A Type-theoretic Approach to Resolution [⋆]

Peng Fu, Ekaterina Komendantskaya

Computer Science, University of Dundee

Abstract. We propose a new type-theoretic approach to SLD-resolution and Horn-clause logic programming. It views Horn formulas as types, and derivations for a given query as a construction of the inhabitant (a proof-term) for the type given by the query. We propose a method of program transformation that allows to transform logic programs in such a way that proof evidence is computed alongside SLD-derivations. We discuss two applications of this approach: in recently proposed productivity theory of structural resolution, and in type class inference.
Keywords: Logic Programming, Typed lambda calculus, Realizability Transformation, Reduction Systems, Structural Resolution.

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 inference mechanism is based on the SLD-resolution, 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:

$$\kappa_1 : \forall x.\forall y.\forall z.\mathrm{Connect}(x, y), \mathrm{Connect}(y, z) \Rightarrow \mathrm{Connect}(x, z)$$
$$\kappa_2 : \Rightarrow \mathrm{Connect}(\mathrm{Node}_1, \mathrm{Node}_2)$$
$$\kappa_3 : \Rightarrow \mathrm{Connect}(\mathrm{Node}_2, \mathrm{Node}_3)$$

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 the following reduction:

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

The first reduction $\rightsquigarrow_{\kappa_1,[x/x_1,y/z_1]}$ unifies the query $\mathrm{Connect}(x, y)$ with the head of the rule κ_1 (which is $\mathrm{Connect}(x_1, z_1)$ 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 κ_1, producing the next queries: $\mathrm{Connect}(x, y_1)$, $\mathrm{Connect}(y_1, y)$. Note that the substitution in the subscript of $\rightsquigarrow$ is a state that will be updated alongside the derivation.

[⋆] This work was funded by EPSRC grant EP/K031864/1.

Viewing a program as a collection of Horn clauses, the above derivation first assumed that $\mathrm{Connect}(x, y)$ 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. However, it is desirable to have methods which capture the proof-theoretic content of SLD-derivations. For example, one may wish to reason in a proof-relevant way, and compute not just $\Phi \vdash A$, but $\Phi \vdash p : A$, where p is the proof-witness for the query A and the program Φ. LP and its dialects are used as part of type inference engines underlying functional [11,6] and dependently typed [4] languages. These applications require proof-relevant automated reasoning.

In type class inference (e.g. Haskell), a type class can be seen as an atomic formula and an instance declaration – as a Horn formula. The instance resolution process in type class inference can then be seen as an SLD-derivation, with one additional requirement: this SLD-derivation must compute the evidence for the type class (or construct a dictionary). For example, the following declaration specifies a way to construct equality class instances for datatypes List and Char:

$$\begin{array}{ll} \kappa_1: & \forall x.\mathrm{Eq}(x) \Rightarrow \mathrm{Eq}(\mathrm{List}(x)) \\ \kappa_2: & \Rightarrow \mathrm{Eq}(\mathrm{Char}) \end{array}$$

Here List is a function symbol, Char is a constant and x is a variable; κ_1, κ_2 will be used as primitives for the evidence construction. When we make a comparison of two lists of characters, such as (eq $['a']$ $['b']$), the compiler will insert the evidence d of the type Eq(List(Char)) in (eq d $['a']$ $['b']$). The construction of this evidence can be viewed as resolving the query Eq(List(Char)), which is witnessed by applying Horn formulas κ_1 and κ_2. Thus, $(\kappa_1\ \kappa_2)$ is the evidence we want for d.

In order to specify the proof-theoretic meaning of derivations, we introduce a type-theoretic approach to recover the notion of proof in LP. It has been noticed by Girard [3], that the 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 while preserving constructive content. In Section 2 we present a type system reflecting this intuition: if p_1 is the proof of $A \Rightarrow B$ and p_2 is the proof of $B \Rightarrow D$, then $\lambda x.p_2\ (p_1\ x)$ is the proof of $A \Rightarrow D$. Thus, proof can be recorded alongside with each cut rule.

We prove that SLD-resolution is sound with respect to the type system (Section 2). We give a formulation of SLD-resolution in the form of a reduction rule, called *LP-Unif*. The soundness result shows that, given a logic program Φ and a query A, if A can be LP-Unif reduced to the empty goal with a substitution γ as an answer, then a proof term can be constructed for γA.

In Section 3, we introduce a technique called *realizability transformation*, that, given a program Φ, produces a program $F(\Phi)$ in which one extra argument is added to every predicate, in order to record the proof-evidence in derivations. The proof evidence is computed by applying substitution to variables held by this additional argument in the course of running SLD-resolution. Let us revisit the List example. Its transformed version will look as follows:

$$\begin{array}{l}\kappa_1 : \ \forall x.\forall u_1.\mathrm{Eq}(x, u_1) \Rightarrow \mathrm{Eq}(\mathrm{List}(x), f_{\kappa_1}(u_1)) \\ \kappa_2 : \ \Rightarrow \mathrm{Eq}(\mathrm{Char}, c_{\kappa_2})\end{array}$$

The query $\mathrm{Eq}(\mathrm{List}(\mathrm{Char}))$ of the original program becomes $\mathrm{Eq}(\mathrm{List}(\mathrm{Char}), u)$ after the transformation, where u is a variable. The derivation reaches the empty goal and outputs the substitution $[f_{\kappa_1}(c_{\kappa_2})/u]$, which corresponds to the proof term $(\kappa_1\ \kappa_2)$ for the query $\mathrm{Eq}(\mathrm{List}(\mathrm{Char}))$.

Realizability transformation bears resemblance to Kleene's [7] method under the same name. We show that realizability transformation preserves the proof-theoretic meaning of the original program and the computational behaviour of LP-Unif reductions. With the help of the transformation, we prove completeness of LP-Unif with repect to the type system.

Together, Sections 2 and 3 introduce a method of constructing proof evidence in the process of LP derivations. Recently, a variant of resolution for Horn Clauses, called *structural resolution* (*S-resolution*) has been introduced [5]. S-resolution represents derivations by SLD-resolution as a combination of derivations by term-matching and by substitution. We explain this idea in detail in Section 4. The main reason for separating out two components of SLD-resolution in such a way is to make use of structural properties of term-matching that have already been exploited in functional programming and term-rewriting. In particular, S-resolution allowed to define a theory of universal productivity for LP that resembles a similar theory in functional programming[2]: given a potentially infinite derivation by S-resolution, termination of term-matching derivations that comprise it determines *productivity* of the derivation (or in other words, observability of finite fragments of the infinite computation).

We conjecture that the combination of the two ideas – the theory of productivity introduced by S-resolution and the proof-witness construction introduced in this paper bear promise for future development of resolution-based methods. This is why, in Section 4 we give a full formal study of how these two methods can be formally combined. We show how S-resolution can be represented by means of *LP-Struct reductions*, combining term-matching reductions and unification. We extend the type-theoretic semantics to S-Resolution. We define conditions which guarantee equivalence of S-Resolution and SLD-resolution, one of which happens to be exactly the property of productivity. We use the realizability transformation as a method for guaranteeing productivity of programs.

Finally, in Section 5 we conclude and explain how the combination of S-Resolution and the type-theoretic approach of this paper could be used in non-terminating cases of type class inference. Detailed proofs for lemmas and theorems in this paper may be found in the extended version[1].

[1] Extended version is available at both authors' homepages.

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

We first formulate a type system to model LP. We show that LP-Unif is 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 $\Phi \ ::= \cdot \mid \kappa : F, \Phi$

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. We write $A \mapsto_\sigma A'$ (resp. $A \sim_\gamma A'$) to mean A is matchable (resp. unifiable) to A' with substitution σ (resp. γ), i.e. $\sigma A \equiv A'$ (resp. $\gamma A \equiv \gamma A'$).

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

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

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

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

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 that the cut rule and proper axioms in intuitionistic sequent calculus can emulate LP [3](§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.

Below is a formulation of SLD-derivation as a reduction system [9].

