A Type-Theoretic Approach to Structural Resolution

Peng Fu, Ekaterina Komendantskaya

School of Computing, University of Dundee

Abstract. Structural resolution (or S-resolution) is a newly proposed alternative to SLD-resolution that allows a systematic separation of derivations into term-matching and unification steps. Productive logic programs are those for which term-matching reduction on any query must terminate. For productive programs with coinductive meaning, finite term-rewriting reductions can be seen as measures of observation in an infinite derivation. Ability of handling corecursion in a productive way is an attractive computational feature of S-resolution.
In this paper, we make first steps towards a better conceptual understanding of operational properties of S-resolution as compared to SLD-resolution. To this aim, we propose a type system for the analysis of both SLD-resolution and S-resolution. We formulate S-resolution and SLD-resolution as reduction systems, and show their soundness relative to the type system. One of the central methods of this paper is *realizability transformation*, which makes logic programs productive and non-overlapping. We show that S-resolution and SLD-resolution are only equivalent for programs with these two properties.
Keywords: Logic Programming, Structural Resolution, Realizability Transformation, Reduction Systems, Typed lambda calculus.

1 Introduction

Logic Programming (LP) is a programming paradigm based on first-order Horn formulas. Informally, given a logic program Φ and a query A, LP provides a mechanism for automatically inferring whether or not $\Phi \vdash A$ holds, i.e., whether or not Φ logically entails A. The mechanism for the logical inference is based on SLD-resolution algorithm, which uses the resolution rule together with first-order unification.

Example 1. Consider the following logic program Φ, consisting of Horn formulas labelled by κ_1, κ_2, κ_3, defining connectivity for a graph with three nodes:

$$\begin{aligned} \kappa_1 &: \forall x.\forall y.\forall z.\mathrm{Connect}(x, y), \mathrm{Connect}(y, z) \Rightarrow \mathrm{Connect}(x, z) \\ \kappa_2 &: \Rightarrow \mathrm{Connect}(\mathrm{node}_1, \mathrm{node}_2) \\ \kappa_3 &: \Rightarrow \mathrm{Connect}(\mathrm{node}_2, \mathrm{node}_3) \end{aligned}$$

In the above program, Connect is a predicate, and $\mathrm{node}_1 - \mathrm{node}_3$ are constants. SLD-derivation for the query $\mathrm{Connect}(x, y)$ can be represented as reduction:

$$\begin{gathered}\Phi \vdash \{\mathrm{Connect}(x,y)\} \rightsquigarrow_{\kappa_1,[x/x_1,y/z_1]} \\ \{\mathrm{Connect}(x,y_1),\mathrm{Connect}(y_1,y)\} \rightsquigarrow_{\kappa_2,[\mathrm{node}_1/x,\mathrm{node}_2/y_1,\mathrm{node}_1/x_1,y/z_1]} \\ \{\mathrm{Connect}(\mathrm{node}_2,y)\} \rightsquigarrow_{\kappa_3,[\mathrm{node}_3/y,\mathrm{node}_1/x,\mathrm{node}_2/y_1,\mathrm{node}_1/x_1,\mathrm{node}_3/z_1]} \emptyset\end{gathered}$$

The first reduction $\rightsquigarrow_{\kappa_1,[x/x_1,y/z_1]}$ unifies query Connect(x, y) with the head of the rule κ_1, Connect(x_1, z_1). Note that x/x_1 means x_1 is replaced by x. After that, the query is *resolved* with the formula of κ_1, producing the next queries: Connect(x, y_1), Connect(y_1, y).

Seeing program as Horn clauses, the above derivation first assumed that Connect(x, y) is false, and then deduced a contradiction (an empty goal) from the assumption. As every SLD-derivation is essentially a proof by contradiction, traditionally, the exact content of such proofs plays little role in determining entailment. Instead, termination of derivations plays a crucial role. When it comes to logical entailment with respect to programs that admit non-terminating derivations, resolution gives only a semi-decision procedure. A long-standing challenge has been to find computationally effective mechanisms that guarantee termination of LP proof search, and to use them to deduce logical entailment for LP [3].

LP approach of preserving a tight connection between entailment and termination makes it hard to model corecursive computations. There are potentially infinite derivations that may bear some interesting computational meaning.

Example 2. The following program defines the predicate Stream:

$$\kappa_1 : \forall x.\forall y.\mathrm{Stream}(y) \Rightarrow \mathrm{Stream}(\mathrm{cons}(x,y))$$

It models infinite streams, and will result in infinite derivations, e.g.:

$$\begin{gathered}\Phi \vdash \{\mathrm{Stream}(\mathrm{cons}(x,y))\} \rightsquigarrow_{\kappa_1,[x/x_1,y/y_1]} \{\mathrm{Stream}(y)\} \rightsquigarrow_{\kappa_1,[\mathrm{cons}(x_2,y_2)/y]} \\ \{\mathrm{Stream}(y_2)\} \rightsquigarrow_{\kappa_1,[\mathrm{cons}(x_3,y_3)/y_2]} \cdots\end{gathered}$$

For the query Stream(cons(x, y)), we may still want to either obtain a description of the solution for the variable y, or make finite observation on its solution, however, none of these are supported by standard SLD-resolution.

Two groups of methods have been proposed to address this problem.

– CoLP ([5], [12]) offers methods for loop invariant analysis in SLD-derivations: infinite derivations are terminated if a loop of a certain shape is detected in resolvents, e.g., Stream(y) and Stream(y_2) above are unifiable, so one may conclude with a regular description [cons$(x_2, y)/y$].

– There are many infinite derivations that do not form a loop, CoALP/S-resolution ([10], [6]) aim to provide general coinductive gurantee that for *productive* infinte derivation, one can make finite observation. E.g. in S-resolution, the derivation for Stream(cons(x, y)) will stop at Stream(y_2) and report the process is infinite with partial answer [cons$(x_2, y_2)/y$], then one can choose to continue the derivation to further inspect y_2.

