}/y_1, \text{Node}/z_1, h/z_1, f_1(u_4)/u\}/u} \{ \text{Connect}(\text{Node}/x, y, u_4) \} \leadsto_{\kappa_3, \{ C_{\kappa_3}/u_4, C_{\kappa_3}/u_3, \text{Node}/x, \text{Node}/y_1, \text{Node}/z_1, \text{Node}/x_1, \text{Node}/z_1, f_1(u_4)/u\}} \emptyset
\]

There are logic programs that are overlapping and LP-TM nonterminating (as e.g. the program of Example 2), we would still like to obtain a meaningful execution behaviour for LP-Struct, especially if LP-Unif already allows successful derivations for the programs. Luckily, we can apply realizability transformation to such programs and apply LP-Struct reductions as normal.

Proposition 1. For any program \( \Phi \), \( F(\Phi) \) is LP-TM terminating and non-overlapping.

Proof. First, we need to show \( \rightarrow \)-reduction is strongly normalizing in \( (F(\Phi), \rightarrow) \). By Definition 19, we can establish a decreasing measurement (from right to left, using the strict subterm relation) for each rule in \( F(\Phi) \), since the last argument in the head of each rule is strictly larger than the ones in the body. Then, non-overlapping property is due to the fact that all the heads of the rules in \( F(\Phi) \) will be guarded by the unique function symbol in Definition 19.

\[ \square \]

Corollary 1 (Equivalence of LP-Unif and LP-Struct). \( F(\Phi) \vdash \{ A_1, \ldots, A_n \} \leadsto^* \{ B_1, \ldots, B_m \} \) iff \( F(\Phi) \vdash \{ A_1, \ldots, A_n \} \leadsto^* \{ B_1, \ldots, B_m \} \).

Proof. By Theorem 9 and Proposition 1.

Using the above corollary and soundness and completeness of LP-Unif, we deduce as a corollary that LP-Struct is sound and complete relative to system H for transformed logic programs.

Example 7. For the program in Example 5, the query \( \text{Connect}(x, y, u) \) can be reduced by LP-Struct successfully:

\[
F(\Phi) \vdash \{ \text{Connect}(x, y, u) \} \leadsto_{\kappa_1, \{ x/x_1, y/z_1, f_1(u_3, u_4)/u\}/u} \{ \text{Connect}(x, y, f_1(u_3, u_4)) \} \leadsto_{\kappa_2, \{ C_{\kappa_2}/u_3, \text{Node}/x, \text{Node}/y_1, \text{Node}/z_1, h/z_1, f_1(u_4)/u\}/u} \{ \text{Connect}(\text{Node}/x, y, u_4) \} \leadsto_{\kappa_3, \{ C_{\kappa_3}/u_4, C_{\kappa_3}/u_3, \text{Node}/x, \text{Node}/y_1, \text{Node}/z_1, \text{Node}/x_1, \text{Node}/z_1, f_1(u_4)/u\}} \emptyset
\]

Note that the answer for \( u \) is \( f_1(C_{\kappa_3}, C_{\kappa_3}) \), which is the first-order term representation of the proof of \( \Rightarrow \text{Connect}(\text{Node}/1, \text{Node}/3) \).
Realizability transformation uses the extra argument as decreasing measurement in the program to achieve termination of $\rightarrow_{\mu}$-reduction. At the same time this extra argument makes the program non-overlapping. Realizability transformation does not modify the proof-theoretic meaning and the execution behaviour of LP-Unif. The next example shows that not every transformation technique for obtaining structurally decreasing LP-TM reductions has such properties:

**Example 8.** Consider the following program:

\[
\begin{align*}
\kappa_1 : & \Rightarrow P(Int) \\
\kappa_2 : & \forall x. P(x), P(List(x)) \Rightarrow P(List(x))
\end{align*}
\]

It is a folklore method to add a structurally decreasing argument as a measurement to ensure finiteness of $\rightarrow_{\mu}$.

\[
\begin{align*}
\kappa_1 : & \Rightarrow P(Int, 0) \\
\kappa_2 : & \forall x. \forall y. P(x, y), P(List(x), y) \Rightarrow P(List(x), S(y))
\end{align*}
\]

