A Practical View on Renaming

Marija Kulaš
FernUniversität in Hagen, Wissensbasierte Systeme, 58084 Hagen, Germany
kulas.marija@online.de

We revisit variable renaming from a practitioner’s point of view, presenting concepts we found useful in dealing with operational semantics of pure Prolog. A concept of relaxed core representation is introduced, upon which a concept of prenaming is built. Prenaming formalizes the intuitive practice of renaming terms by just considering the necessary bindings, where now some passive “bindings” x/x may be necessary as well. As an application, a constructive version of variant lemma for implemented Horn clause logic has been obtained. There, prenamings made it possible to incrementally handle new (local) variables.

1 Introduction

For logic program analysis or formal semantics, the issue of renaming variables and generally handling substitutions is inevitable. Yet the image of substitutions in logic programming research is a somewhat tainted one, at least since it has been pointed out by H.-P. Koller[16, p. 148] that the original claim of strong completeness of SLD-resolution needs to be amended, due to a counter-example using the fact that \( (x) \) is not more general than \( (x) \). The example may look counter-intuitive, but it complies with the definition of substitution generality. Also, by composing substitutions, properties like equivalence, idempotency or restriction are not preserved. Lastly, due to group structure of renamings, permuting any number of variables amounts to “doing nothing”, as in \((x) \sim (x)\), and such equivalences are also felt to be counter-intuitive. Hence the prevalent sentiments that substitutions are “a quite hard matter to deal with” ([13]) or “very tricky” ([16]). As a remedy, in the context of aggregating most general unifiers in a logic programming computation some helpful new concepts and operators were proposed, like parallel composition instead of traditional composition ([13]) and resultant instead of answer substitution ([12]).

Still, for almost anyone embarking on a journey of logic program analysis or formal semantics, sooner or later the need for renaming variables and generally handling substitutions in a new context arises.

In case of this author, the need arose while trying to prove adequacy of an operational semantics for pure Prolog, S1:PP [9], and the context was one of extensibility: Given is a pair of queries that are alphabetic variants of each other. As their respective S1:PP derivations proceed to develop, new variables may crop up, due to standardization-apart (here called local variables, Subsection 6.1), but the status of being variant should hold. This setup is known from the classical variant lemma ([111]). Additionally, the corresponding variables need to be collected, obtaining at each step the temporary variance between the derivations. As an example, assume the first query is \( p(z,u,x) \) and the second \( p(y,z,x) \). There is only one relevant renaming, \( \rho = (z \leftrightarrow y \leftrightarrow u \leftrightarrow y \leftrightarrow u) \). Now assume in the next step the first derivation acquires the variable \( y \), and the second \( w \). The relevant renaming this time would be \( \rho' = (z \leftrightarrow u \leftrightarrow w \leftrightarrow y \leftrightarrow w \leftrightarrow u) \). Clearly, \( \rho' \) is not an extension of \( \rho \), which makes it seem unsafe to proceed: are some properties of the previous step now in danger? So the question is, how to “safely” extend a variable mapping. For this purpose, in Section 5 we introduce a slight generalization of renaming, called prenaming. It is a mathematical underpinning...
of the intuitive practice of renaming terms by just considering the necessary bindings, and not worrying whether the result is a permutation. In the above example, renaming \( p(z, u, x) \) to \( p(y, z, x) \) means mapping \( z \mapsto y, u \mapsto z \) and \( x \mapsto x \). Intuitively, only \( z \mapsto y, u \mapsto z \) are considered necessary bindings, giving the "renaming" \( \left( \frac{zy}{xz} \right) \). For prenaming, \( x \mapsto x \) is necessary as well. It is based on relaxed core representation, which is nothing else than allowing some \( x \mapsto x \) pairs alongside "real" bindings, as placeholders.

Prenamings relate to and are inspired by previous work as follows. In [6], the concept of translation is defined, upon which alphabetic variance and standardization apart are built; this is the same as prenaming but for relaxed core (page 31). A safe prenaming is more general than renaming for a term from [11], and it maximizes \( W \) in the notion of W-renaming from [6] (page 33). Also, it generalizes substitution renaming from [1] (subsubsection 5.3.2).

In Section 6 prenamings are used to express and prove a propagation claim for implemented Horn clause logic, by means of local variable extension (Lemma 6.3). As a corollary, a variant lemma is obtained (Theorem 6.5). Underway, we touch on the discrepancy between the rather abundant theory of logic programming and a scarcity of mathematical claims for implemented logic programming systems. While there are some formal proofs of properties like nominal unification [7], for logic programming and a scarcity of mathematical claims for implemented logic programming systems.

### 2 Substitution

First we need a bit of notation. Assume two disjoint sets: a countably infinite set \( V \) of variables and a set \( \text{Fun} \) of shapes. If \( W \subseteq V \), any mapping \( F \) with \( F(W) \subseteq V \) shall be called variable-pure on \( W \). A mapping variable-pure on the whole set of variables \( V \) shall be simply called variable-pure. If \( V \setminus W \) is finite, \( W \) is said to be co-finite. A mapping \( F \) is injective on \( W \), if whenever \( F(x) = F(y) \) for \( x, y \in W \) also holds \( x = y \). Each \( f/n \in \text{Fun} \) consists of a functor \( f \) and an associated number of arguments \( n \), called arity. Functors of arity 0 are called constants. Starting from \( V \) and \( \text{Fun} \), data objects or terms\(^1\) are built: Any variable \( x \in V \) is a term. If \( t_1, \ldots, t_n \) are terms and \( f/n \in \text{Fun} \), then \( f(t_1, \ldots, t_n) \) is a term with shape \( f/n \) and constructor \( f \). In case of \( f/0 \), the term shall be written without parentheses. If a term \( s \) occurs within a term \( t \), we write \( s \in t \). The ordered pair of terms \( h \) and \( t \) is written as \( [h|t] \), where \( h \) is called the head and \( t \) the tail of the pair. A special case is a non-empty list, distinguished by its tail being a special term \( [] \) called the empty list, or a non-empty list itself. A list of \( n \) elements is the term \( [t_1][t_2][\ldots][t_n][[]][[[]]] \), conveniently written as \( [t_1, \ldots, t_n] \). Let \( \text{Vars}(t) \) be the set of variables in the term \( t \). If the terms \( s \) and \( t \) share a variable, that shall be written \( s \triangleright t \). Otherwise, we say \( s, t \) are variable-disjoint, written as \( s \nless t \).

A recurrent theme in this paper shall be "relevance", meaning "no extraneous variables" relative to some term or terms. It was used in [2] p.38 in unary sense, i.e. no extraneous variables relative to (one) term. This usage shall be reflected in the text as follows: A renaming \( \rho \) embedding a prenaming \( \alpha \) is a relevant embedding, if \( \text{Vars}(\rho) \subseteq \text{Vars}(\alpha) \) (Figure 1). Additionally, relevance in a binary sense, concerning two terms, shall also be needed: A mapping \( F \) is relevant for \( t_1 \) to \( t_2 \), if \( \text{Dom}(F) \subseteq \text{Vars}(t_1) \) and \( \text{Range}(F) \subseteq \text{Vars}(t_2) \) (Figure 2) (Lemma 6.3).

**Definition 2.1** (substitution). A substitution \( \theta \) is a function mapping variables to terms, which is identity almost everywhere. In other words, it is a function \( \theta \) with domain \( \text{Dom}(\theta) = V \) such that the set \( \text{Core}(\theta) := \{ x \in V \mid \theta(x) \neq x \} \) is finite\(^2\).

\(^1\)In Prolog, everything is a term, and so shall term be here the topmost syntactic concept.