Let us view SLD-derivations as *reductions*, starting from a given query and using *unification* with Horn formulas; we call such reductions *LP-Unif reductions* and denote them by $\rightsquigarrow$. If we restrict the unification algorithm underlying such reductions to allow only term-matchers instead of unifiers, we obtain *LP-TM reductions*, denoted by $\to$. They model computations performed by rewriting trees in [6], and it has interesting properties distinguishing them from LP-Unif reductions. Firstly, they may give partial proofs compared to LP-Unif reductions:

Example 3. The following program defines bits and lists of bits:

$$\begin{gathered}\kappa_1 : \Rightarrow \mathrm{Bit}(0)\\ \kappa_2 : \Rightarrow \mathrm{Bit}(1)\\ \kappa_3 : \Rightarrow \mathrm{BList}(\mathrm{nil})\\ \kappa_4 : \forall x.\forall y.\mathrm{BList}(y), \mathrm{Bit}(x) \Rightarrow \mathrm{BList}(\mathrm{cons}(x, y))\end{gathered}$$

Below is an example of LP-TM reduction to normal form:

$$\Phi \vdash \{\mathrm{BList}(\mathrm{cons}(x, y))\} \to_{\kappa_4} \{\mathrm{Bit}(x), \mathrm{BList}(y)\}$$

Above, the head of the rule κ_4 is matched to the query by substitution $[x/x_1, y/y_1]$, which is applied to the body of the rule κ_4, thus resolving to $\{\mathrm{Bit}(x), \mathrm{BList}(y)\}$. But LP-Unif would be able to complete the proof:

$$\begin{gathered}\Phi \vdash \{\mathrm{BList}(\mathrm{cons}(x, y))\} \rightsquigarrow_{\kappa_4,[x/x_1,y/y_1]} \{\mathrm{Bit}(x), \mathrm{BList}(y)\} \rightsquigarrow_{\kappa_1,[0/x,0/x_1,y/y_1]}\\ \{\mathrm{BList}(y)\} \rightsquigarrow_{\kappa_3,[\mathrm{nil}/y,0/x,0/x_1,\mathrm{nil}/y_1]} \emptyset\end{gathered}$$

On the other hand, LP-TM reductions terminate for programs that are traditionally seen as coinductive:

Example 4. Consider LP-TM reduction for the program of Example 2:

$$\Phi \vdash \{\mathrm{Stream}(\mathrm{cons}(x, y))\} \to_{\kappa_1} \{\mathrm{Stream}(y)\}$$

Finally, LP-TM reductions are not guaranteed to terminate in general.

Example 5. For the program of Example 1, we have the following non-terminating reduction by LP-TM.

$$\begin{gathered}\Phi \vdash \{\mathrm{Connect}(x, y)\} \to_{\kappa_1} \{\mathrm{Connect}(x, y_1), \mathrm{Connect}(y_1, y)\}\\ \to_{\kappa_1} \{\mathrm{Connect}(x, y_2), \mathrm{Connect}(y_2, y_1), \mathrm{Connect}(y_1, y)\} \to_{\kappa_1} ...\end{gathered}$$

The programs that admit only finite LP-TM reductions are called productive logic programs ([10], [6]). As S-resolution combines LP-TM with unification, finiteness of LP-TM reductions allows one to observe partial answer, while the whole derivation may be infinite. Finiteness of LP-TM is also the key property to ensure this combination of LP-TM with unification is well-behaved, that is, it ensures the operational equivalence of LP-Struct and SLD resolution (we will show this later).

In Section 2, we formalise S-resolution as reduction rules that combine LP-TM reductions with substitution steps, and call the resulting reductions *LP-Struct reductions*. We see that for the program in Example 1, LP-TM reduction will necessarily diverge, while for LP-Unif there exists a finite success path. This mismatch between LP-TM and LP-Unif makes it difficult to establish the operational relation between LP-Unif and LP-Struct. As Section 4 shows, they are not operationally equivalent, in general. However, they are equivalent for programs that are productive (have finite TM-reductions) and non-overlapping (have no common instances for the Horn formula heads).

In Section 3, we introduce a technique called *realizability transformation*, that, given a program Φ, produces a program $F(\Phi)$ that is productive and non-overlapping. Realizability transformation is an interesting proof technique on its own, bearing resemblance to Kleene's [9] method under the same name. Here, it serves several purposes. 1. It helps to define a class of programs for which S-Resolution and SLD-resolution are operationally equivalent. 2. It gives means to record the proof content alongside reductions. 3. It preserves proof-theoretic meaning of the original program and computational behaviour of LP-Unif reductions.

In order to specify the proof-theoretic meaning of various LP-reductions, we introduce a type-theoretic approach to recover the notion of proof in LP. It has been noticed by Girard [4], that resolution rule $\frac{A \vee B \quad \neg B \vee D}{A \vee D}$ can be expressed by means of the cut rule in intuitionistic sequent calculus: $\frac{A \Rightarrow B \quad B \Rightarrow D}{A \Rightarrow D}$. Although the resolution rule is classically equivalent to the cut rule, the cut rule is better suited for performing computation and at the same time preserving constructive content. In Section 2 we devise a type system reflecting this intuition: if p_1 is a proof of $A \Rightarrow B$ and p_2 is a proof of $B \Rightarrow D$, then $\lambda x.p_2(p_1 x)$ is a proof of $A \Rightarrow D$. Thus, a proof can be recorded along with each cut rule. The type system we propose gives a proof theoretic interpretation for LP in general, and in particular to S-resolution. It also allows us to see clearly the proof-theoretic differences between LP-Unif/LP-Struct and LP-TM. Namely, LP-Unif/LP-Struct give proofs for Horn formulas of the form $\forall x. \Rightarrow \sigma A$, while LP-TM gives proofs for $\forall x. \Rightarrow A$. In Sections 3, the type system provides a precise tool to express the realizability transformation and prove it is a meaning-preserving transformation.

Detailed proofs for lemmas and theorems in this paper may be found in the extended version[1].

2 A Type System for LP: Horn-Formulas as Types

We first formulate a type system to model LP. We show how LP-Unif, LP-TM and LP-Struct can be defined in terms of reduction rules. We show that LP-Unif and LP-TM are sound with respect to the type system.

Definition 1.

Term t $::=$ $x \mid f(t_1, ..., t_n)$
Atomic Formula A, B, C, D $::=$ $P(t_1, ..., t_n)$
(Horn) Formula F $::=$ $[\forall \underline{x}].A_1, ..., A_n \Rightarrow A$
Proof Term p, e $::=$ $\kappa \mid a \mid \lambda a.e \mid e\ e'$
Axioms/LP Programs Φ $::= \cdot \mid \kappa : F, \Phi$

[1] Extended version is available from: `http://staff.computing.dundee.ac.uk/pengfu/document/papers/tm-lp.pdf`

Functions of arity zero are called *term constants*, $\mathrm{FV}(t)$ returns all free term variables of t. We use $\underline{A}$ to denote $A_1, ..., A_n$, when the number n is unimportant. If n is zero for $\underline{A} \Rightarrow B$, then we write $\Rightarrow B$. Note that B is an atomic formula, but $\Rightarrow B$ is a formula, we distinguish the notion of atomic formulas from (Horn) formulas. The formula $A_1, ..., A_n \Rightarrow B$ can be informally read as "the conjunction of A_i implies B". We write $\forall \underline{x}.F$ for quantifying over all the free term variables in F; $[\forall x].F$ denotes F or $\forall x.F$. LP program $B \Leftarrow \underline{A}$ are represented as $\forall \underline{x}.\underline{A} \Rightarrow B$ and query is an atomic formula. Proof terms are lambda terms, where κ denotes a proof term constant and a denotes a proof term variable.