We denote the above program as $\Phi'$. Indeed with the measurement we add, the term-matching reduction in $\Phi'$ will be finite. But the reduction for query $P(List(Int), z)$ using LP-Unif reduction will fail:

\[
\Phi' \vdash \{ P(List(Int), z) \} \nrightarrow_{\kappa_2, \{ Int/x, S(y)/z \}} \{ P(List, y), P(List(Int), y) \} \nrightarrow_{\kappa_2, \{ 0/y, Int/x, S(0)/z \}} \{ P(List(Int), 0) \} \not\Rightarrow
\]

However, the query $P(List(Int))$ on the original program using unification reduction will diverge. Divergence and failure are operationally different. Thus adding arbitrary measurement may modify the execution behaviour of a program (and hence the meaning of the program). In contrast, by Theorems 12-14 realizability transformation does not modify the execution behaviour of LP-Unif reduction.

**Example 9.** Consider the following non-LP-TM terminating and non-overlapping program and its version after the realizability transformation:

**Original program:** $\kappa : \forall x. P(x) \Rightarrow P(x)$

**After transformation:** $\kappa : \forall x. \forall u. P(x, u) \Rightarrow P(x, f_k(u))$

Both LP-Struct and LP-Unif will diverge for the queries $P(x), P(x, y)$ in both original and transformed versions. LP-Struct reduction diverges for different reasons in the two cases, one is due to divergence of $\rightarrow_{\mu}$-reduction: $\Phi \vdash \{ P(x) \} \rightarrow \{ P(x) \} \rightarrow \{ P(x) \} \rightarrow \ldots$. The other is due to $\leftrightarrow$-reduction: $\Phi \vdash \{ P(x, y) \} \leftrightarrow \{ P(x, f_k(u)) \} \leftrightarrow \{ P(x, u) \} \leftrightarrow \{ P(x, f_k(u)) \} \leftrightarrow \{ P(x, u') \} \leftrightarrow \ldots$. Note that a single step of LP-Unif reduction for the original program corresponds to infinite steps of term-matching reduction in LP-Struct. For the transformed version, a single step of LP-Unif reduction corresponds to finite steps of LP-Struct reduction.

### 6. Partial LP-Unif by Labelling

We gave a type-theoretic semantics to LP in Section 2. According to it, an answer for a given query is a substitution applied to a formula that is inhabited by a proof evidence. In that sense, the soundness lemma (Lemma 12) gives type-theoretic meaning to any LP-Unif reduction, even if it is a partial derivation, i.e. has unresolved subgoals. In this section, we build upon this result, and propose a lazy version of LP-Unif, drawing inspiration from lazy functional languages such as Haskell. In particular, we propose to label certain variables in a given query, in order to prioritise those variables for which we want to compute substitutions. Partial LP-Unif reduction will only resolve the subgoals that contain labelled variables. This requires to extend the usual unification to account for labels. We call the resulting unification algorithm labelled unification and the resulting reduction strategy – **partial LP-Unif**.

**Definition 20.** We extend the term definition:

\[
t ::= x | x^v | f(t_1, \ldots, t_n),\text{ where } x^v \text{ is a labelled variable.}
\]

Definitions of a Horn formula and a formula are extended accordingly.

A label on a variable can be informally understood as a case-expression on a variable in lazy functional language. When a query has a labelled variable, it forces resolution to compute a value for it. But since we
are in logic programming, the only way to force such evaluation is through label propagation and elimination. The following definition extends unification to achieve this.

We write \( t^v \) to denote the labelled version of \( t \), in which all the variables of \( t \) are labelled. Note that \( x^v \) is identical to \( (x^v)^v \). We write \( |t| \) to denote erasing all the labels in \( t \).

**Definition 21 (Labelled Unification).** We define \( t \simeq_\gamma t' \) (\( t \) is unifiable with \( t' \) with substitution \( \gamma \)), if \( \{ t = t' \} \not\vdash^\gamma \gamma \). The relation \( \not\vdash^\gamma \) is defined as below.