Definition 3 (LP-Unif reduction). *Given axioms Φ. We define a reduction relation on the multiset of atomic formulas:*
$\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \rightsquigarrow_{\kappa,\gamma\cdot\gamma'} \{\gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n\}$ *for any substitution γ', if there exists $\kappa : \forall \underline{x}.B_1, ..., B_n \Rightarrow C \in \Phi$ such that $C \sim_\gamma A_i$.*

The second subscript in the reduction is intended as a state, it will be updated along with reductions. We assume implicit renaming of all quantified variables each time the above rule is applied. We write $\rightsquigarrow$ when we leave the underlining 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 γ.

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

Definition 4 (LP-Unif). *Given a logic program Φ, LP-Unif is given by an abstract reduction system $(\Phi, \rightsquigarrow)$.*

Lemma 1. *If $\Phi \vdash \{A_1, ..., A_n\} \rightsquigarrow^*_\gamma \emptyset$, 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, $\Phi \vdash \{A\} \rightsquigarrow_{\kappa,\gamma} \emptyset$. 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 $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \rightsquigarrow_{\kappa,\gamma} \{\gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n\}$ $\rightsquigarrow^*_{\gamma'} \emptyset$, where $\kappa : \forall \underline{x}.B_1, ..., B_m \Rightarrow C$ and $C \sim_\gamma A_i$. By inductive hypothesis(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 the proof $(\lambda b.(\kappa_1\ b)\ \kappa_3)\ \kappa_2$ for the formula $\Rightarrow \mathrm{Connect}(\mathrm{node}_1, \mathrm{node}_3)$.

Naturally, we would want to prove the following completeness theorem: If $e : \forall \underline{x}. \Rightarrow A$, then $\Phi \vdash \{A\} \rightsquigarrow^*_\gamma \emptyset$ for some γ. It is tempting to prove this theorem by induction on the derivation of $e : \forall \underline{x}. \Rightarrow A$. However, it becomes quite involved. We will discuss a simpler way to prove this theorem at the end of the next section, where we take advantage of the realizability transformation.

3 Realizability Transformation

We define *realizability transformation* in this section. Realizability [7](§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 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 2 (Strong Normalization). *Let beta-reduction on proof terms be the congruence closure of the following relation: $(\lambda a.p)p' \to_\beta [p'/a]p$. If $e : F$, then e is strongly normalizable with respect to 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 3. *If $e : [\forall \underline{x}.] \Rightarrow B$, then e is normalizable to a first order proof term.*

Lemma 2 and Theorem 3 show that we can use first order terms to represent normalized proof terms; and thus pave the way to realizability transformation.

Definition 5 (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 6. *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 7 (Realizability Transformation). *We define a transformation F on a 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 predicate, so that the proof can be recorded alongside with reductions.

Example 2. The following logic program is the result of applying realizability transformation on the program in Example 1.

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

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

The following theorem shows that realizability transformation does not change the proof-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 4. *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 5. *Suppose $F(\Phi) \vdash \{A[y]\} \leadsto_{\gamma}^{*} \emptyset$. 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. *$\Phi \vdash \{A_1, ..., A_n\} \leadsto^{*} \emptyset$ iff $F(\Phi) \vdash \{A_1[y_1], ..., A_n[y_n]\} \leadsto^{*} \emptyset$.*

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

Theorem 6. *$\Phi \vdash \{A\} \leadsto^{*} \emptyset$ iff $F(\Phi) \vdash \{A[y]\} \leadsto^{*} \emptyset$.*

Example 3. Consider the logic program after realizability transformation in Example 2. Realizability transformation does not change the behaviour of LP-Unif, we still have the following successful unification reduction path for query Connect(x, y, u):

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

Now let us come back to the completeness theorem. The following lemma shows that completeness result holds for the transformed program.

Lemma 4. *For $F(\Phi)$, if $n : \Rightarrow A[[\![n]\!]_\emptyset]$ where n is in normal form, then $F(\Phi) \vdash \{A[[\![n]\!]_\emptyset]\} \leadsto^* \emptyset$.*

Proof. By induction on the structure of n.

- **Base Case:** $n = \kappa$. In this case, $[\![n]\!]_\emptyset = f_\kappa$, $\kappa : \forall \underline{x}. \Rightarrow A'[f_\kappa] \in F(\Phi)$ and $\gamma(A'[f_\kappa]) \equiv A[f_\kappa]$ for some substitution γ. Thus $A'[f_\kappa] \sim_\gamma A[f_\kappa]$, which implies $F(\Phi) \vdash \{A[f_\kappa]\} \leadsto_{\kappa,\gamma} \emptyset$.
- **Step Case:** $n = \kappa\ n_1\ n_2\ ...\ n_m$. In this case, $[\![n]\!]_\emptyset = f_\kappa([\![n_1]\!]_\emptyset, ..., [\![n_m]\!]_\emptyset)$, $\kappa : \forall \underline{x}\underline{y}.\ C_1[y_1], ..., C_m[y_m] \Rightarrow B[f_\kappa(y_1, ..., y_m)] \in F(\Phi)$. To obtain $n : \Rightarrow A[[\![n]\!]_\emptyset]$, we have to use $\kappa : \forall \underline{x}.\ C_1[[\![n_1]\!]_\emptyset], ..., C_m[[\![n_m]\!]_\emptyset] \Rightarrow B[f_\kappa([\![n_1]\!]_\emptyset, ..., [\![n_m]\!]_\emptyset)]$ with $\gamma(B[f_\kappa([\![n_1]\!]_\emptyset, ..., [\![n_m]\!]_\emptyset)]) \equiv A[[\![n]\!]_\emptyset]$. By the inst rule, we have $\kappa : \gamma C_1[[\![n_1]\!]_\emptyset], ..., \gamma C_m[[\![n_m]\!]_\emptyset] \Rightarrow \gamma B[f_\kappa([\![n_1]\!]_\emptyset, ..., [\![n_m]\!]_\emptyset)]$. Furthermore, it has to be the case that $n_1 : \Rightarrow \gamma C_1[[\![n_1]\!]_\emptyset], ..., n_m : \Rightarrow \gamma C_m[[\![n_m]\!]_\emptyset]$. Thus we have $F(\Phi) \vdash \{A[[\![n]\!]_\emptyset]\} \leadsto_{\kappa,\gamma} \{\gamma C_1[[\![n_1]\!]_\emptyset], ..., \gamma C_m[[\![n_m]\!]_\emptyset]\}$. By IH, we have $F(\Phi) \vdash \{\gamma C_1[[\![n_1]\!]_\emptyset]\} \leadsto^*_{\gamma_1} \emptyset$. So $F(\Phi) \vdash \{A[[\![n]\!]_\emptyset]\} \leadsto_{\kappa,\gamma} \cdot \leadsto^*_{\gamma_1}$ $\{\gamma_1\gamma C_2[[\![n_2]\!]_\emptyset], ..., \gamma_1\gamma C_m[[\![n_m]\!]_\emptyset]$. Again, we have $n_2 : \Rightarrow \gamma_1\gamma C_2[[\![n_2]\!]_\emptyset], ..., n_m : \Rightarrow \gamma_1\gamma C_m[[\![n_m]\!]_\emptyset]$. By applying IH repeatedly, we obtain $F(\Phi) \vdash \{A[[\![n]\!]_\emptyset]\} \leadsto^* \emptyset$.

Lemma 5. *For $F(\Phi)$, if $F(\Phi) \vdash \{A_1[t_1], ..., A_n[t_n]\} \leadsto^* \emptyset$ with $\mathrm{FV}(t_i) = \emptyset$ for all i, then $\Phi \vdash \{A_1, ..., A_n\} \leadsto^* \emptyset$.*

Proof. By induction on the length of $\leadsto^*$.

- **Base Case:** $F(\Phi) \vdash \{A[f_\kappa]\} \leadsto^* \emptyset$. We have $\kappa : \Rightarrow A' \in \Phi$ such that $A' \sim_\gamma A$. Thus $\Phi \vdash \{A\} \leadsto_\kappa \emptyset$.
- **Step Case:** $F(\Phi) \vdash \{A_1[t_1], ..., A_i[t_i], ..., A_n[t_n]\} \leadsto_{\kappa,\gamma}$ $\{\gamma A_1[t_1], ..., \gamma B_1[t'_1], ..., \gamma B_l[t'_l], ..., \gamma A_n[t_n]\} \leadsto^* \emptyset$ with $t_i \equiv f_\kappa(t'_1, ..., t'_l)$ and $\kappa : B_1, ..., B_l \Rightarrow C \in \Phi$ where $C \sim_\gamma A_i$. So by IH, we have $\Phi \vdash \{A_1, ..., A_n\} \leadsto_{\kappa,\gamma}$ $\{\gamma A_1, ..., \gamma B_1, ..., \gamma B_l, ..., \gamma A_n\} \leadsto^* \emptyset$.