The following is a new formulation of a type system intended to provide a type theoretic foundation for LP.

Definition 2 (Horn-Formulas-as-Types System for LP).

$$\frac{e : F}{e : \forall \underline{x}.F}\ gen \qquad \frac{e_1 : \underline{A} \Rightarrow D \quad e_2 : \underline{B}, D \Rightarrow C}{\lambda \underline{a}.\lambda \underline{b}.(e_2\ \underline{b})\ (e_1\ \underline{a}) : \underline{A}, \underline{B} \Rightarrow C}\ cut$$

$$\frac{e : \forall \underline{x}.F}{e : [\underline{t}/\underline{x}]F}\ inst \qquad \frac{(\kappa : \forall \underline{x}.F) \in \Phi}{\kappa : \forall \underline{x}.F}\ axiom$$

Note that the notion of type is identified with Horn formulas (atomic intuitionistic sequent), not atomic formulas. The usual sequent turnstile $\vdash$ is internalized as intuitionistic implication $\Rightarrow$. The rule for first order quantification $\forall$ is placed *outside* of the sequent. The cut rule is the only rule that produces new proof terms. In the *cut* rule, $\lambda \underline{a}.t$ denotes $\lambda a_1....\lambda a_n.t$ and $t\ \underline{b}$ denotes $(...(t\ b_1)...b_n)$. The size of $\underline{a}$ is the same as $\underline{A}$ and the size of $\underline{b}$ is the same as $\underline{B}$, and $\underline{a}, \underline{b}$ are not free in e_1, e_2.

Our formulation is given in the style of typed lambda calculus and sequent calculus, the intention for this formulation is to model LP type-theoretically. It has been observed the cut rule and proper axioms in intuitionistic sequent calculus can emulate LP [4](§13.4). Here we add a proof term annotation and make use of explicit quantifiers. Our formulation uses Curry-style in the sense that for the *gen* and *inst* rule, we do not modify the structure of the proof terms. Curry-style formulation allows us to focus on the proof terms generated by applying the *cut* rule.

Definition 3 (Beta-Reduction). *We define beta-reduction on proof terms as the congruence closure of the following relation:* $(\lambda a.p)p' \to_\beta [p'/a]p$

Definition 4 (Term Matching). *We define* $A \mapsto_\sigma B$, *A is matchable to B with a substitution σ and* $t \mapsto_\sigma t'$, *t is matchable to t' with a substitution σ.*

$$\frac{}{x \mapsto_{[t/x]} t} \quad \frac{\{t_i \mapsto_{\sigma_i} t_i'\}_{i \in \{1,...,n\}}}{P(t_1, ..., t_n) \mapsto_{\sigma_1 \cup ... \cup \sigma_n} P(t_1', ..., t_n')} \quad \frac{\{t_i \mapsto_{\sigma_i} t_i'\}_{i \in \{1,...,n\}}}{f(t_1, ..., t_n) \mapsto_{\sigma_1 \cup ... \cup \sigma_n} f(t_1', ..., t_n')}$$

Here $[t_1/x] \cup [t_2/x] = [t_1/x]$ if $t_1 \equiv t_2$, else, the matching process fails; and $[t_1/x] \cup [t_2/y] = [t_1/x, t_2/y]$.

Definition 5 (Unification). *We define $A \sim_\gamma B$, A is unifiable to B with substitution γ and $t \sim_\gamma t'$, t is unifiable with t' with substitution γ.*

$$\frac{}{x \sim_\emptyset x} \qquad \frac{x \notin \mathrm{FV}(t)}{x \sim_{[t/x]} t} \qquad \frac{\{\gamma t_i \sim_{\gamma_i} \gamma t'_i \quad \gamma := \gamma_{i-1} \cdot \ldots \cdot \gamma_0\}_{i \in \{1,\ldots,n\}} \quad \gamma_0 = \emptyset}{f(t_1, ..., t_n) \sim_{\gamma_n \cdot \ldots \cdot \gamma_1} f(t'_1, ..., t'_n)}$$

$$\frac{x \notin \mathrm{FV}(t)}{t \sim_{[t/x]} x} \qquad \frac{\{\gamma t_i \sim_{\gamma_i} \gamma t'_i \quad \gamma := \gamma_{i-1} \cdot \ldots \cdot \gamma_0\}_{i \in \{1,\ldots,n\}} \quad \gamma_0 = \emptyset}{P(t_1, ..., t_n) \sim_{\gamma_n \cdot \ldots \cdot \gamma_1} P(t'_1, ..., t'_n)}$$

Note that γ is updated for each i, and $\gamma \cdot \gamma'$ denotes composition of substitutions γ, γ'.

Below, we formulate different notions of reduction for LP. A similar style of formulating SLD-derivation as reduction system appeared in [11], but we identify two more kinds of reductions here: term-matching and substitutional reductions.

Definition 6 (Reductions). *We define reduction relations on the multiset of atomic formulas:*

- ***Term-matching(LP-TM) reduction:***
  $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \to_{\kappa,\gamma'} \{A_1, ..., \sigma B_1, ..., \sigma B_m, ..., A_n\}$ *for any substitution* γ'*, if there exists* $\kappa : \forall \underline{x}.B_1, ..., B_n \Rightarrow C \in \Phi$ *such that* $C \mapsto_\sigma A_i$.
- ***Unification(LP-Unif) reduction:***
  $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \rightsquigarrow_{\kappa,\gamma\cdot\gamma'} \{\gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n\}$ *for any substitution* γ'*, if there exists* $\kappa : \forall \underline{x}.B_1, ..., B_n \Rightarrow C \in \Phi$ *such that* $C \sim_\gamma A_i$.
- ***Substitutional reduction:***
  $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \hookrightarrow_{\kappa,\gamma\cdot\gamma'} \{\gamma A_1, ..., \gamma A_i, ..., \gamma A_n\}$ *for any substitution* γ'*, if there exists* $\kappa : \forall \underline{x}.B_1, ..., B_n \Rightarrow C \in \Phi$ *such that* $C \sim_\gamma A_i$.