\(^2\) [7] speaks of finite support.
The set $Core(\theta)$ shall be called the \textit{active domain} of $\theta$, and its elements \textit{active variables} of $\theta$. The set $\text{Ran}(\theta) := \theta(\text{Core}(\theta))$ is the \textit{active range} of $\theta$. A variable $x$ such that $\theta(x) = x$ shall be called a \textit{passive variable}, or a \textit{fixpoint}, for $\theta$. Also, we say that $\theta$ is active on the variables from $\text{Core}(\theta)$, and passive on all the other variables. If $\text{Core}(\theta) = \{x_1, \ldots, x_k\}$, where $x_1, \ldots, x_k$ are pairwise distinct variables, and $\theta$ maps each $x_j$ to $t_i$, then $\theta$ shall have the \textit{core representation} $\{x_1/t_1, \ldots, x_k/t_k\}$, or the perhaps more visual $\{x_1 \ldots x_k/t_1 \ldots t_k\}$. Each pair $x_i, t_i$ is called the \textit{binding} for $x_i$ in $\theta$, denoted by $x_i/t_i \in \theta$. Often we identify a substitution with its core representation, and thus regard it as a syntactical object, a term representing a finite set. So the set of variables of a substitution is defined as $\text{Vars}(\theta) := \text{Core}(\theta) \cup \text{Vars}(\text{Ran}(\theta))$.

The notions of restriction and extension of a mapping shall also be transported to core representation: if $\theta \subseteq \sigma$, we say $\theta$ is a \textit{restriction} of $\sigma$, and $\sigma$ is an \textit{extension} of $\theta$. The restriction $\theta|_W$ of a substitution $\theta$ on a set of variables $W \subseteq V$ is defined as follows: if $x \in W$ then $\theta|_W(x) := \theta(x)$, otherwise $\theta|_W(x) := x$. The restriction of $\theta$ upon the variables of $t$ is abbreviated as $\theta|_t := \theta|_{\text{vars}(t)}$.

The \textit{composition} $\theta \cdot \sigma$ of substitutions $\theta$ and $\sigma$ is defined by $(\theta \cdot \sigma)(x) := \theta(\sigma(x))$. Composition may be iterated, written as $\sigma^n := \sigma \cdot \sigma^{n-1}$ for $n \geq 1$, and $\sigma^0 := \varepsilon$. Here $\varepsilon := ()$ is the identity function on $V$. In case a variable-pure substitution $\rho$ is bijective, its inverse shall be denoted as $\rho^{-1}$. A substitution $\theta$ satisfying the equality $\theta \cdot \theta = \theta$ is called \textit{idempotent}.

Definition of substitution is enhanced from variables to arbitrary terms in a structure-preserving way by $\theta(f(t_1, \ldots, t_n)) := f(\theta(t_1), \ldots, \theta(t_n))$. If $t$ is a term, then $\theta(t)$ is an \textit{instance} of $t$ via $\theta$.

**Example 2.2.** $\left(\frac{x w u v}{u v x w}, \frac{u v x y z w}{x w y z w}\right) \cdot \left(\frac{\# f y z}{\# f y w}, \frac{\# f y x}{\# f y z}\right) = \left(\frac{\# f y z}{\# f y w}, \frac{\# f y x}{\# f y z}\right)$.

\section{Renaming}

**Definition 3.1** (renaming). A \textit{renaming} of variables is a bijective variable-pure substitution.

In \cite{6}, it is synonymously called "permutation". We shall reserve the word for the general case where movement of infinitely many variables is possible. Here we synonymously speak of \textit{finite permutation} due to the fact that, being a substitution, any renaming has a finite core, and \textit{Legacy 3.3} holds.

Due to structure preserving, if $s \in t$ then $\sigma(s) \in \sigma(t)$. For bijective substitutions (i.e. renamings), the converse property holds as well, giving

**Lemma 3.2** (renaming stability of "="", "\text{\in}", "\not\in"). Let $\rho$ be a renaming and $s, t$ be terms. Then $s = t$ iff $\rho(s) = \rho(t)$, and also $s \in t$ iff $\rho(s) \in \rho(t)$. As a consequence, $s \not\in t$ iff $\rho(s) \not\in \rho(t)$.

**Legacy 3.3** (\cite{10}). A substitution $\rho$ is a renaming iff $\rho(\text{Core}(\rho)) = \text{Core}(\rho)$.

**Legacy 3.4** (\cite{6}). Every injective variable-pure substitution is a renaming.

So composition of renamings is a renaming. The next property is about cycle decomposition of a finite permutation.

**Lemma 3.5** (cycles). Let $\sigma$ be a variable-pure substitution. It is injective iff for every $x \in V$ there is $n \in \mathbb{N}$ such that $\sigma^n(x) = x$.

**Proof.** Assume $\sigma$ injective, and choose $x_0 \in V$. If $\sigma(x_0) = x_0$, we are done. Otherwise, $\sigma^i(x_0) \neq \sigma^{i-1}(x_0)$ for all $i \geq 1$, due to injectivity. Hence, $\sigma^{i-1}(x_0) \in \text{Core}(\sigma)$ for every $i \geq 1$. Because of the finiteness of $\text{Core}(\sigma)$, there is $m > k \geq 1$ such that $\sigma^m(x_0) = \sigma^k(x_0)$. Due to injectivity, $\sigma^{m-1}(x_0) = \sigma^{k-1}(x_0)$. By

\[3\text{Traditionally called just \textit{domain}. This may be confusing, since in the usual mathematical sense it is always the whole $V$ that is the domain of any substitution.}\]

\[4\text{The name \textit{active variable} appears in [8].}\]
iteration we get \( n := m - k \). For the other direction, assume \( \sigma(x) = \sigma(y) \), and minimal \( m,n \) such that \( \sigma^n(x) = x, \sigma^m(y) = y \). Consider the case \( m \neq n \), say \( m > n \). Then \( \sigma^{m-n}(y) = \sigma^{m-n}(\sigma^n(x)) = \sigma^{m-n}(\sigma^m(y)) = \sigma^m(y) = y \), contradicting minimality of \( m \). Hence \( m = n \), so \( x = \sigma^n(x) = \sigma^n(y) = y \). \( \diamond \)

4 Relaxed core representation

If there is a substitution \( \sigma \) mapping a term \( s \) on a term \( t \), then it is mapping each variable in \( s \) on a subterm of \( t \). It is possible that a variable stays the same, so if we want our mapping to explicitly cover all variables in \( s \), as in the promised application (Section 6), then necessarily \( x/x \) would have to be tolerated as a "binding".

To cater for such wishes, the core of the substitution \( \sigma \) can be relaxed to contain some passive variables, raising those above the rest, as it were. This simple technique is useful beyond the context of renaming, so we assume arbitrary substitutions.

**Definition 4.1** (relaxed core). If \( \text{Core}(\sigma) \subseteq \{x_1, \ldots, x_n\} \), where variables \( x_1, \ldots, x_n \) are pairwise distinct, then \( \{x_1, \ldots, x_n\} \) shall be called a relaxed core and \( \left( \frac{x_1}{\sigma(x_1)} \ldots \frac{x_n}{\sigma(x_n)} \right) \) shall be called a relaxed core representation for \( \sigma \). If we fix a relaxed core for \( \sigma \), it shall be denoted \( \text{C}(\sigma) := \{x_1, \ldots, x_n\} \). The associated range \( \sigma(\text{C}(\sigma)) \) we denote as \( R(\sigma) \). The set of variables of \( \sigma \) is as expected, \( \text{V}(\sigma) := \text{C}(\sigma) \cup \text{Vars}(R(\sigma)) \). To get back to the traditional representation, we denote by \([\sigma]\) the (non-relaxed) core representation of \( \sigma \).

For extending, substitutions are treated like sets of active bindings, so (disjoint) union may be used:

**Definition 4.2** (sum of substitutions). If \( \sigma = \left( \frac{x_1}{s_1} \ldots \frac{x_n}{s_n} \right) \) and \( \theta = \left( \frac{y_1}{t_1} \ldots \frac{y_m}{t_m} \right) \) are substitutions in relaxed representation such that \( \{y_1, \ldots, y_m\} \nsubseteq \{x_1, \ldots, x_n\} \), then \( \sigma \cup \theta := \left( \frac{x_1}{s_1} \ldots \frac{x_n}{s_n} \frac{y_1}{t_1} \ldots \frac{y_m}{t_m} \right) \) is the sum of \( \sigma \) and \( \theta \).