\[
\begin{align*}
\{ f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n) \} \cup E & \not\vdash t_1 = s_1, \ldots, t_n = s_n \cup E \\
\{ f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m) \} \cup E & \not\vdash \perp \\
\{ t = t \} \cup E & \not\vdash E \\
\{ f(t_1, \ldots, t_n) = x \} \cup E & \not\vdash \{ x = f(t_1, \ldots, t_n) \} \cup E \\
\{ f(t_1, \ldots, t_n) = x^v \} \cup E & \not\vdash \{ x^v = f(t_1, \ldots, t_n) \} \cup E \\
\{ x = f(t_1, \ldots, t_n) \} \cup E & \not\vdash \perp \text{ if } x \in \text{FV}(f(t_1, \ldots, t_n)) \\
\{ x^v = f(t_1, \ldots, t_n) \} \cup E & \not\vdash \perp \text{ if } x^v \in \text{FV}(f(t_1, \ldots, t_n)) \\
\{ x = t \} \cup E & \not\vdash \{ x = t \} \cup [t/x]E \text{ if } x \notin \text{FV}(t) \\
\{ x^v = t \} \cup E & \not\vdash \{ x^v = t \} \cup [t^v/x^v](\ell_\gamma(E)) \text{ if } x^v \notin \text{FV}(t)
\end{align*}
\]

We use \( \ell_\gamma(E) \) to denote a labelling operation that labels all the variables in \( E \) that occur in \( t \). Formally, \( \ell_\gamma(E) \) is defined as \( \sigma E \), where \( [x^v/x] \in \sigma \) for any \( x \in \text{FV}(t) \). The set of equations \( \{ x_1 = t_1, \ldots, x_n = t_n \} \) can be viewed as a substitution. The labelled unification of terms can be extended routinely to the unification of atomic formulas.

We write \( |\sigma| \) to denote removing all the labels in the substitution \( \sigma \), and \( L(A) \) to denote the set of labelled variables in \( A \). The following lemma shows that labelled unification is functionally equivalent to the usual (unlabelled) unification.

**Lemma 15.** If \( t \simeq_\gamma t' \) and \( \text{FV}([t]) \cap \text{FV}([t']) = \emptyset \), then \( |t| \sim_\gamma |t'| \).

We use \( \ell_\gamma(A) \) to denote another labelling operation that labels all variables in \( A \) that are labelled in the codomain of \( \gamma \). Formally, \( \ell_\gamma(A) \) is defined as \( \sigma A \), such that, for any \( x^v \in \text{FV}(\text{codom}(\gamma)) \), \( [x^v/x] \in \sigma \).

**Definition 22 (Partial LP-Unif).** We define a reduction relation on a multiset of atomic formulas: \( \Phi \vdash \{ A_1, \ldots, A_i, \ldots, A_n \} \not\vdash_{\kappa, \gamma, \gamma'} \{ \gamma A_1, \ldots, \gamma \ell_\gamma(B_1), \ldots, \gamma \ell_\gamma(B_m), \ldots, \gamma A_n \} \) for any substitution \( \gamma' \), if \( L(A_i) \neq \emptyset \) and there exists \( \kappa : \forall B_1, \ldots, B_m \Rightarrow C \in \Phi \) such that \( C \simeq_\gamma A_i \).

The labelling operation \( \ell_\gamma \) is used in the above definition to make sure that the labels are correctly propagated to \( B_1, \ldots, B_m \). As a consequence of Lemma 15, the partial LP-Unif is essentially a control strategy for LP-Unif.

**Lemma 16.** If \( \Phi \vdash \{ A_1, \ldots, A_i, \ldots, A_n \} \not\vdash_{\kappa, \gamma, \gamma'} \{ \gamma A_1, \ldots, \gamma \ell_\gamma(B_1), \ldots, \gamma \ell_\gamma(B_m), \ldots, \gamma A_n \} \), then \( \Phi \vdash \{|A_1|, \ldots, |A_i|, \ldots, |A_n|\} \not\vdash_{\kappa, |\gamma|, |\gamma'|} \{|\gamma A_1|, \ldots, |\gamma \ell_\gamma(B_1)|, \ldots, |\gamma \ell_\gamma(B_m)|, \ldots, |\gamma A_n|\} \).

Note that the above lemma implies, as a corollary, that partial LP-Unif is sound with respect to \( H \).

**Definition 23.** If \( \Phi \vdash \{ A \} \not\vdash_{\gamma} \{ B_1, \ldots, B_n \} \), \( L(A) \neq \emptyset \) and \( L(B_i) = \emptyset \) for all \( i \), then we say \( \gamma \) is the relative answer for the labelled variables in \( A \) with respect to \( B_1, \ldots, B_n \).