The second subscript of term-matching reduction is used to store the substitutions obtained by unification, it is only used when we combine term-matching reductions with substitutional reductions. The second subscript in unification and substitutional reduction is intended as a state, it will be updated along with reductions. If we just talk about term-matching reduction alone, we usually use $\to$ or $\to_\kappa$. We assume implicit renaming of all quantified variables each time the above rule is applied. We write $\rightsquigarrow$ and $\hookrightarrow$ when we leave the underlining state implicit. We use $\to^*$ to denote the reflexive and transitive closure of $\to$, similarly for $\rightsquigarrow$. Notation $\rightsquigarrow^*_\gamma$ and $\to^*_\gamma$ is used when the final state along the reduction path is γ. Notice the difference between the substitutional reduction and the unification reduction. Unification reduction requires applying the substitution generated by unification to every atomic formula in the multiset. For term-matching reduction, the other atomic formulas are not affected by the computed substitution, thus term-matching reductions can be parallelised.

Given a program Φ and a set of queries $\{B_1, \ldots, B_n\}$, LP-Unif uses only unification reduction to reduce $\{B_1, \ldots, B_n\}$:

Definition 7 (LP-Unif). *Given a logic program* Φ*, LP-Unif is given by an abstract reduction system* $(\Phi, \rightsquigarrow)$.

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

Definition 8 (LP-TM). *Given a logic program Φ, LP-TM is given by an abstract reduction system $(\Phi, \to)$.*

LP-TM seems to be a foreign notion for LP, but it is used in *Context Reduction* [8] in type class instance resolution. LP-TM reductions is all we need to define productivity ([10], [6]):

Definition 9 (Productivity). *We say a program Φ is productive iff every $\to$-reduction is finite.*

Definition 10. *We use $\to^{\mu}$ to denote a reduction path to a $\to$-normal form. If the $\to$-normal form does not exist, i.e. every $\to$-reduction path is infinite, then $\to^{\mu}$ denotes an infinite reduction path. If we know that $\to$ is strongly normalizing, then we use $\to^{\nu}$ to denote a reduction path to a $\to$-normal form. We write $\hookrightarrow^1$ to denote at most one step of $\hookrightarrow$.*

Given a program Φ and a set of queries $\{B_1, \dots, B_n\}$, LP-Struct first uses term-matching reduction to reduce $\{B_1, \dots, B_n\}$ to a normal form, then performs one step substitutional reduction, and then repeats this process.

Definition 11 (LP-Struct). *Given a logic program Φ, LP-Struct is given by an abstract reduction system $(\Phi, \to^{\mu} \cdot \hookrightarrow^1)$.*

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

We first show that LP-Unif and LP-TM are sound w.r.t. the type system of Definition 2, which implies that we can obtain a proof for each successful query.

Lemma 1. *If $\Phi \vdash \{A_1, ..., A_n\} \leadsto^*_{\gamma} \emptyset$, then there exist proofs $e_1 : \forall \underline{x}. \Rightarrow \gamma A_1, ..., e_n : \forall \underline{x}. \Rightarrow \gamma A_n$, given axioms Φ.*

Proof. By induction on the length of the reduction.
Base Case. Suppose the length is one, namely, $\Phi \vdash \{A\} \leadsto_{\kappa,\gamma} \emptyset$. It implies that there exists $(\kappa : \forall \underline{x}. \Rightarrow C) \in \Phi$, such that $C \sim_{\gamma} A$. So we have $\kappa : \Rightarrow \gamma C$ by the *inst* rule. Thus $\kappa : \Rightarrow \gamma A$ by $\gamma C \equiv \gamma A$. Hence $\kappa : \forall \underline{x}. \Rightarrow \gamma A$ by the *gen* rule.
Step Case. Suppose $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \leadsto_{\kappa,\gamma} \{\gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n\}$ $\leadsto^*_{\gamma'} \emptyset$, where $\kappa : \forall \underline{x}. B_1, ..., B_m \Rightarrow C$ and $C \sim_{\gamma} A_i$. By IH, we know that there exist proofs $e_1 : \forall \underline{x}. \Rightarrow \gamma'\gamma A_1, ..., p_1 : \forall \underline{x}. \Rightarrow \gamma'\gamma B_1, ..., p_m : \forall \underline{x}. \Rightarrow \gamma'\gamma B_m, ..., e_n : \forall \underline{x}. \Rightarrow \gamma'\gamma A_n$. We can use *inst* rule to instantiate the quantifiers of κ using $\gamma' \cdot \gamma$, so we have $\kappa : \gamma'\gamma B_1, ..., \gamma'\gamma B_m \Rightarrow \gamma'\gamma C$. Since $\gamma'\gamma A_i \equiv \gamma'\gamma C$, we can construct a proof $e_i = \kappa\ p_1\ ...\ p_m$ with $e_i : \Rightarrow \gamma'\gamma A_i$, by applying the cut rule m times. By *gen*, we have $e_i : \forall \underline{x}. \Rightarrow \gamma'\gamma A_i$. The substitution generated by the unification is idempotent, and γ' is accumulated from γ, i.e. $\gamma' = \gamma'' \cdot \gamma$ for some γ'', so $\gamma'\gamma A_j \equiv \gamma''\gamma\gamma A_j \equiv \gamma''\gamma A_j \equiv \gamma' A_j$ for any j. Thus we have $e_j : \forall \underline{x}. \Rightarrow \gamma' A_j$ for any j.

Theorem 1 (Soundness of LP-Unif). *If $\Phi \vdash \{A\} \rightsquigarrow^*_\gamma \emptyset$, then there exists a proof $e : \forall \underline{x}. \Rightarrow \gamma A$ given axioms Φ.*

For example, by the soundness theorem above, the derivation in Example 1 yields a proof $(\lambda b.(\kappa_1\ b)\ \kappa_3)\ \kappa_2$ for the formula $\Rightarrow \mathrm{Connect}(\mathrm{node}_1, \mathrm{node}_3)$.

Theorem 2 (Soundness of LP-TM). *If $\Phi \vdash \{A\} \to^* \emptyset$, then there exists a proof $e : \forall \underline{x}. \Rightarrow A$ given axioms Φ.*

Observing Theorem 1 and Theorem 2, we see that for LP-TM, there is no need to accumulate substitutions, and the resulting formula is proven as stated, and does not require substitution. This difference is due to the difference of LP-TM and LP-Unif reductions. We are going to postpone the proof of soundness theorem for LP-Struct to Section 4, there we show LP-Struct and LP-Unif are operationally equivalent, which implies the soundness of LP-Struct.