For Subsection 6.2, backward compatibility of an extension shall be needed.

**Lemma 4.3** (backward compatibility). Let \( \sigma, \theta \) be substitutions and \( x \) be a variable. Then \( (\sigma \cup \theta)(x) = \sigma(x) \) iff \( \theta(x) = x \).

**Proof.** If \( x \notin \text{C}(\theta) \), then \( \theta(x) = x \), and \( (\sigma \cup \theta)(x) = \sigma(x) \). If \( x \in \text{C}(\theta) \), then \( (\sigma \cup \theta)(x) = \theta(x) \) and also \( x \notin \text{C}(\sigma) \), hence \( \sigma(x) = x \). The condition \( (\sigma \cup \theta)(x) = \sigma(x) \) collapses to \( \theta(x) = x \). \( \diamond \)

Passivity of \( \theta \) on a term \( t \) is guaranteed if \( \sigma \) is "complete" for \( t \), i.e. lays claim to all its variables:

**Definition 4.4** (complete for term). Let \( \sigma \) be given in relaxed core representation. We say that \( \sigma \) is complete for \( t \) if \( \text{Vars}(t) \subseteq \text{C}(\sigma) \).

In such a case there is no danger that an extension of \( \sigma \) might map \( t \) differently from \( \sigma \):

**Corollary 4.5** (backward compatibility). If \( \sigma \) is complete for \( t \), then for any \( \theta \) holds: \( \sigma \cup \theta \) is complete for \( t \) and \( (\sigma \cup \theta)(t) = \sigma(t) \).

5 Prenaming

In practice, one would like to change the variables in a term without bothering to check whether this change is a permutation of variables, i.e. a renaming in the sense of [Definition 3.1]. For example, the term \( p(z, u, x) \) can be changed to \( p(y, z, x) \) using mapping \( z \mapsto y, u \mapsto z, x \mapsto x \). Let us call such a mapping pre-renaming.\(^5\)

\(^5\)Finding an appropriate name can be a struggle. Shortlisted were pre-renaming and proto-renaming.
Like any substitution, a prenaming $\alpha$ shall also be represented finitely, but in relaxed core representation, in order to capture possible $x \mapsto x$ pairings. The set $C(\alpha)$ is fixed by the terms to map. Obviously, injectivity is important for such a mapping, since $p(z,u,x)$ cannot be mapped on $p(y,y,x)$ without losing a variable. Hence,

**Definition 5.1** (prenaming). A prenaming $\alpha$ is a variable-pure substitution injective on a finite set of variables $C(\alpha) \supseteq \text{Core}(\alpha)$.

Clearly, any renaming is a prenaming. For **Theorem 6.5** we need to extend a given prenaming.

**Lemma 5.2** (extension of prenaming). Let $\alpha = \left( x_1 \ldots x_n \right)$ and $\beta = \left( u_1 \ldots u_m \right)$ be prenamings such that $\{u_1, \ldots, u_m\} \not\subseteq \{x_1, \ldots, x_n\}$ and $\{v_1, \ldots, v_m\} \not\subseteq \{y_1, \ldots, y_n\}$. Then $\alpha \uplus \beta = \left( x_1 \ldots x_n u_1 \ldots u_m \right)$ is also a prenaming, with $C(\alpha \uplus \beta) = C(\alpha) \uplus C(\beta)$ and $R(\alpha \uplus \beta) = R(\alpha) \uplus R(\beta)$.

Plotkin’s concept of translation [14, p. 46] corresponds to prenaming without passive bindings. There, the inverse translation for $\tau := \left( \begin{smallmatrix} u & v \\ w & z \end{smallmatrix} \right)$ would be $\tau^{-1} := \left( \begin{smallmatrix} y & z \\ u & v \end{smallmatrix} \right)$. Clearly, $\tau^{-1} \cdot \tau = \left( \begin{smallmatrix} y & z \\ u & v \end{smallmatrix} \right) \cdot \left( \begin{smallmatrix} z & u \\ y & v \end{smallmatrix} \right) = \left( \begin{smallmatrix} y & z \\ u & v \end{smallmatrix} \right)$, which is not identity substitution. Although $(\tau^{-1} \cdot \tau)|\{u,v\} = \varepsilon$, so $\tau$ is reversible and thus ”safe” to use on $\{u,v\}$, one might instinctively be wary of the possibility that handling several translations in the same computation could somehow produce ”unsafety”. Presumably for that reason, the concept of translation did not catch on, and it is meanwhile customary to define alphabetic variance using renaming rather than translation (2). We revisit Plotkin’s concept, enriched with passive bindings and deemed fit for a new name, prenaming, and show that its safe application on a term and safe (even backward-compatible) extension are easily achievable, thus justifying the intuitive practice.

### 5.1 The question of inverse

So a prenaming is more natural in practice, but a ”full” renaming is better mathematically tractable, due to its being invertible on $V$. The next property shows how to extend a prenaming $\alpha$ to obtain a renaming, and a relevant one at that, i.e. active only on the variables from $V(\alpha)$. The claim is essentially given in 12, 2 and 1 with emphasis on the core of such an extension. Originally the claim appears in 6, with emphasis on the extent of coincidence which is our concern as well. We rephrase the claim around the notion of prenaming, and provide a constructive proof based on **Lemma 3.5**.

**Theorem 5.3** (embedding). If $\alpha$ is a prenaming, there is a renaming $\overline{\alpha}$ which coincides with $\alpha$ on $V \setminus (R(\alpha) \setminus C(\alpha))$ such that $\text{Vars}(\overline{\alpha}) \subseteq V(\alpha)$. Additionally, if $\alpha(x) \neq x$ on $C(\alpha)$, then $\overline{\alpha}(x) \neq x$ on $V(\alpha)$.

$$
\begin{align*}
\overline{\alpha}(x) := \begin{cases}
\alpha(x), & \text{if } x \in C(\alpha) \\
 z, & \text{if } x \in R(\alpha) \setminus C(\alpha) \text{ and } \alpha^m(z) = x \text{ for maximal } m \leq n \\
x, & \text{outside of } C(\alpha) \cup R(\alpha)
\end{cases}
\end{align*}
$$

Figure 1: Closure, the natural relevant embedding

---

6 [2] p. 23: "Every finite 1-1 mapping $f$ from $A$ onto $B$ can be extended to a permutation $g$ of $A \cup B$. Moreover, if $f$ has no fixpoints, then it can be extended to a $g$ with no fixpoints."

6 [6] p. 35: "Let $W$ be a co-finite set of variables (...) and let $\sigma$ be a $W$-renaming. Then there is a permutation $\pi$ which coincides with $\sigma$ on the set $W"
Proof. If \( \alpha \) is a prenaming, then \( C(\alpha) \) and \( R(\alpha) \) are sets of \( n \) distinct variables each. The wanted renaming is constructed in Figure 1 with the intention to close the possibly open chain \( x, \alpha(x), \alpha^2(x), \ldots \). So let us see whether for every \( x \) there is a \( j \) such that \( \overline{\alpha}(x) \). If \( x \in C(\alpha) \), we start as in the proof of Lemma 3.5 and consider the sequence \( x, \alpha(x), \alpha^2(x), \ldots \). Since \( C(\alpha) \) is finite, either we get two equals (and proceed as there), or we get \( \alpha^k(x) \) we know \( \overline{\alpha}(y) = z \) such that \( \alpha^m(z) = y \) with maximal \( m \), so \( m \geq k \). Therefore, \( \alpha^m(\overline{\alpha}(y)) = y = \alpha^k(x) \). Due to injectivity of \( \alpha \) on \( C(\alpha) \), we get \( \alpha^{m-k}(\overline{\alpha}(\alpha^k(x))) = x \), and hence \( \overline{\alpha}(\alpha^k(x)) = x \).