Comparing to LP-Unif, partial LP-Unif does not resolve subgoals without labelled variables, and therefore it terminates as soon as all the labelled formulas are resolved. From the pragmatic perspective, termination of partial LP-Unif signifies that we have obtained all the answers we need for the labelled variables, and thus no further computation is necessary. From the computational perspective, some queries may give rise to nonterminating LP-Unif reduction and the partial LP-Unif strategy offers a lazy version of LP-Unif as an
alternative. This is a useful lightweight solution, as checking (non)termination for LP can be at best only semi-decidable. From the theoretical perspective, relative answers are meaningful according to type-theoretic semantics of Section 2 via Lemma 10.

Labels on variables give us a precise way to formalize the notion of local productivity we mentioned in Introduction.

Definition 24 (Local Productivity). We say the queries $A_1, \ldots, A_n$ are locally productive at all of their labelled variables iff there exists $\Phi \vdash \{A_1, \ldots, A_n\} \rightarrow \gamma \{Q_1, \ldots, Q_l\}$ where $\gamma(x^n) = f(t_1, \ldots, t_n)$ for all $x^n \in \bigcup_i L(A_i)$ and the queries $Q_1, \ldots, Q_l$ are locally productive at all of their labelled variables.

Note that the above definition can be viewed as a form of coinductive definition (similar to the definition of Böhm trees). The requirement $\gamma(x^n) = f(t_1, \ldots, t_n)$ in the definition ensures that the answer for a labelled variable $x^n$ is at least observable at the function symbol $f$. Since the partial LP-Unif can be nonterminating, checking local productivity may result in a potentially infinite reduction.

To illustrate local productivity, let us consider the query $P(x^n)$ on the logic program $\kappa : \forall x. P(x) \Rightarrow P(f(x))$. We want to show that $P(x^n)$ is locally productive at $x^n$. We have a partial LP-Unif reduction $\Phi \vdash \{P(x^n)\} \rightarrow [f(x_1^n)/x^n] \{P(x_1^n)\}$, and then we just need to show that $P(x_1^n)$ is locally productive at $x_1^n$, which we prove by coinductive assumption. This is a very simple case of showing local productivity. In general, it is very challenging to prove local productivity in advance, and we leave this to future work.

We give two further examples of performing finite computations on infinite data structures using partial LP-Unif.

Example 10. Consider the following logic program $\Phi$, that observes, via the $Nth$ predicate, elements of an infinite stream of successive integers defined by $From$:

$$\kappa_1 : From(S(x), y) \Rightarrow From(x, Cons(x, y))$$

$$\kappa_2 : Nth(Z, Cons(x, y), x)$$

$$\kappa_3 : Nth(x, z, u) \Rightarrow Nth(S(x), Cons(y, z), u)$$

For query $Nth(S(Z), y, z^n)$, $From(S(Z), y)$, we only want to know the answer for $z^n$, i.e., the 2th element in the stream generated by $From(S(Z), y)$. We observe the following reduction:

$$\Phi \vdash \{Nth(S(Z), y, z^n), From(S(Z), y)\} \rightarrow_{\kappa_3, \gamma_1} [Z/x_1, Cons(y_1, z_1)/y, z^n/z]$$

$$\{Nth(Z, z_1, x^n), From(S(Z), Cons(y_1, z_1))\} \rightarrow_{\kappa_2, \gamma_2} [x^n/y, Cons(x^n, y_1)/z_1]$$

$$\{From(S(Z), Cons(y_1, Cons(x^n, y_2)))\} \rightarrow_{\kappa_3, \gamma_3} [Cons(x^n, y_1)/y_2, S(y_2)/z_1]$$

$$\{From(S(S(Z)), Cons(x^n, y_2))\} \rightarrow_{\kappa_1, \gamma_4} [S(S(Z))/x_1, From(S(S(Z)))/y_2]$$

Thus $S(S(Z))$ is the answer for $z^n$ relative to $From(S(S(Z)), y_4)$, i.e. the 2th element in the stream generated by $From(S(Z), y)$ is $S(S(Z))$.