3 Realizability Transformation

We define *realizability transformation* in this section. Realizability described in [9](§82) is a technique that uses a number to represent a 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 formula. More specifically, we use a first order term as an extra argument for a formula to represent a proof of that formula. Before we define the transformation, we first state several basic results about the type system in Definition 2.

Theorem 3 (Strong Normalization). *If $e : F$, then e is strongly normalizable w.r.t. beta-reduction on proof terms.*

The proof of strong normalization (SN) is an adaptation of Tait-Girard's reducibility proof. Since the first order quantification does not impact the proof term, the proof is very similar to the SN proof of simply typed lambda calculus.

Lemma 2. *If $e : [\forall \underline{x}.]\underline{A} \Rightarrow B$ given axioms Φ, then either e is a proof term constant or it is normalizable to the form $\lambda \underline{a}.n$, where n is first order normal proof term.*

Theorem 4. *If $e : [\forall \underline{x}.] \Rightarrow B$, then e is normalizable to a first order proof term.*

Lemma 2 and Theorem 4 show that we can use first order terms to represent normalized proof terms; and thus pave the way to realizability transformation.

Definition 12 (Representing First Order Proof Terms). *Let ϕ be a mapping from proof term variables to first order terms. We define a representation function $[\![\cdot]\!]_\phi$ from first order normal proof terms to first order terms.*
– $[\![a]\!]_\phi = \phi(a)$.
– $[\![\kappa\ p_1 ... p_n]\!]_\phi = f_\kappa([\![p_1]\!]_\phi, ..., [\![p_n]\!]_\phi)$*, where f_κ is a function symbol.*

Definition 13. *Let $A \equiv P(t_1, ..., t_n)$ be an atomic formula, we write $A[t']$, where $(\bigcup_i \mathrm{FV}(t_i)) \cap \mathrm{FV}(t') = \emptyset$, to abbreviate a new atomic formula $P(t_1, ..., t_n, t')$.*

Definition 14 (Realizability Transformation). *We define a transformation F on formula and its normalized proof term:*

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

The realizability transformation systematically associates a proof to each atomic formula, so that the proof can be recorded along with reductions.

Example 6. The following logic program is the result of applying realizability transformation on the program in Example 1.

$$\begin{aligned}\kappa_1 : \forall x.\forall y.\forall u_1.\forall u_2.\mathrm{Connect}(x, y, u_1), \mathrm{Connect}(y, z, u_2) &\Rightarrow \mathrm{Connect}(x, z, f_{\kappa_1}(u_1, u_2))\\ \kappa_2 : &\Rightarrow \mathrm{Connect}(\mathrm{node}_1, \mathrm{node}_2, c_{\kappa_2})\\ \kappa_3 : &\Rightarrow \mathrm{Connect}(\mathrm{node}_2, \mathrm{node}_3, c_{\kappa_3})\end{aligned}$$

Before the realizability transformation, we have the following judgement:

$$\lambda b.(\kappa_1\ b)\ \kappa_2 : \mathrm{Connect}(\mathrm{node}_2, z) \Rightarrow \mathrm{Connect}(\mathrm{node}_1, z)$$

We can apply the transformation, we get:

$$\lambda b.(\kappa_1\ b)\ \kappa_2 : \mathrm{Connect}(\mathrm{node}_2, z, u_1) \Rightarrow \mathrm{Connect}(\mathrm{node}_1, z, [\![(\kappa_1\ b)\ \kappa_2]\!]_{[u_1/b]})$$

which is the same as

$$\lambda b.(\kappa_1\ b)\ \kappa_2 : \mathrm{Connect}(\mathrm{node}_2, z, u_1) \Rightarrow \mathrm{Connect}(\mathrm{node}_1, z, f_{\kappa_1}(u_1, c_{\kappa_2}))$$

Observe that the transformed formula:
$\mathrm{Connect}(\mathrm{node}_2, z, u_1) \Rightarrow \mathrm{Connect}(\mathrm{node}_1, z, f_{\kappa_1}(u_1, c_{\kappa_2}))$ is provable by $\lambda b.(\kappa_1\ b)\ \kappa_2$ using the transformed program.

Let $F(\Phi)$ mean applying the realizability transformation to every axiom in Φ. We write $(F(\Phi), \rightsquigarrow), (F(\Phi), \to), (F(\Phi), \to^{\mu} \cdot \hookrightarrow^{1})$, to mean given axioms $F(\Phi)$, use LP-Unif, LP-TM, LP-Struct respectively to reduce a given query. Note that for query A in $(\Phi, \rightsquigarrow), (\Phi, \to), (\Phi, \to^{\mu} \cdot \hookrightarrow^{1})$, it becomes query $A[t]$ for some t such that $\mathrm{FV}(A) \cap \mathrm{FV}(t) = \emptyset$ in $(F(\Phi), \rightsquigarrow), (F(\Phi), \to), (F(\Phi), \to^{\mu} \cdot \hookrightarrow^{1})$.

The next Theorem establishes that, for any program Φ, LP-TM reductions for $F(\Phi)$ are strongly normalizing.

Theorem 5. *For any $(\Phi, \to^{\mu} \cdot \hookrightarrow^{1})$, we have $(F(\Phi), \to^{\nu} \cdot \hookrightarrow^{1})$.*

Proof. We just need to show $\to$-reduction is strongly normalizing in $(F(\Phi), \to)$. By Definition 12 and 14, we can establish a decreasing measurement(from right to left) 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.

The above theorem shows that we can use realizability transformation to obtain productive logic programs, moreover, this transformation is general, meaning that any logic program can be transformed to an equivalent productive one. The following theorem shows that realizability transformation does not change the proof-theoretic meaning of a program.

Theorem 6. *Given axioms Φ, if $e : [\forall \underline{x}].\underline{A} \Rightarrow B$ holds with e in normal form, then $F(e : [\forall \underline{x}].\underline{A} \Rightarrow B)$ holds for axioms $F(\Phi)$.*

The other direction for the theorem above is not true if we ignore the transformation F, namely, if $e : \forall \underline{x}. \Rightarrow A[t]$ for axioms Φ, it may not be the case that $e : \forall \underline{x}. \Rightarrow A$, since the axioms Φ may not be set up in a way such that t is a representation of proof e. The following theorem shows that the extra argument is used to record the term representation of the corresponding proof.