The cases \( x \in R(\alpha) \setminus C(\alpha) \) or \( x \not\in C(\alpha) \cup R(\alpha) \) are easy. By Lemma 3.5 \( \overline{\alpha} \) is injective. By Legacy 3.4 \( \overline{\alpha} \) is a renaming. The discussion of the case \( \alpha(x) \neq x \) on \( C(\alpha) \) is straightforward. \( \diamond \)

Definition 5.4 (closure of a prenaming). The renaming \( \overline{\alpha} \) constructed in Figure 1 shall be called the closure of \( \alpha \).

Remark 5.5 (relevant embedding is not unique). Let \( \alpha = \left( \frac{z u y w_1}{y z x w_2} \right) \), and let us embed it in a relevant renaming. The Figure 1 gives \( \overline{\alpha} = \left( \frac{z u y w_1 x w_2}{y z x w_2 u w_1} \right) \). But \( \rho = \left( \frac{z u y w_1 x w_2}{y z x w_2 u w_1} \right) \) is also a relevant renaming which is embedding \( \alpha \). In the usual notation for cycle decomposition, \( \rho = \{(x, w_1, w_2, u, z, y)\} \) and \( \overline{\alpha} = \{(x, u, z, y), (w_1, w_2)\} \).

If we reverse the prenaming, the closure algorithm shall be closing the same open chains but in the opposite direction, hence

Lemma 5.6 (reverse prenaming). Let \( \alpha := \left( \frac{x_1 \ldots x_n}{y_1 \ldots y_n} \right) \) and \( \alpha_{\text{inv}} := \left( \frac{y_1 \ldots y_n}{x_1 \ldots x_n} \right) \). Then \( \overline{\alpha_{\text{inv}}} = \overline{\alpha}^{-1} \).

Remark 5.7 (closure is not compositional). Take \( \alpha := \left( \frac{z u y}{x z y} \right) \) and \( \rho := \left( \frac{x y}{x y} \right) \). Then \( \overline{\alpha} = \left( \frac{z u x y}{x z y u} \right) \), \( \overline{\rho} \cdot \overline{\alpha} = \left( \frac{z u x y}{x z y u} \right) \) and \( \overline{\rho} \cdot \overline{\alpha} = \left( \frac{z u x y}{x z y u} \right) \).

Remark 5.8 (closure is not monotone). If \( \alpha \geq \alpha' \), then not always \( \overline{\alpha} \geq \overline{\alpha'} \). To see this, let \( \alpha = \left( \frac{z u y}{x z u} \right) \) and \( \alpha' = \left( \frac{z u y}{x z u} \right) \). Then \( \overline{\alpha} = \left( \frac{z u y}{x z u} \right) \) and \( \overline{\alpha} = \left( \frac{z u y}{x z u} \right) \).

5.2 Staying safe

Let us look more closely into Remark 5.8 \( \alpha(y) = x \) and \( \alpha(x) = x \), so \( y \) and \( x \) may not simultaneously occur in the candidate term. Otherwise, a variable shall be lost, which we call "aliasing", like in \( \left( \frac{y}{x} \right) \) \( p(x, f(y)) = p(x, f(x)) \).

Definition 5.9 (aliasing). Let \( \alpha \) be a prenaming. If \( x \neq y \) but \( \alpha(x) = \alpha(y) \), then \( \alpha \) is aliasing \( x \) and \( y \).

So what Remark 5.8 means is: if we want to use \( \alpha \) on a larger set than \( C(\alpha) \), then the set \( \text{Pit}(\alpha) := R(\alpha) \setminus C(\alpha) \) should be avoided, because aliasing may happen. But, luckily, its complement is safe:

Lemma 5.10 (larger set). A prenaming \( \alpha \) is injective on the co-finite set \( V \setminus \text{Pit}(\alpha) \). The set is maximal containing \( C(\alpha) \).

Proof. Let \( x, y \in V \setminus \text{Pit}(\alpha) \). Is it possible that \( \alpha(x) = \alpha(y) \)? Possible cases: If \( x, y \in C(\alpha) \), then by definition of prenaming \( \alpha(x) \neq \alpha(y) \). If \( x, y \not\in C(\alpha) \), then \( \alpha(x) = x \neq y = \alpha(y) \). It remains to consider the mixed case \( x \in C(\alpha), y \not\in C(\alpha) \). We have \( \alpha(x) \in R(\alpha) \) and \( \alpha(y) = y \). So is \( \alpha(x) = y \) possible? If yes, then \( y \in R(\alpha) \), but since \( y \not\in C(\alpha) \), that would mean \( y \in \text{Pit}(\alpha) \). Contradiction.

The set cannot be made larger: if \( y \in \text{Pit}(\alpha) \), then there is \( x \in C(\alpha) \) with \( x \neq y \) and \( \alpha(x) = y = \alpha(y) \). \( \diamond \)
Definition 5.11 (injectivity domain). Since $\text{InDom}(\alpha) := V \setminus \text{Pit}(\alpha)$ is the largest co-finite set containing $C(\alpha)$ on which $\alpha$ is injective, it shall be called the injectivity domain of $\alpha$.

The injectivity domain of a prenaming is clearly the only safe place for it to be mapping terms from.

Definition 5.12 (safety of prenaming). A prenaming $\alpha$ is safe for a term $t$ if $\text{Vars}(t) \subseteq \text{InDom}(\alpha)$.

Clearly, $\text{InDom}(\alpha) = C(\alpha) \cup \left(V \setminus \text{R}(\alpha)\right)$, so $\alpha$ is safe for its relaxed core. Hence,

Corollary 5.13 (complete and safe). If a prenaming is complete for a term, it is safe for that term.

For a prenaming $\alpha$ with the quality $R(\alpha) = C(\alpha)$, i.e. a renaming, it is no surprise that $\text{InDom}(\alpha) = V$ and hence safety is guaranteed for any term.

A prenaming behaves like a renaming on its injectivity domain, since it coincides with its closure there. This follows immediately from Theorem 5.3.

Lemma 5.14 (injectivity domain). Let $x \in \text{InDom}(\alpha)$. Then $\alpha(x) = \overline{x}(x)$.

Corollary 5.15 (prenaming stability). A generalization of Lemma 3.2 holds: Let $s, t$ be terms and $\alpha$ be a prenaming safe for $s, t$. Then $s = t$ iff $\alpha(s) = \alpha(t)$ and also $s \in t$ iff $\alpha(s) \in \alpha(t)$. As a consequence, $s \not\in t$ iff $\alpha(s) \not\in \alpha(t)$.

Our definition of prenaming was inspired by the following more general notion from [6].

Definition 5.16 (W-renaming, [5]). Let $W \subseteq V$. A substitution $\sigma$ is a W-renaming if $\sigma$ is variable-pure on $W$, and $\sigma$ is injective on $W$.

With this notion, Lemma 5.10 can be summarized as: $\text{InDom}(\alpha)$ is a co-finite set of variables, and the largest set $W \supseteq C(\alpha)$ such that $\alpha$ is a W-renaming.

What about safety of extension? If $\alpha$ is safe for $t$, $\alpha \uplus \beta$ does not have to be, even if $\beta(t) = t$, as the following example shows: $\alpha := \left(\kappa^z_y\right), \beta := \left(\zeta^u_y\right), t := p(x)$ (here no aliasing happened, though). The next two claims address safety of extension.

Lemma 5.17 (monotonicity). Assume $\alpha \uplus \beta$ is defined. Then

1. $\text{InDom}(\alpha) \cup \text{InDom}(\beta) = V$
2. $\text{InDom}(\alpha) \cap \text{InDom}(\beta) \subseteq \text{InDom}(\alpha \uplus \beta)$

Proof. Since $(V \setminus A) \cup (V \setminus B) = V \setminus (A \cup B)$, and $\text{Pit}(\alpha) \not\subseteq \text{Pit}(\beta)$, we get $\text{InDom}(\alpha) \cup \text{InDom}(\beta) = V$. Further, $(V \setminus A) \cap (V \setminus B) = V \setminus (A \cap B)$ and so $\text{Pit}(\alpha \uplus \beta) = \left(R(\alpha) \uplus R(\beta)\right) \setminus (C(\alpha) \uplus C(\beta)) \subseteq \left(R(\alpha) \setminus C(\alpha)\right) \cup \left(R(\beta) \setminus C(\beta)\right) = \text{Pit}(\alpha) \cup \text{Pit}(\beta)$. \hfill $\Diamond$

In Remark 5.3 $\text{Pit}(\alpha') = \{y\}$, $\text{Pit}(\beta') = \{x\}$, and $\text{Pit}(\alpha) = \{x\}$, hence $\text{InDom}(\alpha') = V \setminus \{y\}$, $\text{InDom}(\beta') = V \setminus \{x\}$ and $\text{InDom}(\alpha) = V \setminus \{x\}$.