Now we are ready to prove the completeness result.

Theorem 7 (Completeness). *If $n : [\forall \underline{x}]. \Rightarrow A$, where n is in normal form, then $\Phi \vdash \{A\} \leadsto^*_\gamma \emptyset$.*

Proof. By Theorem 4, we have $n : [\forall \underline{x}]. \Rightarrow A[\llbracket n \rrbracket_\emptyset]$ holds in $F(\Phi)$. By Lemma 4, we have $F(\Phi) \vdash \{A[\llbracket n \rrbracket_\emptyset]\} \rightsquigarrow^* \emptyset$. By Lemma 5, we have $\Phi \vdash \{A\} \rightsquigarrow^*_\gamma \emptyset$.

The completeness result relies on realizability transformation to record the proof steps for a query, so the LP-Unif reduction can just follow the proof steps to reduce the query to the empty set. Together with Theorem 1, this proof system gives new semantics for derivations in LP.

4 Structual Resolution

S-resolution [5] is a newly proposed alternative to SLD-resolution that allows a systematic separation of derivations into term-matching and unification steps. A logic program is called *productive* if the term-matching reduction is terminating for any query. For productive programs with coinductive meaning, finite term-rewriting reductions can be seen as measures of observation in an infinite derivation. The ability to handle corecursion in a productive way is an attractive computational feature of S-resolution.

Example 4. The following program defines the predicate Stream:

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

It will result in infinite LP-Unif reduction:

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

But it will yield finite term-matching reduction since Stream(y) can not be matched by the head of κ_1 (Stream(Cons(x,y))):

$$\Phi \vdash \{\mathrm{Stream}(\mathrm{Cons}(x,y))\} \to_{\kappa_1} \{\mathrm{Stream}(y)\} \not\to$$

In general, term-matching reductions are not complete relative to LP-Unif reductions, but we can combine them with substitutional steps to complete derivations. This is exactly the idea behind S-resolution.

Example 5. The following program defines bits and lists of bits:

$$\kappa_1 : \Rightarrow \mathrm{Bit}(0)$$
$$\kappa_2 : \Rightarrow \mathrm{Bit}(1)$$
$$\kappa_3 : \Rightarrow \mathrm{BList}(\mathrm{Nil})$$
$$\kappa_4 : \forall x.\forall y.\mathrm{BList}(y), \mathrm{Bit}(x) \Rightarrow \mathrm{BList}(\mathrm{Cons}(x,y))$$

LP-Unif would give a complete reduction:

$$\Phi \vdash \{\mathrm{BList}(\mathrm{Cons}(x,y))\} \rightsquigarrow_{\kappa_4,[x/x_1,y/y_1]} \{\mathrm{Bit}(x), \mathrm{BList}(y)\} \rightsquigarrow_{\kappa_1,[0/x,0/x_1,y/y_1]}$$
$$\{\mathrm{BList}(y)\} \rightsquigarrow_{\kappa_3,[\mathrm{Nil}/y,0/x,0/x_1,\mathrm{Nil}/y_1]} \emptyset$$

But term-matching reduction will not be able to compute an answer in this case.

$$\Phi \vdash \{\mathrm{BList}(\mathrm{Cons}(x,y))\} \to_{\kappa_4} \{\mathrm{Bit}(x), \mathrm{BList}(y)\} \not\to$$

This is why, S-resolution combines term-matching reductions with additional substitutional steps, in order to compute the same answer:

$$\Phi \vdash \{\mathrm{BList}(\mathrm{Cons}(x,y))\} \to_{\kappa_4} \{\mathrm{Bit}(x), \mathrm{BList}(y)\} \hookrightarrow_{\kappa_1,[0/x]} \{\mathrm{Bit}(0), \mathrm{BList}(y)\} \to_{\kappa_1,[0/x]} \{\mathrm{BList}(y)\} \hookrightarrow_{\kappa_3,[0/x,\mathrm{Nil}/y]} \{\mathrm{BList}(\mathrm{Nil})\} \to_{\kappa_3,[0/x,\mathrm{Nil}/y]} \emptyset$$

Completing derivation for Stream in the same way will result in an infinite derivation, in which every term-matching reduction is finite.

In this section, we embed S-resolution into the type theoretic framework we have developed in the previous sections. We first define S-derivations in terms of LP-Struct reductions, in the uniform style with LP-Unif reductions, thereby also defining LP-TM reductions, which resemble reductions in term-rewriting systems [10]. We then prove that LP-Unif and LP-Struct are operationally equivalent subject to two conditions: productivity and non-overlapping. Finally, we show how realizability transformation can be used to guarantee productivity of logic programs in the setting of S-resolution.

4.1 S-resolution in the Type-Theoretic Setting

Definition 8.

– ***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 $C \mapsto_\sigma A_i$.
– ***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 the substitutional reduction is intended as a state, it will be updated along with reductions.

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

LP-TM is also sound w.r.t. the type system of Definition 2, which implies that we can obtain a proof for each successful query.

Theorem 8 (Soundness of LP-TM). *If $\Phi \vdash \{A\} \to^* \emptyset$, then there exists a proof $e : \forall \underline{x}. \Rightarrow A$ given axioms Φ.*

Comparing Theorem 1 and Theorem 8, we see that for LP-TM, there is no need to accumulate substitutions, and the resulting formula is proven as stated. This difference is due to the use of term-matching instead of unification for the reduction. The following example shows that the LP-TM is incomplete with respect to the type system.

Example 6. Consider the following program Φ.

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

For query $P(\mathrm{C})$, we have $\Phi \vdash \{P(\mathrm{C})\} \to_{\kappa_2} \{Q(x)\} \not\to$. However, there exist a proof $(\kappa_2\ \kappa_1) : \Rightarrow P(\mathrm{C})$, by instantiating x, y to C in κ_2.

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

We can now formally define S-Resolution within our formal framework. 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 10 (Structural Resolution (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 $\Phi \vdash \{\underline{A}\} \to^{\mu} \cdot \hookrightarrow \cdot ... \cdot \to^{\mu} \cdot \hookrightarrow \cdot \to^{\mu} \{\underline{C}\}$.

Now let us see the execution trace of Stream using LP-Struct:

$$\begin{gathered}
\Phi \vdash \{\mathrm{Stream}(\mathrm{Cons}(x, y))\} \to_{\kappa_1} \{\mathrm{Stream}(y)\} \hookrightarrow_{\kappa_1, [\mathrm{Cons}(x_2, y_2)/y]} \\
\{\mathrm{Stream}(\mathrm{Cons}(x_2, y_2))\} \to_{\kappa_1, [\mathrm{Cons}(x_2, y_2)/y]} \\
\{\mathrm{Stream}(y_2)\} \hookrightarrow_{\kappa_1, [\mathrm{Cons}(x_3, y_3)/y_2, \mathrm{Cons}(x_2, \mathrm{Cons}(x_3, y_3))/y]} \\
\{\mathrm{Stream}(\mathrm{Cons}(x_3, y_3))\} \to_{\kappa_1, [\mathrm{Cons}(x_3, y_3)/y_2, \mathrm{Cons}(x_2, \mathrm{Cons}(x_3, y_3))/y]} \{\mathrm{Stream}(y_3)\} \dots
\end{gathered}$$

Note that the overall reduction is infinite, but each LP-TM reduction is finite.

4.2 LP-Struct and LP-Unif

The next question one may ask is how LP-Struct compares to LP-Unif. They are not equivalent. Consider the program and the finite LP-Unif derivation of Example 1. LP-Unif has a finite successful derivation for the query $\mathrm{Connect}(x, y)$, but we have the following non-terminating reduction by LP-Struct:

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

The diverging behavior above is due to the divergence of LP-TM reduction. Therefore, the program of Example 1 is not productive in the sense of [8,5].