Theorem 7. *Suppose $F(\Phi) \vdash \{A[y]\} \rightsquigarrow^*_{\gamma} \emptyset$. We have $p : \forall \underline{x}. \Rightarrow \gamma A[\gamma y]$ for $F(\Phi)$, where p is in normal form and $[\![p]\!]_{\emptyset} = \gamma y$.*

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

Lemma 3. *$\Phi \vdash \{A_1, ..., A_n\} \rightsquigarrow^* \emptyset$ iff $F(\Phi) \vdash \{A_1[y_1], ..., A_n[y_n]\} \rightsquigarrow^* \emptyset$.*

Proof. For each direction, by induction on the length of the reduction. Each proof will be similar to the proof of Lemma 1, see the extended version for the details.

Theorem 8. *$\Phi \vdash \{A\} \rightsquigarrow^* \emptyset$ iff $F(\Phi) \vdash \{A[y]\} \rightsquigarrow^* \emptyset$.*

Example 7. Consider the logic program after realizability transformation in Example 6. Realizability transformation does not change the behaviour of LP-Unif, we still have the following successful unification reduction path for query $\mathrm{Connect}(x, y, u)$:

$$F(\Phi) \vdash \{\mathrm{Connect}(x, y, u)\} \rightsquigarrow_{\kappa_1, [x/x_1, y/z_1, f_{\kappa_1}(u_3, u_4)/u]}$$
$$\{\mathrm{Connect}(x, y_1, u_3), \mathrm{Connect}(y_1, y, u_4)\}$$
$$\rightsquigarrow_{\kappa_2, [c_{\kappa_2}/u_3, \mathrm{node}_1/x, \mathrm{node}_2/y_1, \mathrm{node}_1/x_1, b/z_1, f_{\kappa_1}(c_{\kappa_2}, u_4)/u]}$$
$$\{\mathrm{Connect}(\mathrm{node}_2, y, u_4)\}$$
$$\rightsquigarrow_{\kappa_3, [c_{\kappa_3}/u_4, c_{\kappa_2}/u_3, \mathrm{node}_3/y, \mathrm{node}_1/x, \mathrm{node}_2/y_1, \mathrm{node}_1/x_1, \mathrm{node}_3/z_1, f_{\kappa_1}(c_{\kappa_2}, c_{\kappa_3})/u]} \emptyset$$

The realizability transformation uses the extra argument as decreasing measurement to the program to achieve the termination of $\to$-reduction. We want to point out that realizability transformation does not modify the proof-theoretic meaning and the execution behaviour. The next example shows that not every transformation technique for obtaining productive programs have such properties:

Example 8. Consider the following program:

$$\kappa_1 : \Rightarrow P(\mathrm{int})$$
$$\kappa_2 : \forall x.P(x), P(\mathrm{list}(x)) \Rightarrow P(\mathrm{list}(x))$$

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

$$\kappa_1 : \Rightarrow P(\mathrm{int}, 0)$$
$$\kappa_2 : \forall x.\forall y.P(x, y), P(\mathrm{list}(x), y) \Rightarrow P(\mathrm{list}(x), \mathrm{s}(y))$$

We denote the above program as Φ'. Indeed with the measurement we add, the term-matching reduction in Φ' will be finite. But the reduction for query $P(\mathrm{list}(\mathrm{int}), z)$ using unification will fail:

$$\Phi' \vdash \{P(\mathrm{list}(\mathrm{int}), z)\} \rightsquigarrow_{\kappa_2,[\mathrm{int}/x,\mathrm{s}(y_1)/z]}$$
$$\{P(\mathrm{int}, y_1), P(\mathrm{list}(\mathrm{int}), y_1)\} \rightsquigarrow_{\kappa_2,[0/y_1,\mathrm{int}/x,\mathrm{s}(0)/z]} \{P(\mathrm{list}(\mathrm{int}), 0)\} \not\rightsquigarrow$$

However, the query $P(\mathrm{list}(\mathrm{int}))$ 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), but by Theorems 6-8, realizability transformation does not modify the execution behaviour of unification reduction.

4 Operational Equivalence of LP-Struct and LP-Unif

Since realizability transformation does not change the proof theoretic meaning of the program or modify the behaviour of unification reduction, we will work directly on $F(\Phi)$ in this section. We will show that LP-Struct and LP-Unif are equivalent after the realizability transformation. By Theorem 5, it suffices to consider $(F(\Phi), \to^{\nu} \cdot \hookrightarrow^{1})$ for LP-Struct.

The following lemma shows that each LP-Unif reduction can be emulated by one step of substitutional reduction followed by one step of term-matching reduction.

Lemma 4. *If $F(\Phi) \vdash \{A_1, ..., A_i, ..., A_n\} \rightsquigarrow_{\gamma} \{\gamma A_1, ..., \gamma \underline{B}, ..., \gamma A_n\}$ for $\kappa : \forall \underline{x}.\underline{B} \Rightarrow C \in F(\Phi)$ such that $C \sim_{\gamma} A_i$, then $F(\Phi) \vdash \{A_1, ..., A_i, ..., A_n\} \hookrightarrow_{\kappa,\gamma} \{\gamma A_1, ..., \gamma A_i, ..., \gamma A_n\} \to_{\kappa} \{\gamma A_1, ..., \gamma \underline{B}, ..., \gamma A_n\}$.*

The following lemma shows that for $\to$-normal form, each $\hookrightarrow \cdot \to$ step is equivalent to a step of $\rightsquigarrow$ reduction.

Lemma 5. *Let $\{A_1[x_1], ..., A_n[x_n]\}$ be a multiset of atomic formulas in $\to$-normal form, and suppose there exists*
$\kappa : \forall \underline{x}.\underline{y}.B_1[y_1], ..., B_m[y_m] \Rightarrow C[f_{\kappa}(y_1, ..., y_m)] \in F(\Phi)$ such that
$C[f_{\kappa}(y_1, ..., y_m)] \sim_{\gamma} A_i[x_i]$. Then we have the following:

1. *$F(\Phi) \vdash \{A_1[x_1], ..., A_i[x_i], ..., A_n[x_n]\} \hookrightarrow_{\kappa,\gamma} \{\gamma A_1[x_1], ..., \gamma A_i[\gamma x_i], ..., \gamma A_n[x_n]\}$*
   $\to_{\kappa} \{\gamma A_1, ..., \gamma B_1[y_1], ..., \gamma B_m[y_m], ..., \gamma A_n[y_n]\}$,
   with $\{\gamma A_1[x_1], ..., \gamma B_1[y_1], ..., \gamma B_m[y_m], ..., \gamma A_n[x_n]\}$ in $\to$-normal form.

2. $F(\Phi) \vdash \{A_1[x_1], ..., A_i[x_i], ..., A_n[x_n]\} \leadsto_{\kappa,\gamma}$
$\{\gamma A_1[x_1], ..., \gamma B_1[y_1], ..., \gamma B_m[y_m], ..., \gamma A_n[x_n]\}$.