Example 11. Consider the following logic program $\Phi$:

$$\kappa_1 : Take(Z, App(x, y), Nil)$$

$$\kappa_2 : Take(x, z, r) \Rightarrow Take(S(x), App(y, z), Cons(y, r))$$

$$\kappa_3 : Fib(y, App(x, y), s) \Rightarrow Fib(x, y, App(x, s))$$

The formula $Fib(y, App(x, y), s) \Rightarrow Fib(x, y, App(x, s))$ (intended to generate (potentially infinitely long) Fibonacci word. For example, $A, B, A \cdot B, B \cdot (A \cdot B), (A \cdot B) \cdot (B \cdot (A \cdot B)) \cdots$ (where “\cdot” and “\cdot\cdot” both are shorthand for $App$, all the elements are concatenation of previous two) for query $Fib(A, B, y^n)$. Now let us execute the query $Take(S(S(S(Z))))$, $y, z^n$, $Fib(A, B, y)$, i.e. taking the prefix of length 3 in a Fibonacci word:

$$\Phi \vdash \{Take(S(S(S(Z))), y, z^n), Fib(A, B, y)\} \rightarrow_{\kappa_2, \gamma_1} [S(S(S(Z))/x_1, App(y_1, z_1)/y, Cons(y_1, r_1^n)/z^n]$$

$$\{Take(S(S(Z)), z_1, r_1^n), Fib(A, B, App(y_1, z_1))\} \rightarrow_{\kappa_3, \gamma_2} [A/x_1, B/y_2, A/y_3, z_2]$$

$$\{Take(S(S(Z)), z_2, r_1^n), Fib(B, App(A, B), s_2)\} \rightarrow_{\kappa_3, \gamma_3} [S(S(Z))/x_1, App(y_2, z_2)/s_2, Cons(y_2, r_2^n)/r_1^n]$$

$$\{Take(S(Z), z_3, r_3^n), Fib(B, App(A, B), App(y_3^n, z_3))\} \rightarrow_{\kappa_3, \gamma_4} [B/x_1, App(A, B)/y_4, B/y_5, z_4/z_3]$$
\[ \{ \text{Take}(S(Z), s_4, r^3), \text{Fib}(A, B, A, B, (A, B, (A, B))), s_4 \} \xrightarrow{\gamma_2, \gamma_5} \{ \text{Take}(Z, z_5, r^3), \text{Fib}(A, B, A, B, (A, B)), y^5, z_5 \} \xrightarrow{\gamma_2, \gamma_5} \{ \text{Fib}(A, B, A, B, (A, B)), y^5, \text{App}(x_6, y_6) \} \]

We can see that \( \{ \text{Cons}(A, \text{Cons}(B, \text{Cons}(A, B)), \text{Nil})) / z^\omega \} \) is the answer relative to \( \text{Fib}(B, (A, B), A, B, (A, B), (A, B)), x_6, y_6, \) i.e. the prefix of length 3 in a Fibonacci word is indeed \( A, B, A \cdot B \).

6.1. Discussion

This section presented an experiment in applying the type-theoretic semantics formulated in the earlier sections to a practical problem of establishing a sound lazy derivation strategy for resolution. It de-emphasizes the usual notions of refutation and entailment in logic programming. Based on the type-theoretic semantics given to resolution via soundness and completeness theorems of Section 2, every reduction path of a given query is computationally meaningful. This contrasts with the traditional LP approach to declarative and operational semantics, according to which only refutations – i.e. reductions that lead to the normal form given by the empty set – are given a model theoretic meaning.

Labels we introduced in this section allowed us to annotate the intention of making an observation, and the labelled unification was formulated to hereditarily preserve this intention. Thus, we achieved a computational behavior that is similar to lazy functional programming languages, i.e. partial LP-Unif can make finite observations on the infinite data.

Related work exists on supporting lazy computation in logic programming. One is by annotating each predicate to be inductive/coinductive [22], with the intention of resolving the inductive predicate eagerly and memorizing the coinductive predicate at each step, so that one can stop the resolution whenever the current query is a variant of the previous memorized coinductive predicate. Our approach differs in that memorization and variant detection are not needed and our case.

7. Related Work

Proof Search, Logic Programming and Type Theory. To the best of our knowledge, studying logic programming proof-theoretically dates back to Girard’s suggestion to use the cut rule to model resolution for Horn formulas [5, Chapter 13.4]. Miller et. al. [19] use cut-free sequent calculus to represent a proof for a query. More specifically, given a query \( Q \) and a logic program \( P \), \( Q \) has a refutation if there is a derivation in cut-free sequent calculus for \( P \vdash Q \). Using sequent calculus as a proof theoretic framework gives the flexibility to incorporate different kinds of formulas, e.g. classical formulas and linear formulas into this framework.