By the last claim, staying within $\text{InDom}(\alpha)$ and $\text{InDom}(\beta)$ ensures staying within $\text{InDom}(\alpha \uplus \beta)$. By assuming a bit more about $\alpha$ than just safety, we may ignore the nature of extension $\beta$, and still ensure safety and even backward compatibility of $\alpha \uplus \beta$. This shall be used in Section 6.

Theorem 5.18 (safety of extension). Assume $\alpha \uplus \beta$ is defined.

1. If $\alpha$ is safe for $t$ and $\beta$ is safe for $t$, then $\alpha \uplus \beta$ is safe for $t$.
2. If $\alpha$ is complete for $t$, then $\alpha \uplus \beta$ is complete (hence safe) for $t$, and $(\alpha \uplus \beta)(t) = \alpha(t)$.

The first part follows from Lemma 5.17 and the second from Corollary 4.5 and Corollary 5.13.

\footnote{Safe prenaming is more general than renaming for a term in [II] p. 22, since we do not require $\text{Core}(\alpha) \subseteq \text{Vars}(t)$.}
5.3 Variant of term and substitution

The traditional notion of term variance, which is term renaming, shall be generalized to prenaming. As a special case, substitution variance is defined, inspired by substitution renaming from \cite{1}. For this, substitution shall once again be regarded as a special case of term. The term is of course the relaxed core representation. This concept shall come in handy for proving properties of renamed derivations (Subsection 6.2).

5.3.1 Term variant

**Definition 5.19** (term variant). If \( \alpha \) is a prenaming safe for \( t \), then \( \alpha(t) \) is a variant of \( t \), written \( \alpha(t) \cong t \). The particular variance and the direction of its application may be explicated by \( s =_\alpha t \) if \( s = \alpha(t) \).

If \( s \cong t \), then there is a unique \( \alpha \) mapping \( s \) to \( t \) in a complete and relevant manner, i.e. mapping each variable pair and nothing else, as computed by Figure 2. The algorithm makes do with only one set for equations and bindings, thanks to different types. Termination can be seen from the tuple \((l_{\text{fun}}(E), \text{card}(E))\) decreasing in lexicographic order with each rule application, where \( l_{\text{fun}}(E) \) is the number of function symbols in equations in \( E \), and \( \text{card}(E) \) is the number of equations in \( E \).

Start from the set \( E := \{ s = t \} \) and transform according to the following rules. The transformation is bound to stop. If the stop was not due to failure, then the final set \( E \) is the prenaming of \( s \) to \( t \), \( \text{Pren}(s,t) \).

- **elimination** \( E \uplus \{ x = y \} \leadsto E \), if \( x/y \in E \)
- **failure: alias** \( E \uplus \{ x = y \} \leadsto \text{failure} \), if \( (x/z \in E, z \neq y) \) or \( (z/y \in E, z \neq x) \)
- **binding** \( E \uplus \{ x = y \} \leadsto E \cup \{ x/y \} \), if \( (x/y \notin E) \) and \( (z/y \notin E) \)
- **failure: instance** \( E \uplus \{ x = t \} \leadsto \text{failure} \), if \( t \notin V \); \( E \uplus \{ t = x \} \leadsto \text{failure} \), if \( t \notin V \)
- **decomposition** \( E \uplus \{ f(s_1,\ldots,s_n) = f(t_1,\ldots,t_n) \} \leadsto E \cup \{ s_1 = t_1,\ldots,s_n = t_n \} \)
- **failure: clash** \( E \uplus \{ f(s_1,\ldots,s_n) = g(t_1,\ldots,t_m) \} \leadsto \text{failure} \), if \( f \neq g \) or \( m \neq n \)

**Figure 2:** Computing the prenaming of \( s \) to \( t \)

**Notation 5.20** (epsoid). The prenaming constructed in Figure 2 shall be simply called the prenaming of \( s \) to \( t \), and denoted \( \text{Pren}(s,t) \). It is complete for \( s \) and relevant for \( s \) to \( t \).

In case \( s = t \), we obtain for \( \text{Pren}(s,t) \) essentially the identity substitution. However, regarded as prenaminings, \( \text{Pren}(t,t) \) and \( \varepsilon \) are not the same. A prenaming \( \alpha \) with relaxed core \( W \) mapping each variable on itself (in other words, \( C(\alpha) = W \) and \( [\alpha] = \varepsilon \)) shall be called the \( W \)-epsoid and denoted \( \varepsilon_W \).

For a term \( t \), we abbreviate \( \varepsilon_t := \varepsilon_{\gamma_{\text{var}}(t)} \).

Regarding composition, an epsoid behaves just like \( \varepsilon \). Its use is for providing completeness, and hence extensibility, by means of placeholder bindings \( x/x \).

5.3.2 Special case: substitution variant

Even substitutions themselves can be renamed. To rename a substitution, one regards it as a syntactical object, a set of bindings, and renames those bindings. If \( \rho \) is a renaming and \( \sigma \) is a substitution, \( \rho \circ \sigma \) is a renaming. However, regarded as substitutions, \( \rho \circ \sigma \) and \( \sigma \) are not the same. A renaming \( \rho \) with relaxed core \( W \) mapping each variable on itself (in other words, \( C(\rho) = W \) and \( [\rho] = \varepsilon \)) shall be called the \( W \)-epsoid and denoted \( \varepsilon_W \).

Regarding composition, an epsoid behaves just like \( \varepsilon \). Its use is for providing completeness, and hence extensibility, by means of placeholder bindings \( x/x \).

---

\(^9\)“Relevant” in the binary sense (page 28). In case of prenaming, we naturally use \( C \) as \( \text{Dom} \) and \( R \) as \( \text{Range} \).
defines substitution renaming by \( \rho(\sigma) := \{ \rho(x)/\rho(\sigma(x)) \mid x \in \text{Core}(\sigma) \} \). It is easy to see that \( \rho(\sigma) \) is a substitution in core representation. For this only two properties of \( \rho \) were needed: variable-purity on \( \text{Vars}(\sigma) \) and injectivity on \( \text{Vars}(\sigma) \). These requirements are clearly fulfilled by prenamings safe on \( \sigma \) as well. Hence,

**Definition 5.21** (substitution variant). Let \( \sigma \) be a substitution and let \( \alpha \) be a prenaming safe for \( \sigma \), i.e. \( \text{Vars}(\sigma) \subseteq \text{InDom}(\alpha) \). Then a **variant** of \( \alpha \) by \( \alpha \) is

\[
\alpha(\sigma) := \{ \alpha(x)/\alpha(\sigma(x)) \mid x \in \text{Core}(\sigma) \}
\]

We may write \( \theta =_{\alpha} \sigma \) if \( \theta = \alpha(\sigma) \), as with any other terms. As can be expected, the concept of variance by prenaming is well-defined, owing to safety. Otherwise, the result of prenaming would not even have to be a substitution again, as in the case of \( \alpha = \{ x \} \) and \( \sigma = (x_a) \).

**Lemma 5.22** (well-defined). **Substitution variant is well-defined**, i.e. \( (\ ) \) is a core representation of a substitution, and \( \alpha \) does not introduce aliasing.