Proof. We only prove *1.* here. We know $\{y_1, ..., y_m, x_1, ..., x_{i-1}, x_{i+1}, .., x_n\} \cap \mathrm{dom}(\gamma) = \emptyset$, $x_i \in \mathrm{dom}(\gamma)$, and every head in $F(\Phi)$ is of the form $D[f(z)]$, so $\{\gamma A_1[x_1], ..., \gamma B_1[y_1], ..., \gamma B_m[y_m], ..., \gamma A_n[x_n]\}$ is in $\rightarrow$-normal form.

Theorem 9 (Equivalence of LP-Struct and LP-Unif). *$F(\Phi) \vdash \{A[y]\} \leadsto^* \emptyset$ iff $F(\Phi) \vdash \{A[y]\}(\rightarrow^\nu \cdot \hookrightarrow^1)^*\emptyset$.*

Proof. From left to right, by Lemma 4 and Lemma 5(1), we know that each $\leadsto$ step can be simulated by $\hookrightarrow \cdot \rightarrow$. From right to left, by Lemma 5(1), we know that the concrete shape of $F(\Phi) \vdash \{A[y]\}(\rightarrow^\nu \cdot \hookrightarrow^1)^*\emptyset$ must be of the form $F(\Phi) \vdash \{A[y]\}(\hookrightarrow \cdot \rightarrow)^*\emptyset$, then by Lemma 5(2), we have $F(\Phi) \vdash \{A[y]\} \leadsto^* \emptyset$.

Example 9. For the program in Example 6, the query $\mathrm{Connect}(x, y, u)$ can be reduced by LP-Struct successfully:

$$F(\Phi) \vdash \{\mathrm{Connect}(x,y,u)\} \hookrightarrow_{\kappa_1,[x/x_1,y/z_1,f_{\kappa_1}(u_3,u_4)/u]}$$
$$\{\mathrm{Connect}(x,y,f_{\kappa_1}(u_3,u_4))\} \rightarrow_{\kappa_1} \{\mathrm{Connect}(x,y_1,u_3), \mathrm{Connect}(y_1,y,u_4)\}$$
$$\hookrightarrow_{\kappa_2,[c_{\kappa_2}/u_3,\mathrm{node}_1/x,\mathrm{node}_2/y_1,\mathrm{node}_1/x_1,b/z_1,f_{\kappa_1}(c_{\kappa_2},u_4)/u]}$$
$$\{\mathrm{Connect}(\mathrm{node}_1,\mathrm{node}_2,c_{\kappa_2}), \mathrm{Connect}(\mathrm{node}_2,y,u_4)\} \rightarrow_{\kappa_2} \{\mathrm{Connect}(\mathrm{node}_2,y,u_4)\}$$
$$\hookrightarrow_{\kappa_3,[c_{\kappa_3}/u_4,c_{\kappa_2}/u_3,\mathrm{node}_3/y,\mathrm{node}_1/x,\mathrm{node}_2/y_1,\mathrm{node}_1/x_1,\mathrm{node}_3/z_1,f_{\kappa_1}(c_{\kappa_2},c_{\kappa_3})/u]}$$
$$\{\mathrm{Connect}(\mathrm{node}_2,\mathrm{node}_3,c_{\kappa_3})\} \rightarrow_{\kappa_3} \emptyset$$

Note that the answer for u is $f_{\kappa_1}(c_{\kappa_2}, c_{\kappa_3})$, which is the first order term representation of the proof of $\Rightarrow \mathrm{Connect}(\mathrm{node}_1, \mathrm{node}_3)$.

After the realizability transformation, LP-Struct is equivalent to LP-Unif in the sense of Theorem 9. As a consequence, we have the soundness theorem for LP-Struct w.r.t. the type system in Definition 2.

Corollary 1 (Soundness of LP-Struct). *If $F(\Phi) \vdash \{A[y]\}(\rightarrow^\nu \cdot \hookrightarrow^1)^*_\gamma\emptyset$, then there exist $e : \forall \underline{x}. \Rightarrow \gamma(A[y])$ for $F(\Phi)$.*

We have seen that without realizability transformation, LP-Struct is not operationally equivalent to LP-Unif by Example 1. Example 9 shows that after realizability transformation, we do get operational equivalence of LP-Struct and LP-Unif. The mismatch of LP-Unif and LP-Struct seems to be due to the infinity of the $\rightarrow$-reduction. One may wonder whether it is the case that for any productive program, LP-Struct and LP-Unif are operationally equivalent. The following example shows that it is not the case in general.

Example 10.

$$\kappa_1 : \Rightarrow P(\mathrm{c})$$
$$\kappa_2 : \forall x. Q(x) \Rightarrow P(x)$$

Here c is a constant. The program is $\rightarrow$-terminating. However, for query $P(x)$, we have $\Phi \vdash \{P(x)\} \leadsto_{\kappa_1,[\mathrm{c}/x]} \emptyset$ with LP-Unif, but $\Phi \vdash \{P(x)\} \rightarrow_{\kappa_2} \{Q(x)\} \not\rightarrow$ for LP-Struct.

So termination of $\rightarrow$-reduction is insufficient for establishing the relation between LP-Struct and LP-Unif. In Example 10, the problem is caused by the overlapping heads $P(\mathrm{c})$ and $P(x)$. Motivated by the notion of non-overlapping in term rewrite system ([2], [1]), we have the following definition.

Definition 15 (Non-overlapping Condition). *Axioms Φ are non-overlapping if for any two formulas $\forall \underline{x}.\underline{B} \Rightarrow C, \forall \underline{x}.\underline{D} \Rightarrow E \in \Phi$, there are no substitution σ, δ such that $\sigma C \equiv \delta C'$.*

The theorem below shows that for any non-overlapping program, terminating reductions in LP-Struct are operationally equivalent to terminating reductions in LP-Unif. However, without productivity, LP-Struct and LP-Unif are no longer operationally equivalent for the diverging program.