Definition 11 (Productivity). *We say a program Φ is productive iff every $\to$-reduction is finite.*

Perhaps LP-Unif and LP-Struct are operationally equivalent for all productive programs? The following example shows this is not the case.

Example 7.

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

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

Thus, productivity is insufficient for establishing the relation between LP-Struct and LP-Unif. In Example 7, the problem is caused by the overlapping heads $P(\mathrm{C})$ and $P(x)$. Motivated by the notion of non-overlapping rules in term rewriting systems ([10,1]), we introduce the following definition.

Definition 12 (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 $\sigma C \equiv \delta E$.*

Theorem 9. *Suppose Φ is non-overlapping. $\Phi \vdash \{A_1, ..., A_n\} \leadsto_\gamma^* \{C_1, ..., C_m\}$ with $\{C_1, ..., C_m\}$ in $\leadsto$-normal form iff $\Phi \vdash \{A_1, ..., A_n\}(\to^\mu \cdot \hookrightarrow^1)_\gamma^* \{C_1, ..., C_m\}$ with $\{C_1, ..., C_m\}$ in $\to^\mu \cdot \hookrightarrow^1$-normal form.*

The theorem above still requires the termination of the $\leadsto$ to establish equivalence LP-Unif and LP-Struct. We can weaken this requirement by only requiring termination of the $\to$-reduction, i.e. by requiring productivity.

Theorem 10 (Equivalence of LP-Struct and LP-Unif). *Suppose Φ is non-overlapping and productive.*

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

Note that the above theorem does not rely on termination of LP-Unif reductions and therefore establishes equivalence of LP-Unif and LP-Struct even for coinductive programs like Stream of Example 4, as long as they are productive and non-overlapping. This effect of productivity has not been described in previous work.

4.3 Realizability Transformation and LP-Struct

Even when programs are overlapping and unproductive (as e.g. the program of Example 1), we would still like to obtain a meaningful execution behaviour for LP-Struct, especially if LP-Unif allows successful derivations for the programs. Luckily, we already have a method to achieve that, it is the realizability transformation defined in Section 3:

Proposition 1. *For any program Φ, $F(\Phi)$ is productive and non-overlapping.*

Proof. First, we need to show $\rightarrow$-reduction is strongly normalizing in $(F(\Phi), \rightarrow)$. By Definition 7, 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 7.

Corollary 1.
$F(\Phi) \vdash \{A_1, ..., A_n\}(\rightarrow^{\mu} \cdot \hookrightarrow^{1})^{*}\{B_1, ..., B_m\}$ *iff* $F(\Phi) \vdash \{A_1, ..., A_n\} \rightsquigarrow^{*} \{B_1, ..., B_m\}$.

Proof. By Theorem 10 and Theorem 1.

Example 8. For the program in Example 2, the query 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 $\Rightarrow \mathrm{Connect}(\mathrm{node}_1, \mathrm{node}_3)$.

Realizability transformation uses the extra argument as decreasing measurement in the program to achieve termination of $\rightarrow$-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 9. 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 $\rightarrow^{\mu}$.

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

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 $P(\text{List}(\text{Int}))$ 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 4-6, realizability transformation does not modify the execution behaviour of unification reduction.

Example 10. 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 $\to$-reduction:
$\Phi \vdash \{P(x)\} \to \{P(x)\} \to \{P(x)\}...$
The another is due to $\hookrightarrow$-reduction:
$\Phi \vdash \{P(x,y)\} \hookrightarrow \{P(x, f_k(u))\} \to \{P(x,u)\} \hookrightarrow \{P(x, f_k(u'))\} \to \{P(x,u')\}...$

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, which is exactly the correspondence we were looking for.

5 Conclusions and Future Work

We proposed a type system that gives a proof theoretic interpretation for LP: 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 operational behaviour of LP-Unif.

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 ([5,8]) only gives soundness and completeness of LP-Struct with respect to the Herbrand models. We identified that productivity and non-overlapping are essential for showing their operational equivalence.

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 4-6. 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 5 establishes, which helps to prove completeness of LP-Unif (Theorem 7).

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. [11,6].

In the context of type class inference [11,6], the infinite term-matching behaviour seems pervasive. The example below specifies a possible equality instance declaration for nested datatype such as
`data Bush a = Nil | Con a (Bush (Bush a))`:

$$\begin{array}{l} \kappa_1 : \ \mathrm{Eq}(x), \mathrm{Eq}(\mathrm{Bush}(\mathrm{Bush}(x))) \Rightarrow \mathrm{Eq}(\mathrm{Bush}(x)) \\ \kappa_2 : \ \Rightarrow \mathrm{Eq}(\mathrm{Char}) \end{array}$$

Here Bush is a function symbol, Char is a constant and x is variable. Consider the query Eq(Bush(Char)), both LP-Unif and LP-Struct will generate an infinite reduction path by repeatedly applying κ_1. Using the realizability transformation, we can obtain a well-behaved (productive) program:

$$\begin{array}{l} \kappa_1 : \ \mathrm{Eq}(x, y_1), \mathrm{Eq}(\mathrm{Bush}(\mathrm{Bush}(x)), y_2) \Rightarrow \mathrm{Eq}(\mathrm{Bush}(x), f_{\kappa_1}(y_1, y_2)) \\ \kappa_2 : \ \Rightarrow \mathrm{Eq}(\mathrm{Char}, \mathrm{c}_{\kappa_2}) \end{array}$$

The substitution for u in the query $\mathrm{Eq}(\mathrm{Bush}(\mathrm{Char}), u)$ will be an infinite term. But we need a finite representation for such infinite term to construct a dictionary. Such coinductive dictionary construction is the subject of our further investigations. We would also like to investigate generalizing the type-theoretic approach from Horn formulas to implicational intuitionistic formulas, the type system in this case will correspond to a version of simply type lambda calculus.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, New York, NY, USA, 1998.
2. Y. Bertot and E. Komendantskaya. Inductive and coinductive components of corecursive functions in coq. *Electronic notes in theoretical computer science*, 203(5):25–47, 2008.
3. J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types.* Cambridge University Press, New York, NY, USA, 1989.
4. G. Gonthier, B. Ziliani, A. Nanevski, and D. Dreyer. How to make ad hoc proof automation less ad hoc. *ACM SIGPLAN Notices*, 46(9):163–175, 2011.
5. P. Johann, E. Komendantskaya, and V. Komendantskiy. Structural resolution for logic programming. In *Technical Communications, ICLP*, 2015.
6. M. P. Jones. *Qualified types: theory and practice.* Cambridge University Press, 2003.
7. S. C. Kleene. *Introduction to metamathematics.* North-Holland Publishing Company, 1952. Co-publisher: Wolters–Noordhoff; 8th revised ed.1980.
8. E. Komendantskaya, J. Power, and M. Schmidt. Coalgebraic logic programming: from semantics to implementation. *Journal of Logic and Computation*, 2014.
9. U. Nilsson and J. Małuszyński. *Logic, programming and Prolog.* Wiley Chichester, 1990.
10. Terese. *Term rewriting systems.* Cambridge University Press, 2003.
11. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Symposium on Principles of Programming Languages*, pages 60–76. ACM, 1989.

A Proof of Theorem 2 and 3

We are going to prove a nontrival property about the type system that we just set up. The proof is a simplification of Tait-Girard's reducibility method.

Definition 13 (Reducibility Set). *Let N denotes the set of all strong normalizing proof terms. We define reducibility set RED_F by induction on structure of F:*

- $p \in \mathsf{RED}_{A_1,...,A_n \Rightarrow B}$ *with* $n \geq 0$ *iff for any* $p_i \in N$, $p\ p_1\ ...\ p_n \in N$.
- $p \in \mathsf{RED}_{\forall \underline{x}.A_1,...,A_n \Rightarrow B}$ *iff* $p \in \mathsf{RED}_{A_1,...,A_n \Rightarrow B}$.

Lemma 6. $\mathsf{RED}_{\underline{A} \Rightarrow B} = \mathsf{RED}_{\phi \underline{A} \Rightarrow \phi B}$.

Lemma 7. *If* $p \in \mathsf{RED}_F$, *then* $p \in N$.