Interactive theorem prover Twelf [21] pioneered implementation of proof search on top of a type system called LF [7]. Similar to Twelf, we believe that type systems serve as a suitable foundation for logic programming. Comparing to Twelf, we specify and analyse different resolution strategies (other than SLD-resolution) and study their intrinsic relations. Proof evidence in its connection with type system is also studied in Mark Jones’s thesis [11, Chapter 4.2] in the context of type class resolution. Our special attention to coinduction and various kinds of productivity is also novel compared to Twelf and type class resolution literature.

Structural Resolution. Structural resolution is a result of joint research efforts by Komendantskaya et. al. [10, 15, 16]. The goal of the analysis of structural resolution is to support sound coinductive reasoning in LP. For example, given the query \( \text{Take}(S(S(Z))), y, z \), \( \text{Fib}(A, B, y) \) in Example 11 one may want not only to obtain a substitution for \( z \), but also a guarantee that the queries to Fib are nonterminating and, moreover, that derivations for Fib will not fail if continued to infinity. Other than this coinductive soundness property, productivity analysis is being developed as a compile time technique to detect observational productivity of logic programs.

Coinductive Logic Programming. Gupta et al. [8]’s coinductive logic programming (CoLP) extends SLD-resolution with a method to construct and use atomic coinductive hypotheses. That is, during the execution, if the current queries \( \{ C_1, ..., C_i, ..., C_n \} \) contain a query \( C_i \) that unifies via \( \gamma \) with a \( C'_i \) in the earlier execution, then the next step of resolution will be given by \( \{ \gamma C_1, ..., \gamma C_{i-1}, \gamma C_{i+1}, ..., \gamma C_n \} \). The coinductive hypotheses
mechanism in CoLP can be viewed as a form of loop detection. However, CoLP cannot detect hypotheses for
more complex patterns of coinduction that produce coinductive subgoals that fail to unify; and, as discussed
in introduction, it is not a suitable tool to analyse productivity of infinite data structures in LP.

Proof Relevant Corecursive Resolution. In our previous work [4], we extended system H with fixpoint
operator to allow constructing corecursive proof evidence (given by proof terms containing fixpoint operator)
for certain nonterminating LP-TM reductions. The type system that we use to justify the corecursive proof
evidence is an extension of Howard’s system H with the fixpoint typing rule. There, the main challenge
was to heuristically construct corecursive evidence for a given query and thus to support automation of
such proofs. In general, the problem of constructing a recursive proof term for nonterminating queries is
equivalent to generation of recursive schemes and is therefore undecidable.

Logic Programming by Term Matching. LP-TM reductions may seems to be a rare kind of resolution, but
they underlie many applications. The process of simplifying type class constraints is formally described as
the notion of context reduction by Peyton Jones et. al. [12]. The context reduction process uses exactly the
LP-TM reduction that we described in this paper. The logic-based multi-paradigm programming language
PiCAT [24, 25] makes extensive use of term-matching with explicit unification. For example, the Fibonacci
sequence in PiCAT is the following:

\[
\begin{align*}
    \text{fib}(0,F) & \Rightarrow F=1. \\
    \text{fib}(1,F) & \Rightarrow F=1. \\
    \text{fib}(N,F),N>1 & \Rightarrow \text{fib}(N-1,F_1),\text{fib}(N-2,F_2),F=F_1+F_2. 
\end{align*}
\]

Through the functionalisation process, the existing termination detection techniques in term rewriting sytems
[23] can be directly applied to LP-TM. Thus we think our work in Section 4 builds a useful link between the
two paradigms, LP-TM and term rewriting.

8. Conclusions

We have shown that Howard’s system H is a suitable foundation for logic programming. We have proven
soundness and completeness of LP-Unif with respect to the type system H. We have formally defined struc-
tural resolution as LP-Struct, exhibiting that its operational semantics combines term-matching resolution
with unification. We have shown that LP-Struct is operationally equivalent to LP-Unif if the program is
LP-TM terminating and non-overlapping. Realizability transformation was suggested as an efficient method
to render all logic programs LP-TM terminating and non-overlapping. We have shown that realizability
transformation preserves the meaning of the logic program relative to H. Equivalence of LP-Struct and
LP-Unif has been shown, for transformed programs, and allowed to obtain soundness and completeness of
LP-Struct as a corollary.