Theorem 10. *Suppose Φ is non-overlapping. $\Phi \vdash \{A_1, ..., A_n\} \rightsquigarrow^*_\gamma \{C_1, ..., C_m\}$ with $\{C_1, ..., C_m\}$ in $\rightsquigarrow$-normal form iff $\Phi \vdash \{A_1, ..., A_n\}(\rightarrow^\mu \cdot \hookrightarrow^1)^*_\gamma\{C_1, ..., C_m\}$ with $\{C_1, ..., C_m\}$ in $\rightarrow^\mu \cdot \hookrightarrow^1$-normal form.*

Example 11. Consider the following non-productive and non-overlapping program and its version after the realizability transformation:

$$\textit{Original program: } \kappa : \forall x.P(x) \Rightarrow P(x)$$
$$\textit{After transformation: } \kappa : \forall x.\forall u.P(x, u) \Rightarrow P(x, f_\kappa(u))$$

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 $\rightarrow$-reduction:
$\Phi \vdash \{P(x)\} \rightarrow \{P(x)\} \rightarrow \{P(x)\}...$
The another is due to $\hookrightarrow$-reduction:
$\Phi \vdash \{P(x, y)\} \hookrightarrow \{P(x, f_k(u))\} \rightarrow \{P(x, u)\} \hookrightarrow \{P(x, f_k(u'))\} \rightarrow \{P(x, u')\}...$

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.

The next theorem shows that we need both productivity and non-overlapping to establish operational equivalence of LP-Struct and LP-Unif for both finite and infinite reductions. Note that realizability transformation guarantees exactly these two properties.

Theorem 11. *Suppose Φ is non-overlapping and productive.*

1. *If $\Phi \vdash \{A_1, ..., A_n\} \rightsquigarrow \{B_1, ..., B_m\}$, then $\Phi \vdash \{A_1, ..., A_n\}(\rightarrow^\nu \cdot \hookrightarrow^1)^*\{C_1, ..., C_l\}$ and $\Phi \vdash \{B_1, ..., B_m\} \rightarrow^* \{C_1, ..., C_l\}$.*
2. *If $\Phi \vdash \{A_1, ..., A_n\}(\rightarrow^\nu \cdot \hookrightarrow^1)^*\{B_1, ..., B_m\}$, then $\Phi \vdash \{A_1, ..., A_n\} \rightsquigarrow^* \{B_1, ..., B_m\}$.*

For the diverging but productive programs (like Stream of Example 2), productivity gives opportunity to make finite observations for potentially infinite derivations [6], and allows us not to eargerly unfold the infinite derivation.

5 Conclusions and Future Work

We proposed a type system that gives a proof theoretic interpretation for LP, where Horn formulas correspond to the notion of type, and a successful query yields a first order proof term. The type system also provided us with a precise tool to show that realizability transformation preserves both proof-theoretic meaning of the program and the execution behaviour of the unification reduction.

We formulated S-resolution as LP-Struct reduction, which can be seen as a reduction strategy that combines term-matching reduction with substitutional reduction. This formulation allowed us to study the operational relation between LP-Struct and LP-Unif. The operational equivalence of LP-Struct and LP-Unif is by no means obvious. Previous work ([6], [10]) only gives soundness and completeness of LP-Struct with respect to the Herbrand model. We identified that productivity and non-overlapping are essential for showing their operational equivalence. Therefore, these two properties identify the "structural" fragment of logic programs.

Realizability transformation proposed here ensures that the resulting programs are productive and non-overlapping. It preserves the proof-theoretic meaning of the program, in a formally defined sense of Theorems 6-8. It serves as a proof-method that enables us to show the operational equivalence of LP-Unif and LP-Struct, for productive and non-overlapping programs. It is general, applies to any logic program, and can be easily mechanised. Finally, it allows to automatically record the proof content in the course of reductions, as Theorem 7 establishes.

With the proof system for LP-reductions we proposed, we are planning to further investigate the interaction of LP-TM/Unif/Struct with typed functional languages. We expect to find a tight connection between our work and the type class inference, cf. ([13,7]). Using terminology of this paper, a type class corresponds to an atomic formula, an instance declaration corresponds to a Horn formula, and the instance resolution process in type class inference uses LP-TM reductions, in which evidence for the type class corresponds to our notion of proof. Realizability transformation then gives a method to record the proof automatically. A careful examination of these connections is warranted.

If one works only with Horn-formulas in LP, then we know that the proof of a successful query can be normalized to a first order proof term. It seems that nothing interesting can happen to the proof term. But when we plug the proof system into a typed functional language in the form of a type class and instance declaration, the proof will correspond to the evidence for the type class, and it will interact with the underlining functional program, and eventually will be run as a program. For example, the following declaration specifies a way to construct equality class instance for datatype list and int:

$$\begin{array}{l} \kappa_1 : \ \mathrm{Eq}(x) \Rightarrow \mathrm{Eq}(\mathrm{list}(x)) \\ \kappa_2 : \ \Rightarrow \mathrm{Eq}(\mathrm{int}) \end{array}$$

Here list is a function symbol, int is a constant and x is variable; κ_1, κ_2 will be defined as functional programs that are used to construct the evidence. When the underlining functional system makes a query Eq(list(int)), we can use LP-TM to construct a proof for Eq(list(int)), which is $\kappa_2\ \kappa_1$, and then $\kappa_2\ \kappa_1$ will serve as runtime evidence for the corresponding method, thus yielding computational meaning of the proof.

References

1. Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, New York, NY, USA, 1998.
2. Marc Bezem, Jan Willem Klop, and Roel de Vrijer. *Term rewriting systems.* Cambridge University Press, 2003.
3. Danny De Schreye and Stefaan Decorte. Termination of logic programs: The never-ending story. *The Journal of Logic Programming*, 19:199–260, 1994.
4. Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types.* Cambridge University Press, New York, NY, USA, 1989.
5. Gopal Gupta, Ajay Bansal, Richard Min, Luke Simon, and Ajay Mallya. Coinductive logic programming and its applications. In *Logic Programming*, pages 27–44. Springer, 2007.
6. Patricia Johann, Ekaterina Komendantskaya, and Vladimir Komendantskiy. Structural resolution for logic programming. In *Technical Communications of ICLP*, 2015.
7. Mark P Jones. *Qualified types: theory and practice*, volume 9. Cambridge University Press, 2003.
8. Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: An exploration of the design space. In *In Haskell Workshop*, 1997.
9. Stephen Cole Kleene. *Introduction to metamathematics.* North-Holland Publishing Company, 1952. Co-publisher: Wolters–Noordhoff; 8th revised ed.1980.
10. Ekaterina Komendantskaya, John Power, and Martin Schmidt. Coalgebraic logic programming: from semantics to implementation. *Journal of Logic and Computation*, page exu026, 2014.
11. Ulf Nilsson and Jan Małuszyński. *Logic, programming and Prolog.* Wiley Chichester, 1990.
12. Luke Simon, Ajay Bansal, Ajay Mallya, and Gopal Gupta. Co-logic programming: Extending logic programming with coinduction. In *Automata, Languages and Programming*, pages 472–483. Springer, 2007.
13. Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.