Proof. By Induction on F:

- Base Case: F is of the form $A_1, ..., A_n \Rightarrow B$. By definition, $p\ p_1\ ...\ p_n \in N$ for any $p_i \in N$. Thus $p \in N$.
- Step Case: F is of the form $\forall \underline{x}.A_1, ..., A_n \Rightarrow B$. $p \in \mathsf{RED}_{\forall \underline{x}.A_1,...,A_n \Rightarrow B}$ implies $p \in \mathsf{RED}_{A_1,...,A_n \Rightarrow B}$. Thus by IH, $p \in N$.

Lemma 8. *If* $e : F$, $e \in \mathsf{RED}_F$.

Proof. By induction on derivation of $e : F$.

- Base Case:
  $$\frac{}{\kappa : \forall \underline{x}. \Rightarrow B}$$
  This case $\kappa \in N$.
- Base Case:
  $$\frac{}{\kappa : \forall \underline{x}.A_1, ..., A_n \Rightarrow B}$$
  Since κ is a constant, thus for any $p_i \in N$, $\kappa\ p_1\ ...\ p_n \in N$. So $\kappa \in \mathsf{RED}_{A_1,...,A_n \Rightarrow B}$, thus $\kappa \in \mathsf{RED}_{\forall \underline{x}.A_1,...,A_n \Rightarrow B}$.
- Step Case:
  $$\frac{e_1 : \underline{A} \Rightarrow D \quad e_2 : \underline{B}, D \Rightarrow C}{\lambda \underline{a}.\lambda \underline{b}.(e_2\ \underline{b})\ (e_1\ \underline{a}) : \underline{A}, \underline{B} \Rightarrow C}\ cut$$
  We need to show $\lambda \underline{a}.\lambda \underline{b}.(e_2\ \underline{b})\ (e_1\ \underline{a}) \in \mathsf{RED}_{\underline{A},\underline{B} \Rightarrow C}$. By IH, we know that $(e_2\ \underline{b})\ (e_1\ \underline{a}) \in N$. Let $p_1 \in N, ..., p_n \in N, q_1 \in N, ..., q_m \in N$. We are going to show for any e with $(\lambda \underline{a}.\lambda \underline{b}.(e_2\ \underline{b})\ (e_1\ \underline{a}))\ \underline{p}\ \underline{q} \to_\beta e$, then $e \in N$. We proceed by induction on $(\mu((e_2\ \underline{b})\ (e_1\ \underline{a})), \mu(\underline{p}), \mu(\underline{q}))$, where μ is a function to get the length of the reduction path to normal form.
  - Base Case: $(\mu((e_2\ \underline{b})\ (e_1\ \underline{a})), \mu(\underline{p}), \mu(\underline{q})) = (0, ..., 0)$. The only reduction possible is $(\lambda \underline{a}.\lambda \underline{b}.(e_2\ \underline{b})\ (e_1\ \underline{a}))\ \underline{p}\ \underline{q} \to_\beta (e_2\ \underline{q})\ (e_1\ \underline{p})$. We know that $(e_2\ \underline{q})\ (e_1\ \underline{p}) \in N$.
  - Step Case: There are several possible reductions, but all will decrease $(\mu((e_2\ \underline{b})\ (e_1\ \underline{a})), \mu(\underline{p}), \mu(\underline{q}))$, thus we conclude that by induction hypothesis.

  So $\lambda \underline{a}.\lambda \underline{b}.(e_2\ \underline{b})\ (e_1\ \underline{a}) \in N$.

– Step Case:
$$\frac{e : \forall \underline{x}.F}{e : [\underline{t}/\underline{x}]F}\ \mathit{inst}$$
By IH, we konw that $e \in \mathsf{RED}_{\forall \underline{x}.F}$, so by definition we know that $e \in \mathsf{RED}_F$. By Lemma 6, $e \in \mathsf{RED}_{[\underline{t}/\underline{x}]F}$.
– Step Case:
$$\frac{e : F}{e : \forall \underline{x}.F}\ \mathit{gen}$$
By IH, we know that $e \in \mathsf{RED}_F$, so we know that $e \in \mathsf{RED}_{\forall \underline{x}.F}$.

Theorem 11 (Strong Normalization). *If $e : F$, then $e \in N$.*

Proof. By Lemma 8.

Definition 14 (First Orderness). *We say p is first order inductively:*

– *A proof term variable a or proof term constant κ is first order.*
– *if n, n' are first order, then $n\ n'$ is first order.*

Lemma 9. *If n, n' are first order, then $[n'/a]n$ is first order.*

Lemma 10. *If $e : [\forall \underline{x}.]\underline{A} \Rightarrow B$, 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 term.*

Proof. By induction on the derivation of $e : [\forall \underline{x}.]\underline{A} \Rightarrow B$.

– Base Cases: Axioms, in this case e is a proof term constant.
– Step Case:
$$\frac{e_1 : \underline{A} \Rightarrow D \quad e_2 : \underline{B}, D \Rightarrow C}{\lambda \underline{a}.\lambda \underline{b}.(e_2\ \underline{b})\ (e_1\ \underline{a}) : \underline{A}, \underline{B} \Rightarrow C}\ \mathit{cut}$$
By IH, we know that $e_1 = \kappa$ or $e_1 = \lambda \underline{a}.n_1$; $e_2 = \kappa'$ or $e_2 = \lambda \underline{b}d.n_2$. We know that $e_1\underline{a}$ will be normalizable to a first order proof term. And $e_2\underline{b}$ will be normalized to either $\kappa'\underline{b}$ or $\lambda d.n_2$. So by Lemma 9, we conclude that $\lambda \underline{a}.\lambda \underline{b}.(e_2\ \underline{b})\ (e_1\ \underline{a})$ is normalizable to $\lambda \underline{a}.\lambda \underline{b}.n$ for some first order normal term n.
– The other cases are straightforward.

Theorem 12. *If $e : [\forall \underline{x}.] \Rightarrow B$, then e is normalizable to a first order proof term.*

Proof. By lemma 10, subject reduction and strong normalization theorem.

B Proof of Theorem 4

Theorem 13. *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 $F(\Phi)$.*

Proof. By induction on the derivation of $e : [\forall \underline{x}].\underline{A} \Rightarrow B$.

– Base Case:

$$\frac{}{\kappa : \forall \underline{x}.\underline{A} \Rightarrow B}$$

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

– Step Case:

$$\frac{e_1 : \underline{A} \Rightarrow D \quad e_2 : \underline{B}, D \Rightarrow C}{\lambda \underline{a}.\lambda \underline{b}.(e_2\ \underline{b})\ (e_1\ \underline{a}) : \underline{A}, \underline{B} \Rightarrow C}\ \mathit{cut}$$

We know that the normal form of e_1 must be κ_1 or $\lambda \underline{a}.n_1$; the normal form of e_1 must be κ_2 or $\lambda \underline{b}d.n_2$, with n_1, n_2 are first order.