We have paid a special attention to a study of LP-TM resolution. We have defined a process called
functionalisation that transforms logic programs without existential variables into term rewriting systems.
We have shown the exact relation of LP-TM and term rewriting systems, and gave an example of using
dependency pair technique from term rewriting to detect termination of LP-TM. Finally, we have developed
a partial LP-Unif resolution strategy based on labels to control LP-Unif reductions and achieve a form of
lazy computation. Based on partial LP-Unif, we have also defined a new notion of local productivity.

For future work, we would like to provide a method to establish local productivity for a given query and
study the relation between global productivity and local productivity in more detail. We plan to implement
partial LP-Unif and explore its implications.

9. Acknowledgements

We thank the anonymous referees of the earlier conference versions of this paper and LOPSTR’15 program
committee for their helpful comments and suggestions. We thank Tom Schrijvers for exposing us to type
class resolution problems and methods.
References

[1] T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(12):133 – 178, 2000.

[2] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, New York, NY, USA, 1998.

[3] H. P. Barendregt. Lambda calculi with types, handbook of logic in computer science (vol. 2): background: computational structures, 1993.

[4] P. Fu, E. Komendantskaya, T. Schrijvers, and A. Pond. Proof relevant corecursive resolution. In *Functional and Logic Programming - 15th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, pages 126–143, 2016.

[5] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, New York, NY, USA, 1989.

[6] G. Gupta, A. Bansal, R. Min, L. Simon, and A. Mallya. Coinductive logic programming and its applications. In *Logic Programming*, pages 27–44. Springer, 2007.

[7] R. Harper and D. R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4-5):613–673, 2007.

[8] W. Howard. The formulae-as-types notion of construction. In J. Seldin and J. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[9] B. Jacobs and J. Rutten. A tutorial on (co) algebras and (co) induction. *Bulletin-European Association for Theoretical Computer Science*, 62:222–259, 1997.

[10] P. Johann, E. Komendantskaya, and V. Komendantskiy. Structural resolution for logic programming. In *Technical Communications, ICLP*, 2015.

[11] M. P. Jones. *Qualified types: theory and practice*. Cambridge University Press, 2003.

[12] S. P. Jones, M. Jones, and E. Meijer. Type classes: An exploration of the design space. In *In Haskell Workshop*, 1997.

[13] S. C. Kleene. *Introduction to metamathematics*. North-Holland Publishing Company, 1952. Co-publisher: Wolters-Noordhoff, 8th revised ed.1980.

[14] E. Komendantskaya and P. Johann. Structural resolution: a framework for coinductive proof search and proof construction in horn clause logic. *ACM Transactions on Computational Logic, submitted*, 2015.

[15] E. Komendantskaya and J. Power. Coalgebraic derivations in logic programming. In *CSL*, pages 352–366, 2011.

[16] E. Komendantskaya, J. Power, and M. Schmidt. Coalgebraic logic programming: from semantics to implementation. *Journal of Logic and Computation*, 2014.

[17] R. Lämmel and S. Peyton-Jones. Scrap your boilerplate with class: Extensible generic functions. In *Proc. 10th ACM SIGPLAN Int. Conference on Functional Programming, ICFP ’05*, pages 204–215, New York, NY, USA, 2005. ACM.

[18] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.

[19] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. Technical report, 1989.

[20] U. Nilsson and J. Mahuszyński. *Logic, programming and Prolog*. Wiley Chichester, 1990.

[21] F. Pfenning and C. Schürmann. System description: Twelfa meta-logical framework for deductive systems. In *Automated DeductionCADE-16*, pages 202–206. Springer, 1999.

[22] L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *Automata, Languages and Programming*, pages 472–483. Springer, 2007.

[23] Terese. *Term rewriting systems*. Cambridge University Press, 2003.

[24] N.-F. Zhou and J. Fruhman. A users guide to picat, 2013.

[25] N.-F. Zhou, H. Kjellerstrand, and J. Fruhman. *Constraint Solving and Planning with Picat*. Springer, 2015.