**Proof.** Let \( \text{Core}(\sigma) = \{ x_1, \ldots, x_n \} \). Due to injectivity of \( \alpha \) on \( \text{Vars}(\sigma) \), if \( \alpha(x_i) = \alpha(x_j) \), then \( x_i = x_j \), so \( i = j \). To finish the proof that \( (\ ) \) a core representation, observe \( x \in \text{Core}(\sigma) \) iff \( x \neq \sigma(x) \) iff \( \alpha(x) \neq \alpha(\sigma(x)) \), due to injectivity again. Re aliasing, by **Corollary 5.15** if \( \alpha(\sigma(x_i)) = \alpha(\sigma(x_j)) \), then \( \sigma(x_i) \sim_{\infty} \sigma(x_j) \), meaning that \( \alpha \) does not introduce aliasing. \( \diamond \)

From [Definition 5.21] and [Lemma 5.14] follows

**Lemma 5.23.** Let \( \sigma \) be a substitution, \( \alpha, \beta \) be prenamings and \( \alpha(\sigma) \) and \( (\alpha \cdot \beta)(\sigma) \) be defined. Then

1. \( (\alpha \cdot \beta)(\sigma) = \alpha(\beta(\sigma)) \)
2. \( \alpha(\sigma) = \overline{\alpha}(\sigma) \)

For the case of “full” renaming, there is a way to dissolve the new expression\(^{10}\)

**Legacy 5.24** (\( [\ ) \]). **For any renaming** \( \rho \) and substitution \( \sigma \)

\[
\rho(\sigma) = \rho \cdot \sigma \cdot \rho^{-1}
\]

Would such a claim hold for the weakened case, prenamings?

**Theorem 5.25** (substitution variant). Let \( \sigma \) be a substitution and \( \alpha \) be a prenaming safe for \( \sigma \). Then

1. \( \alpha(\sigma) \cdot \alpha = \alpha \cdot \sigma \)
2. \( \alpha(\sigma) = \overline{\alpha}(\sigma) \cdot \alpha \cdot \overline{\sigma}^{-1} \)

**Proof.** First part: According to [Definition 5.21] for every \( x \in V \) holds \( (\alpha(\sigma) \cdot \alpha)(x) = \alpha(\sigma(x)) \). Since any substitution is structure-preserving, the claim holds for any term \( t \) as well. Second part: From the first part we know \( \overline{\alpha}(\sigma) \cdot \alpha = \overline{\alpha} \cdot \sigma \), hence \( \overline{\alpha}(\sigma) = \overline{\alpha} \cdot \sigma \cdot \overline{\sigma}^{-1} \). By [Lemma 5.14] \( \alpha(\sigma) = \overline{\alpha}(\sigma) \).

It is known that idempotence and equivalence of substitutions are not compatible with composition\(^{6}\). Luckily, the concept of variance, with constant prenaming, does not share this handicap:

**Theorem 5.26** (compositionality). Let \( \sigma, \theta \) be substitutions and \( \alpha \) be their safe prenamings. Then

\[
\alpha(\sigma \cdot \theta) = \alpha(\sigma) \cdot \alpha(\theta)
\]

**Proof.** Since \( \text{Vars}(\sigma \cdot \theta) \subseteq \text{Vars}(\sigma) \cup \text{Vars}(\theta) \), clearly \( \text{Vars}(\sigma \cdot \theta) \subseteq \text{InDom}(\alpha) \). By [Theorem 5.25] \( \alpha(\sigma) \cdot \alpha(\theta) = \overline{\alpha} \cdot \sigma \cdot \overline{\sigma}^{-1} \cdot \alpha \cdot \theta \cdot \overline{\alpha}^{-1} = \overline{\alpha} \cdot \sigma \cdot \theta \cdot \overline{\alpha}^{-1} = \alpha(\sigma \cdot \theta) \).

\( ^{10} \) an immediate consequence being \( \rho(\sigma) \neq \rho \cdot \sigma \)
6 Application

Implementing logic programming means that the freedom of Horn clause logic (HCL) must be restrained:

- most general unifier (mgu) is provided by a fixed algorithm,
- standardization-apart is provided by a fixed algorithm.

Every implementation of HCL is parametrized by the two algorithms. Here we shall consider only the unification algorithm, so by HCL\((U)\) an implementation of HCL using unification algorithm \(U\) is denoted. From the literature (variant lemma) we know that such a restriction is not compromising soundness and completeness of SLD-resolution. Yet, there may be lots of “lowlier” claims which more or less implicitly rely on freedom of mgu and standardization-apart. For example, with both choices fixed we may not any more just rename an SLD-derivation wholesale (the resolvers, the mgu’s, the input clauses), as was possible in Horn clause logic, based on Lemma 3.2. This is because the two algorithms do not have to be renaming-compatible. In fact, the second one cannot be, which makes claims like Lemma 6.3 necessary.

Let us cast a look at the first restriction. For any two unifiable terms \(s, t\) holds that the set of their mgu’s, written as \(Mgus(s,t)\), is infinite. On the other hand, in practice any unification algorithm \(U\) produces, for the given two unifiable terms, just one deterministic value as their mgu. We shall denote this particular mgu of \(s\) and \(t\) as \(U(s,t)\), the algorithmic mgu of \(s\) and \(t\) obtained by \(U\).

The abundancy of mgu’s is not only good, it also stands in the way of proofs. The simplest unification problem \(p(x) = p(y)\) has among others two equally attractive candidate mgu’s, \((\theta_y)\) and \((\theta_z)\). Assume our unification algorithm decided upon \((\theta_y)\). Assume further that we rename the protagonists and obtain \(p(x) = p(z)\). What mgu shall be chosen this time? To ensure some dependability in this issue, we shall require of any unification algorithm the following simple requirement, postulated as an axiom:

**Axiom 6.1** (renaming compatibility). Let \(U\) be a unification algorithm. For any renaming \(\rho\) and any equation \(E\), it has to hold \(U(\rho(E)) = \rho(U(E))\).

Since classical unification algorithms like Robinson’s and Martelli-Montanari’s do not depend upon the actual names of variables (as observed in [11]), this requirement is in practice always satisfied.

**Remark 6.2** (renaming compatibility of mgus). For every \(\rho\) and \(E\) holds \(Mgus(\rho(E)) = \rho(Mgus(E))\). This is due to Theorem 5.25 and Lemma 3.2. Assume \(\sigma \in Mgus(s,t)\), then \(\rho(\sigma)\) is \(\rho(\sigma(s)) = \rho(\sigma(t)) = \rho(\sigma)(\rho(t))\). Further, if \(\theta\) is a unifier of \(\rho(s), \rho(t)\), then \(\theta \cdot \rho\) is a unifier of \(s, t\), hence there is a renaming \(\delta\) with \(\theta \cdot \rho = \delta \cdot \sigma\), giving \(\theta = \delta \cdot \sigma \cdot \rho^{-1} = \delta \cdot \rho^{-1} \cdot \sigma \cdot \rho^{-1} = (\delta \cdot \rho^{-1}) \cdot \rho(\sigma)\), meaning \(\rho(\sigma) \in Mgus(\rho(E))\). For the other direction, observe \(\theta = \rho \cdot \rho^{-1} \cdot \delta \cdot \sigma \cdot \rho^{-1} = \rho(\rho^{-1} \cdot \delta \cdot \sigma)\).

6.1 Handling local variables in HCL\((U)\)

With \(U\) complying to Axiom 6.1 and yielding relevant mgus, that is to say with practically any \(U\) a propagation result for SLD-derivations can be proved, which leads to a constructive and incremental version of the variant lemma.

Regarding SLD-derivations, for the most part we shall assume traditional concepts as given in [11] and [2], but with some changes and additions listed below. An input clause \(\mathcal{K}_i\) obtained from a program clause \(\mathcal{K}\) by replacing the variables in order of appearance with \(t_1, \ldots, t_n\) may be denoted as \(\mathcal{K}_i = \mathcal{K}[t_1, \ldots, t_n]\). Assume now an SLD-derivation \(D\) for \(G\) of the form \(G \leftarrow \mathcal{K}_1 : \sigma_1 \quad G_1 \leftarrow \mathcal{K}_2 : \sigma_2 \quad \ldots \leftarrow \mathcal{K}_n : \sigma_n \quad G_n\).