- $e_1 \equiv \kappa_1, e_2 \equiv \kappa_2$. By IH, we know that $F(\kappa_1 : \underline{A} \Rightarrow D) = \kappa_1 : A_1[y_1], ..., A_1[y_1] \Rightarrow D[f_{\kappa_1}(y_1, ..., y_n)]$ and $F(\kappa_2 : \underline{B}, D \Rightarrow C) = \kappa_2 : B_1[z_1], ..., B_m[z_m], D[y] \Rightarrow C[f_{\kappa_2}(z_1, ..., z_m, y)]$ hold. So by *gen* and *inst*, we have
  $\kappa_2 : B_1[z_1], ..., B_m[z_m], D[f_{\kappa_1}(y_1, ..., y_n)] \Rightarrow C[f_{\kappa_2}(\underline{z}, f_{\kappa_1}(\underline{y}))]$.
  Then by the cut rule, we have
  $\lambda \underline{a}.\lambda \underline{b}.\kappa_2 \underline{b}(\kappa_1 \underline{a}) : A_1[y_1], ..., A_1[y_1], B_1[z_1], ..., B_m[z_m] \Rightarrow C[f_{\kappa_2}(\underline{z}, f_{\kappa_1}(\underline{y}))]$.
  We can see that $\llbracket \kappa_2 \underline{b}(\kappa_1 \underline{a}) \rrbracket_{[\underline{y}/\underline{a}, \underline{z}/\underline{b}]} = f_{\kappa_2}(\underline{z}, f_{\kappa_1}(\underline{y}))$.
- $e_1 \equiv \lambda \underline{a}.n_1, e_2 \equiv \lambda \underline{b}d.n_2$. By IH, we know that $F(\lambda \underline{a}.n_1 : \underline{A} \Rightarrow D) = \lambda \underline{a}.n_1 : A_1[y_1], ..., A_1[y_1] \Rightarrow D[\llbracket n_1 \rrbracket_{[\underline{y}/\underline{a}]}]$ and $F(\lambda \underline{b}d.n_2 : \underline{B}, D \Rightarrow C) = \lambda \underline{b}d.n_2 : B_1[z_1], ..., B_m[z_m], D[y] \Rightarrow C[\llbracket n_2 \rrbracket_{[\underline{z}/\underline{b}, y/d]}]$ hold. So by *gen* and *inst*, we have
  $\lambda \underline{b}d.n_2 : B_1[z_1], ..., B_m[z_m], D[\llbracket n_1 \rrbracket_{[\underline{y}/\underline{a}]}] \Rightarrow C[\llbracket n_2 \rrbracket_{[\underline{z}/\underline{b}, \llbracket n_1 \rrbracket_{[\underline{y}/\underline{a}]}/d]}]$.
  Then by the cut rule and beta reductions, we have $\lambda \underline{a}.\lambda \underline{b}.([n_1/d]n_2) : A_1[y_1], ..., A_1[y_1], B_1[z_1], ..., B_m[z_m] \Rightarrow C[\llbracket n_2 \rrbracket_{[\underline{z}/\underline{b}, \llbracket n_1 \rrbracket_{[\underline{y}/\underline{a}]}/d]}]$. We know that $\llbracket [n_1/d]n_2 \rrbracket_{[\underline{y}/\underline{a}, \underline{z}/\underline{b}]} = \llbracket n_2 \rrbracket_{[\underline{z}/\underline{b}, \llbracket n_1 \rrbracket_{[\underline{y}/\underline{a}]}/d]}$.
- The other cases are handle similarly.

– Step Case:

$$\frac{\lambda \underline{a}.n : \forall \underline{x}.\underline{A} \Rightarrow B}{\lambda \underline{a}.n : [\underline{t}/\underline{x}]\underline{A} \Rightarrow [\underline{t}/\underline{x}]B}\ \mathit{inst}$$

By IH, we know that $F(\lambda \underline{a}.n : \forall \underline{x}.\underline{A} \Rightarrow B) = \lambda \underline{a}.n : \forall \underline{x}.\forall \underline{y}.A_1[y_1], ..., A_n[y_n] \Rightarrow B[\llbracket n \rrbracket_{[\underline{y}/\underline{a}]}]$ holds for $F(\Phi)$. By *Inst* rule, we instantiate y_i with y_i, we have $\lambda \underline{a}.n : [\underline{t}/\underline{x}]A_1[y_1], ..., [\underline{t}/\underline{x}]A_n[y_n] \Rightarrow [\underline{t}/\underline{x}]B[\llbracket n \rrbracket_{[\underline{y}/\underline{a}]}]$

– Step Case:

$$\frac{e : F}{e : \forall \underline{x}.F}\ \mathit{gen}$$

This case is straightforwardly by IH.

C Proof of Theorem 5

Lemma 11. *If $F(\Phi) \vdash \{A_1[y_1], ..., A_n[y_n]\} \rightsquigarrow^*_\gamma \emptyset$, and $y_1, ..., y_n$ are fresh, then there exists proofs $e_1 : \forall \underline{x}. \Rightarrow \gamma A_1[\gamma y_1], ..., e_n : \forall \underline{x}. \Rightarrow \gamma A_n[\gamma y_n]$ with $[\![e_i]\!]_\emptyset = \gamma y_i$ given axioms $F(\Phi)$.*

Proof. By induction on the length of the reduction.

- Base Case. Suppose the length is one, namely, $F(\Phi) \vdash \{A[y]\} \rightsquigarrow_{\kappa,\gamma_1} \emptyset$. Thus there exists $(\kappa : \forall \underline{x}. \Rightarrow C[f_\kappa]) \in F(\Phi)$(here f_κ is a constant), such that $C[f_\kappa] \sim_{\gamma_1} A[y]$. Thus $\gamma_1(C[f_\kappa]) \equiv \gamma_1 A[\gamma_1 y]$. So $\gamma_1 y \equiv f_\kappa$ and $\gamma_1 C \equiv \gamma_1 A$. We have $\kappa : \Rightarrow \gamma_1 C[f_\kappa]$ by the *inst* rule, thus $\kappa : \Rightarrow \gamma_1 A[\gamma_1 y]$, hence $\kappa : \forall \underline{x}. \Rightarrow \gamma_1 A[\gamma_1 y]$ by the *gen* rule and $[\![\kappa]\!]_\emptyset = f_\kappa$.
- Step Case. Suppose $F(\Phi) \vdash \{A_1[y_1], ..., A_i[y_i], ..., A_n[y_n]\} \rightsquigarrow_{\kappa,\gamma_1}$ $\{\gamma_1 A_1[y_1], ..., \gamma_1 B_1[z_1], ..., \gamma_1 B_m[z_m], ..., \gamma_1 A_n[y_n]\} \rightsquigarrow^*_\gamma \emptyset$, where $\kappa : \forall \underline{x}.\forall \underline{z}.B_1[z_m], ..., B_n[z_m] \Rightarrow C[f_\kappa(z_1, ..., z_m)] \in F(\Phi)$, and $C[f_\kappa(z_1, ..., z_m)] \sim_{\gamma_1} A_i[y_i]$. So we know $\gamma_1 C[f_\kappa(z_1, ..., z_m)] \equiv \gamma_1 A_i[\gamma_1 y_i]$, $\gamma_1 y_i \equiv f_\kappa(z_1, ..., z_m)$, $\gamma_1 C \equiv \gamma_1 A_i$ and $\mathrm{dom}(\gamma_1) \cap \{z_1, ..., z_m, y_1, .., y_{i-1}, y_{i+1}, y_n\} = \emptyset$. By IH, we know that there exists proofs $e_1 : \forall \underline{x}. \Rightarrow \gamma\gamma_1 A_1[\gamma y_1], ..., p_1 : \forall \underline{x}. \Rightarrow \gamma\gamma_1 B_1[\gamma z_1], ..., p_m : \forall \underline{x}. \Rightarrow \gamma\gamma_1 B_m[\gamma z_m], ..., e_n : \forall \underline{x}. \Rightarrow \gamma\gamma_1 A_n[\gamma y_n]$ and $[\![e_1]\!]_\emptyset = \gamma y_1, ..., [\![p_1]\!]_\emptyset = \gamma z_1, ..., [\![e_n]\!]_\emptyset = \gamma y_n$. We can construct a proof $e_i = \kappa\ p_1\ ...p_m$ with $e_i : \forall \underline{x}. \Rightarrow \gamma\gamma_1 A_i[\gamma\gamma_1 y_i]$, by first use the *inst* to instantiate the quantifiers of κ, then applying the cut rule m times. Moreover, we have $[\![\kappa\ p_1\ ...p_m]\!]_\emptyset = f_\kappa([\![p_1]\!]_\emptyset, ..., [\![p_m]\!]_\emptyset) = \gamma(f_\kappa(z_1, ..., z_m)) = \gamma\gamma_1 y_i$.

Theorem 14. *Given axioms Φ, suppose $F(\Phi) \vdash \{A[y]\} \rightsquigarrow^*_\gamma \emptyset$. We have $p : \forall \underline{x}. \Rightarrow \gamma A[\gamma y]$ where p is in normal form and $[\![p]\!]_\emptyset = \gamma y$.*

Proof. By Lemma 11.

D Proof of Lemma 3

Lemma 12. *If $\Phi \vdash \{A_1, ..., A_n\} \rightsquigarrow^* \emptyset$, then $F(\Phi) \vdash \{A_1[y_1], ..., A_n[y_n]\} \rightsquigarrow^* \emptyset$ with y_i fresh.*

Proof. By induction on the length of reduction.

- Base Case. Suppose the length is one, namely, $\Phi \vdash \{A\} \rightsquigarrow_{\kappa,\gamma_1} \emptyset$. Then there exists $(\kappa : \forall \underline{x}. \Rightarrow C) \in \Phi$ such that $C \sim_{\gamma_1} A$. Thus $\kappa : \forall \underline{x}. \Rightarrow C[f_\kappa] \in F(\Phi)$ and $(C[f_\kappa]) \sim_{\gamma_1[f_\kappa/y]} A[y]$. So $F(\Phi) \vdash A[y] \rightsquigarrow \emptyset$.
- Step Case. Suppose $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \rightsquigarrow_{\kappa,\gamma_1} \{\gamma_1 A_1, ..., \gamma_1 B_1, ..., \gamma_1 B_m, ..., \gamma_1 A_n\} \rightsquigarrow^*_\gamma \emptyset$, where $\kappa : \forall \underline{x}.B_1, ..., B_m \Rightarrow C \in \Phi$, $C \sim_{\gamma_1} A_i$. So we know that $\kappa : \forall \underline{x}.B_1[z_1], ..., B_m[z_m] \Rightarrow C[f_\kappa(\underline{z})] \in F(\Phi)$ and $C[f_\kappa(\underline{z})] \sim_{\gamma_1[f_\kappa(\underline{z})/y_i]} A_i[y_i]$. Thus $F(\Phi) \vdash \{A_1[y_1], ..., A_i[y_i], ..., A_n[y_n]\} \rightsquigarrow_{\kappa,\gamma_1[f_\kappa(\underline{z})/y_i]}$ $\{\gamma_1[f_\kappa(\underline{z})/y_i]A_1[y_1], ..., \gamma_1[f_\kappa(\underline{z})/y_i]B_1[z_1], ..., \gamma_1[f_\kappa(\underline{z})/y_i]B_m[z_m], ..., \gamma_1[f_\kappa(\underline{z})/y_i]A_n[y_n]\} \equiv$ $\{\gamma_1 A_1[y_1], ..., \gamma_1 B_1[z_1], ..., \gamma_1 B_m[z_m], ..., \gamma_1 A_n[y_n]\}$. By IH, $F(\Phi) \vdash \{\gamma_1 A_1[y_1], ..., \gamma_1 B_1[z_1], ..., \gamma_1 B_m[z_m], ..., \gamma_1 A_n[y_n]\} \rightsquigarrow^* \emptyset$.

Lemma 13. *If $F(\Phi) \vdash \{A_1[y_1], ..., A_n[y_n]\} \rightsquigarrow^* \emptyset$, then $\Phi \vdash \{A_1, ..., A_n\} \rightsquigarrow^* \emptyset$.*

Proof. By induction on the length of reduction.

– Base Case. Suppose the length is one, namely, $F(\Phi) \vdash \{A[y]\} \rightsquigarrow_{\kappa,\gamma_1} \emptyset$. Thus there exists $(\kappa : \forall \underline{x}. \Rightarrow C[f_\kappa]) \in F(\Phi)$ such that $C[f_\kappa] \sim_{\gamma_1} A[y]$. Thus $C \sim_{\gamma_1 - [f_\kappa/y]} A$. So $\Phi \vdash A \rightsquigarrow \emptyset$.
– Step Case. Suppose $F(\Phi) \vdash \{A_1[y_1], ..., A_i[y_i], ..., A_n[y_n]\} \rightsquigarrow_{\kappa,\gamma_1}$ $\{\gamma_1 A_1[y_1], ..., \gamma_1 B_1[z_1], ..., \gamma_1 B_m[z_m], ..., \gamma_1 A_n[y_n]\} \rightsquigarrow^*_\gamma \emptyset$, where $\kappa : \forall \underline{x}.\forall \underline{z}.B_1[z_m], ..., B_m[z_m] \Rightarrow C[f_\kappa(z_1, ..., z_m)] \in F(\Phi)$, and $C[f_\kappa(z_1, ..., z_m)] \sim_{\gamma_1} A_i[y_i]$. So we know $C \sim_{\gamma_1 - [f_\kappa(\underline{z})/y_i]} A_i$. Let $\gamma = \gamma_1 - [f_\kappa(\underline{z})/y_i]$. We have
$\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \rightsquigarrow \{\gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n\}$
$\equiv \{\gamma_1 A_1, ..., \gamma_1 B_1, ..., \gamma_1 B_m, ..., \gamma_1 A_n\}$. By IH, we know
$\Phi \vdash \{\gamma_1 A_1, ..., \gamma_1 B_1, ..., \gamma_1 B_m, ..., \gamma_1 A_n\} \rightsquigarrow^* \emptyset$.

E Proof of Theorem 9

Lemma 14. *If $\Phi \vdash \{D_1, ..., D_i, ..., D_n\} \rightarrow_{\kappa,\gamma} \{D_1, .., \sigma E_1, ..., \sigma E_m, ..., D_n\}$, with $\kappa : \forall \underline{x}.\underline{E} \Rightarrow C \in \Phi$ and $C \mapsto_\sigma D_i$ for any γ, then $\Phi \vdash \{D_1, ..., D_i, ..., D_n\} \rightsquigarrow_{\kappa,\gamma}$ $\{D_1, .., \sigma E_1, ..., \sigma E_m, ..., D_n\}$.*

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 $\kappa : \forall \underline{x}.\underline{E} \Rightarrow C \in \Phi$ and $C \mapsto_\sigma D_i$, we have $\Phi \vdash \{D_1, ..., D_i, ..., D_n\} \rightsquigarrow_{\kappa,\sigma\cdot\gamma}$ $\{\sigma D_1, .., \sigma E_1, ..., \sigma E_m, ..., \sigma D_n\}$. But $\mathrm{dom}(\sigma) \in \mathrm{FV}(C)$, thus we have
$\Phi \vdash \{D_1, ..., D_i, ..., D_n\} \rightsquigarrow_{\kappa,\gamma} \{D_1, .., \sigma E_1, ..., \sigma E_m, ..., D_n\}$.

Lemma 15.
Given Φ is non-overlapping, if $\Phi \vdash \{A_1, ..., A_n\}(\hookrightarrow_{\kappa,\gamma} \cdot \rightarrow^\mu_\gamma)\{C_1, ..., C_m\}$, then $\Phi \vdash \{A_1, ..., A_n\} \rightsquigarrow^*_\gamma \{C_1, ..., C_m\}$.

Proof. Given $\Phi \vdash \{A_1, ..., A_n\}(\hookrightarrow_{\kappa,\gamma} \cdot \rightarrow^\mu_\gamma)\{C_1, ..., C_m\}$, we know the actual reduction path must be of the form $\Phi \vdash \{A_1, ..., A_n\} \hookrightarrow_{\kappa,\gamma} \{\gamma A_1, ..., \gamma A_n\} \rightarrow_{\kappa,\gamma}$ $\{\gamma A_1, ..., \gamma B_1, ..., \gamma B_n, ..., \gamma A_n\} \rightarrow^\mu_\gamma \{C_1, ..., C_m\}$. Note that γ is unchanged along the term-matching reduction. The $\rightarrow$ following right after $\hookrightarrow$ can not use a different rule other than κ, it would mean $\gamma A_i \equiv \gamma C$ with $\kappa : \forall \underline{x}.\underline{B} \Rightarrow C \in \Phi$ and $A_i \equiv \sigma B$ with $\kappa' : \forall \underline{x}.\underline{D} \Rightarrow B \in \Phi$. This implies $\gamma C \equiv \gamma\sigma B$, contradicting the non-overlapping restriction. Thus we have $\Phi \vdash \{A_1, ..., A_n\} \rightsquigarrow_{\kappa,\gamma}$ $\{\gamma A_1, ..., \gamma B_1, ..., \gamma B_n, ..., \gamma A_n\}$. By Lemma 14, we have $\Phi \vdash \{A_1, ..., A_n\} \rightsquigarrow_{\kappa,\gamma}$ $\{\gamma A_1, ..., \gamma B_1, ..., \gamma B_n, ..., \gamma A_n\} \rightsquigarrow^*_\gamma \{C_1, ..., C_m\}$