\(\text{[11]}\) Classical unification algorithms not only satisfy Axiom 6.1 but also yield idempotent mgus. Idempotent mgus are always relevant (2).
\[ \mathcal{K}_i \] is here the \textit{actually used} variant of a program clause (i.e., the current \textit{input clause}) and not the program clause itself.

- The substitution \( \sigma_n \cdots \sigma_1 \) shall be called the \textit{partial answer} for \( G \) at step \( n \) of the derivation. A final partial answer, whenever \( G_n = \Box \), shall be called a \textit{complete answer} for \( G \).

Relation to earlier concepts of answer: In a derivation, the \textit{resultant} of level \( n \) is defined in [12] as \( \sigma_n \cdots \sigma_1 (G) \leftarrow G_n \). So partial answer is at heart of resultant. An \textit{answer substitution} for \( G \) is defined in [3] exactly as complete answer; later ([11], [2]) it is made relevant by restricting to variables of \( G \), hence \( (\sigma_n \cdots \sigma_1) \rvert_G \) whenever \( G_n = \Box \), and called \textit{computed answer substitution} (c.a.s.).

Showing the actually used variants of program clauses (instead of program clauses themselves) enables a simple definition of derivation variables: the annotations \( \mathcal{K}_i ; \sigma_i \) are regarded as part of the derivation, so \( \text{Vars}(D) := (\text{Vars}(G) \cup \ldots \cup \text{Vars}(G_n)) \cup (\text{Vars}(\sigma_1) \cup \ldots \cup \text{Vars}(\sigma_n)) \cup (\text{Vars}(\mathcal{K}_1) \cup \ldots \cup \text{Vars}(\mathcal{K}_n)) \).

Now to the propagation result. Assume the program "son(S) ← male(S), child(S, P).", and let us enquire about \textit{son} in two derivations. If we know that one query, say \textit{son}(X), is a variant of the other, \textit{son}(A), does the same connection hold between the resolvents as well?

As can be seen from Table 1 and Figure 3, in a resolution some new variables may crop up, originating from standardization-apart. Let us call them \textit{local variables}, as opposed to \textit{query variables}. Likely causes are a pattern in a clause head (nat/1 in Table 1) or surplus variables in a clause body (son/1 in Figure 3). But even without those, local variables can appear (p/1 in Table 1), except with a restriction to \textit{normal SLD-derivation} ([4]), preventing "needless renaming of variables". Were it not for local variables, the resolvents in both derivations would clearly be variants, with the same prenaming as for the original queries. Yet, even though the variables new in one derivation do not have to be new in the other, the prenaming can (under reasonable conditions) be extended to accomodate them. The claim is proved in a constructive manner.

| query    | input clause         | resolvent      |
|----------|----------------------|----------------|
| nat(X)   | nat(s(A)) ← nat(A).  | nat(A)         |
| nat(A)   | nat(s(B)) ← nat(B).  | nat(B)         |
| p(X)     | p(A) ← q(A).         | q(X) or q(A)   |
| p(A)     | p(B) ← q(B).         | q(A) or q(B)   |

Table 1: Resolution may produce local variables

\begin{lemma}[propagation of variance]\end{lemma}

Assume a unification algorithm \( \mathcal{U} \) satisfying [Axiom 6.1]. Assume an SLD-derivation \( D \) ending with \( G \) and an SLD-derivation \( D' \) ending with \( G' \) such that \( \alpha(G) = G' \) for some prenaming \( \alpha \) which is complete for \( G \) and relevant for \( D \) to \( D' \).

Further assume that \( G \leftarrow_{\mathcal{K}, \sigma} H \) and \( G' \leftarrow_{\mathcal{K}', \sigma'} H' \) such that in \( G \) and \( G' \) atoms in the same positions were selected and \( \mathcal{K}, \mathcal{K}' \) are variants. Lastly assume that \( \sigma \) is a relevant mgu. Then for \( \lambda := \text{Pren}(\mathcal{K}, \mathcal{K}') \) holds

1. \( \alpha \cup \lambda \) is complete for \( H \) and \( \sigma \)
2. \( \alpha \cup \lambda \) is relevant for \( D \leftarrow_{\mathcal{K}, \sigma} H \) to \( D' \leftarrow_{\mathcal{K}', \sigma'} H' \)
3. \( H' = (\alpha \cup \lambda)(H) \) and \( \sigma' = (\alpha \cup \lambda)(\sigma) \)

The claim can be summarized in Figure 3 together with the rôle of relevance requirement. From Lemma 6.3 and Table 1 follows that resolution is not prenaming-stable (or even renaming-stable); extending the prenaming with local variables may be necessary.
Due to standardization-apart, \( \mathcal{X} \not\equiv D \) and \( \mathcal{X}' \not\equiv D' \), hence

\[
C(\lambda) \not\equiv D \quad \text{and} \quad R(\lambda) \not\equiv D'
\]

Thus \( C(\alpha) \not\subseteq\vars(D) \) and \( R(\alpha) \not\subseteq\vars(D') \), so \( \alpha \not\subseteq \lambda \) is defined. Also, \((3)\) proves that \( \lambda \) is passive on old variables, i.e. \( \lambda(\mathcal{D}) = \mathcal{D} \). Let \( G := (M,A,N) \) and \( H := \sigma(M, B, N) \), where \( M, B, N \) may be conjunctions. Then \( G' = (M', A', N') = (\alpha(M), \alpha(A), \alpha(N)) \). Let \( \mathcal{X} : A_1 \leftarrow B_1 \) and \( \mathcal{X}' : A_2 \leftarrow B_2 \). Then \( \sigma = U(A_1, A_1), B = \sigma(B_1) \) and \( \sigma' = U(A'_1, A_2), B' = \sigma'(B_2) \). Also,

\[
\begin{align*}
\alpha \text{ is complete for } &M, A, N: \vars((M, A, N)) \subseteq C(\alpha), \quad \text{by completeness of } \alpha \text{ for } G \\
\lambda \text{ is complete for } A_1, B_1: \vars(\mathcal{X}) = \vars((A_1, B_1)) = C(\lambda), \quad \text{by definition of } \lambda
\end{align*}
\]

Due to relevance of \( \sigma \), we have \( \vars(\sigma) \subseteq \vars(A) \cup \vars(A_1) \), which together with \((4)\) and \((5)\) gives

\[
\alpha \not\subseteq \lambda \text{ is complete for } \sigma: \vars(\sigma) \subseteq C(\alpha \not\subseteq \lambda)
\]

Having thus fielded all the assumptions, we obtain

\[
\alpha \not\subseteq \lambda \text{ is safe for } M, A, N, A_1, B_1, \sigma: \quad \text{by } (4), (5), (6) \text{ and Theorem 5.18}\]

\[\text{(7)}\]

**Proof of (7):** Completeness of \( \alpha \not\subseteq \lambda \) for \( \sigma \) is proved above. Completeness of \( \alpha \not\subseteq \lambda \) for \( H \) follows from \( \vars(H) \subseteq \vars(G) \cup \vars(\mathcal{X}) \subseteq C(\alpha) \cup C(\lambda) = C(\alpha \not\subseteq \lambda) \), by \((4)\) and \((5)\).

**Proof of (2):** By definition, \( C(\lambda) = \vars(\mathcal{X}) \) and \( R(\lambda) = \vars(\mathcal{X}') \). Hence, and due to relevance of \( \alpha \), \( C(\alpha \not\subseteq \lambda) = C(\alpha \not\subseteq \lambda) \subseteq \vars(D) \cup \vars(\mathcal{X}) \subseteq \vars(D) \searrow_{\mathcal{X}, \sigma} H \). Similarly, \( R(\alpha \not\subseteq \lambda) \subseteq \vars(D') \searrow_{\mathcal{X}', \sigma} H' \), therefore \( \alpha \not\subseteq \lambda \) is relevant for \( D \searrow_{\mathcal{X}, \sigma} H \) to \( D' \searrow_{\mathcal{X}', \sigma} H' \).