Lemma 16. *Given Φ is non-overlapping, if $\Phi \vdash \{A_1, ..., A_n\}(\rightarrow^\mu \cdot \hookrightarrow^1)^*_\gamma\{C_1, ..., C_m\}$ with $\{C_1, ..., C_m\}$ in $\rightarrow^\mu \cdot \hookrightarrow^1$-normal form, then $\Phi \vdash \{A_1, ..., A_n\} \rightsquigarrow^*_\gamma \{C_1, ..., C_m\}$ with $\{C_1, ..., C_m\}$ in $\rightsquigarrow$-normal form.*

Proof. Since $\Phi \vdash \{A_1, ..., A_n\}(\rightarrow^\mu \cdot \hookrightarrow^1)^*_\gamma\{C_1, ..., C_m\}$, this means the reduction path must be of the form $\Phi \vdash \{A_1, ..., A_n\} \rightarrow^\mu \cdot \hookrightarrow^1 \cdot \rightarrow^\mu \cdot \hookrightarrow^1 \; ... \rightarrow^\mu \cdot \hookrightarrow^1$ $\cdot \rightarrow^\mu \{C_1, ..., C_m\}$. Thus $\Phi \vdash \{A_1, ..., A_n\} \rightarrow^\mu \cdot(\hookrightarrow^1 \cdot \rightarrow^\mu) \cdot (\hookrightarrow^1 \; ... \rightarrow^\mu) \cdot (\hookrightarrow^1$ $\cdot \rightarrow^\mu)\{C_1, ..., C_m\}$. By Lemma 14 and Lemma 15, we have $\Phi \vdash \{A_1, ..., A_n\} \rightsquigarrow^*_\gamma$ $\{C_1, ..., C_m\}$ with $\{C_1, ..., C_m\}$ in $\rightsquigarrow$-normal form.

Lemma 17. *Given Φ is a non-overlapping, if $\Phi \vdash \{A_1, ..., A_n\} \rightsquigarrow^*_\gamma \{C_1, ..., C_m\}$ with $\{C_1, ..., C_m\}$ in $\rightsquigarrow$-normal form , then $\Phi \vdash \{A_1, ..., A_n\}(\to^\mu \cdot \hookrightarrow^1)^*_\gamma\{C_1, ..., C_m\}$ with $\{C_1, ..., C_m\}$ in $\to^\mu \cdot \hookrightarrow^1$-normal form.*

Proof. By induction on the length of $\rightsquigarrow^*_\gamma$.

- Base Case: $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \rightsquigarrow_{\kappa,\gamma} \{\gamma A_1, ..., \gamma B_1, ..., \gamma B_m..., \gamma A_n\}$ with $\kappa : \forall \underline{x}.\ \underline{B} \Rightarrow C \in \Phi$, $C \sim_\gamma A_i$ and $\{\gamma A_1, ..., \gamma B_1, ..., \gamma B_m..., \gamma A_n\}$ in $\rightsquigarrow$-normal form . We have $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \hookrightarrow_{\kappa,\gamma} \{\gamma A_1, ..., \gamma A_i, ..., \gamma A_n\} \to_\kappa \{\gamma A_1, ..., \gamma B_1, ..., \gamma B_m..., \gamma A_n\}$ with $\{\gamma A_1, ..., \gamma B_1, ..., \gamma B_m..., \gamma A_n\}$ in $\to^\mu \cdot \hookrightarrow$-normal form. Note that there can not be another $\kappa' : \forall \underline{x}.\underline{B} \Rightarrow C' \in \Phi$ 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: $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \rightsquigarrow_{\kappa,\gamma} \{\gamma A_1, ..., \gamma B_1, ..., \gamma B_l, ..., \gamma A_n\} \rightsquigarrow^*_{\gamma'} \{C_1, ..., C_m\}$ with $\kappa : \forall \underline{x}.B_1, ..., B_l \Rightarrow C \in \Phi$ and $C \sim_\gamma A_i$.
  We have $\Phi \vdash \{A_1, ..., A_i, ..., A_n\} \hookrightarrow_{\kappa,\gamma} \{\gamma A_1, ..., \gamma A_i, ..., \gamma A_n\} \to \{\gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n\}$. By the non-overlapping requirement, there can not be another $\kappa' : \forall \underline{x}.\underline{D} \Rightarrow C' \in \Phi$ such that $\sigma C' \equiv A_i$.
  By IH, we know $\Phi \vdash \{\gamma A_1, ..., \gamma B_1, ..., \gamma B_m, ..., \gamma A_n\}(\to^\mu \cdot \hookrightarrow)^*_{\gamma'}\{C_1, ..., C_m\}$.
  Thus we conclude that $\Phi \vdash \{A_1, ..., A_i, ..., A_n\}(\hookrightarrow \cdot \to)^*_{\gamma'}\{C_1, ..., C_m\}$.

F Proof of Theorem 10

We assume a non-overlapping and productive program Φ in this section.

Lemma 18. *If $\Phi \vdash \{A_1, ..., A_n\} \rightsquigarrow \{B_1, ..., B_m\}$, then $\Phi \vdash \{A_1, ..., A_n\}(\to^\mu \cdot \hookrightarrow^1)^*\{C_1, ..., C_l\}$ and $\Phi \vdash \{B_1, ..., B_m\} \to^* \{C_1, ..., C_l\}$.*

Proof. Suppose $\Phi \vdash \{A_1, ..., A_n\} \rightsquigarrow_{\kappa,\gamma} \{\gamma A_1, ..., \gamma E_1, ..., \gamma E_l, ..., \gamma A_n\}$, with $\kappa : \underline{E} \Rightarrow D \in \Phi$ and $D \sim_\gamma A_i$. Suppose $D \not\mapsto_\gamma A_i$. In this case, we have $\Phi \vdash \{A_1, ..., A_n\} \hookrightarrow_{\kappa,\gamma} \cdot \to_{\kappa,\gamma} \{\gamma A_1, ..., \gamma E_1, ..., \gamma E_q, ..., \gamma A_n\} \to^\mu_\gamma \{C_1, ..., C_l\}$. Suppose $D \mapsto_\gamma A_i$, we have $\Phi \vdash \{A_1, ..., A_n\} \to_{\kappa,\gamma} \{\gamma A_1, ..., \gamma E_1, ..., \gamma E_q, ..., \gamma A_n\} \to^\mu_\gamma \{C_1, ..., C_l\}$.

Lemma 19. *If $\Phi \vdash \{A_1, ..., A_n\} \hookrightarrow_{\kappa,\gamma} \{\gamma A_1, ..., \gamma A_n\} \to^\mu_\gamma \{B_1, ..., B_m\}$, then $\Phi \vdash \{A_1, ..., A_n\} \rightsquigarrow^*_\gamma \{B_1, ..., B_m\}$.*

Proof. Suppose $\Phi \vdash \{A_1, ..., A_n\} \hookrightarrow_{\kappa,\gamma} \{\gamma A_1, ..., \gamma A_n\} \to^\mu_\gamma \{B_1, ..., B_m\}$, we have $\Phi \vdash \{A_1, ..., A_n\} \hookrightarrow_{\kappa,\gamma} \{\gamma A_1, ..., \gamma A_n\} \to_\kappa \{\gamma A_1, ..., \gamma C_1, ..., \gamma C_l.\gamma A_n\} \to^\mu_\gamma \{B_1, ..., B_m\}$ with $\kappa : \underline{C} \Rightarrow D \in \Phi$ and $D \sim_\gamma A_i$. Thus we have $\Phi \vdash \{A_1, ..., A_n\} \rightsquigarrow_{\kappa,\gamma} \{\gamma A_1, ..., \gamma C_1, ..., \gamma C_l, ..., \gamma A_n\}$. By Lemma 14, we have $\Phi \vdash \{A_1, ..., A_n\} \rightsquigarrow_{\kappa,\gamma} \{\gamma A_1, ..., \gamma C_1, ..., \gamma C_l, ..., \gamma A_n\} \rightsquigarrow^*_\gamma \{B_1, ..., B_m\}$.

Lemma 20. *If $\Phi \vdash \{A_1, ..., A_n\}(\to^\mu \cdot \hookrightarrow^1)^*_\gamma\{B_1, ..., B_m\}$, then $\Phi \vdash \{A_1, ..., A_n\} \rightsquigarrow^*_\gamma \{B_1, ..., B_m\}$.*

Proof. By Lemma 19.