**Proof of (3):**

\[
\begin{aligned}
\sigma' &= U(A'_1, A_2) = U(\alpha(A), \lambda(A_1)) = U((\alpha \not\subseteq \lambda)(A), (\alpha \not\subseteq \lambda)(A_1)), \quad \text{by } (4), (5) \text{ and Corollary 4.5} \\
&= U((\alpha \not\subseteq \lambda)(A), (\alpha \not\subseteq \lambda)(A_1)) = (\alpha \not\subseteq \lambda)(\sigma) = (\alpha \not\subseteq \lambda)(\sigma), \quad \text{by } (7), \text{ Lemma 5.14 and Axiom 6.1} \\
B' &= \sigma'(B_2) = (\alpha \not\subseteq \lambda)(\sigma)(B_1) = (\alpha \not\subseteq \lambda)(\sigma)((\alpha \not\subseteq \lambda)(B_1)), \quad \text{by } (5) \text{ and Corollary 4.5} \\
&= (\alpha \not\subseteq \lambda)(\sigma)(B_1) = (\alpha \not\subseteq \lambda)(B), \quad \text{by Theorem 5.25} \\
H' &= \sigma'(\alpha(M), B', \alpha(N)) = (\alpha \not\subseteq \lambda)(\sigma)(\alpha(M), (\alpha \not\subseteq \lambda)(B), \alpha(N)), \quad \text{by preceding line} \\
&= (\alpha \not\subseteq \lambda)(\sigma)((\alpha \not\subseteq \lambda)(M), (\alpha \not\subseteq \lambda)(B), (\alpha \not\subseteq \lambda)(N)), \quad \text{by } (4) \text{ and Corollary 4.5} \\
&= (\alpha \not\subseteq \lambda)(\sigma(M, B, N)) = (\alpha \not\subseteq \lambda)(H), \quad \text{by Theorem 5.25}
\end{aligned}
\]

\(\Diamond\)
6.2 Variant lemma for HCL(U)

**Definition 6.4** (similarity). SLD-derivations of the same length

\[
G \leftrightarrow_{\mathcal{X}_i; \sigma_i} G_1 \leftrightarrow_{\mathcal{X}_i; \sigma_i} \ldots \leftrightarrow_{\mathcal{X}_i; \sigma_i} G_n
\]

\[
G' \leftrightarrow_{\mathcal{X}_i; \sigma'_i} G'_1 \leftrightarrow_{\mathcal{X}_i; \sigma'_i} \ldots \leftrightarrow_{\mathcal{X}_i; \sigma'_i} G'_n
\]

are similar if \( G \) and \( G' \) are variants and additionally at each step \( i \) holds: atoms in the same position are selected, and the input clauses \( \mathcal{X}_i \) and \( \mathcal{X}'_i \) are variants.

That the name "similarity" is justified, follows from the claim known as **variant lemma** ([11], [12], [2]), here in the formulation from [5]: *Finite derivations which are similar and start from variant queries have variant partial answers*. For logic programming systems obeying **Axiom 6.1** and relevance of mgu, a more precise claim can be proved. The added assumptions (the axiom and relevance) are practically void (see footnote on page 496), yet the added conclusion has substance: first, renaming a query now costs a degree of freedom – if we treat the two variants of the program clause at each step as independent, then the two mgu are not independent. Second, the precise variance is now known.

**Theorem 6.5** (variant claim for HCL(U)). Assume a unification algorithm \( U \) satisfying **Axiom 6.1** and yielding relevant mgu. Then:

- finite SLD-derivations which are similar and start from variant queries have variant partial answers
- the variance depends only on the starting queries and input clauses.

In particular, assume our similar derivations to be as in (8). Then for every \( i = 1, \ldots, n \) holds \( G'_i = \beta_i(G_i), \sigma'_i = \beta_i(\sigma_i) \) and \( \sigma'_i \cdot \ldots \cdot \sigma'_1 = \beta_i(\sigma_1 \cdot \ldots \cdot \sigma_1) \), where \( \beta_i := \alpha \cup \lambda_1 \cup \ldots \cup \lambda_i, \alpha := \text{Pren}(G, G') \) and \( \lambda_i := \text{Pren}(\mathcal{X}_i, \mathcal{X}'_i) \).

**Proof.** By assumption, \( G \) and \( G' \) are variants, so

\[
\alpha := \text{Pren}(G, G')
\]

exists. Clearly, \( \alpha \) is complete for \( G \). By construction, \( \alpha \) is also relevant for \( D_0 := G \) to \( D'_0 := G' \). We may iterate Lemma 6.3 obtaining for every \( i = 1, \ldots, n \)

\[
\sigma'_i = (\alpha \cup \lambda_1 \cup \ldots \cup \lambda_i)(\sigma_i)
\]

(10)

\[
G'_i = (\alpha \cup \lambda_1 \cup \ldots \cup \lambda_i)(G_i)
\]

(11)

where \( \lambda_i := \text{Pren}(\mathcal{X}_i, \mathcal{X}'_i) \). Due to completeness of \( \alpha \) for \( G \) and Corollary 4.5

\[
\alpha(G) = (\alpha \cup \lambda_1 \cup \ldots \cup \lambda_i)(G)
\]

(12)

So \( \sigma'_i \cdot \sigma'_{i-1} \cdot \ldots \cdot \sigma'_1 = (\alpha \cup \lambda_1 \cup \ldots \cup \lambda_i)(\sigma_i) \cdot (\alpha \cup \lambda_1 \cup \ldots \cup \lambda_{i-1})(\sigma_{i-1}) \cdot \ldots \cdot (\alpha \cup \lambda_1)(\sigma_1) \). We would like to extract \( \sigma_i \cdot \sigma_{i-1} \cdot \ldots \cdot \sigma_1 \) on the right, to have a connection between partial answers.

Assume \( k < i \). Since \( \text{Vars}(\sigma_i) \subseteq \text{Vars}(G) \cup \text{Vars}(\mathcal{X}_i) \cup \ldots \cup \text{Vars}(\mathcal{X}_k) \subseteq C(\alpha) \cup C(\lambda_1) \cup \ldots \cup C(\lambda_k) = C(\alpha \cup \lambda_1 \cup \ldots \cup \lambda_k) \), by Theorem 5.18(2)

\[
(\alpha \cup \lambda_1 \cup \ldots \cup \lambda_{i-1})(\sigma_{i-1}) = (\alpha \cup \lambda_1 \cup \ldots \cup \lambda_{i-1})(\sigma_{i-1})
\]

\[
(\alpha \cup \lambda_1)(\sigma_1) = (\alpha \cup \lambda_1 \cup \ldots \cup \lambda_i)(\sigma_1)
\]

(13)
Let us abbreviate $\beta_n := \alpha \uplus \lambda_1 \uplus \ldots \uplus \lambda_i$. Then from (10) and (13) by Theorem 5.26

$$\sigma_i' \cdot \sigma_{i-1}' \cdot \ldots \cdot \sigma_1' = \beta_i(\sigma_i) \cdot \beta_i(\sigma_{i-1}) \cdot \ldots \cdot \beta_i(\sigma_1) = \beta_i(\sigma_i \cdot \sigma_{i-1} \cdot \ldots \cdot \sigma_1)$$

which is the promised connection between partial answers.

Clearly, variance of partial answers means variance of complete answers, c.a.s.es and resultants as well. For the cases when $G_n = \emptyset$, we obtain by (14), the expected relationship between the respective complete answers: $\sigma_n' \cdot \ldots \cdot \sigma_1' = \beta_n(\sigma_n \cdot \ldots \cdot \sigma_1)$. C.a.s. differs from complete answer by the added restriction on the query variables. Due to renaming-compatibility of restriction, (14) and $\beta_n(G) = \alpha(G) = G'$, we obtain $\sigma_n' \cdot \ldots \cdot \sigma_1'|G = \beta_n(\sigma_n \cdot \ldots \cdot \sigma_1|G)$, i.e. the same relationship. Finally, knowing that the resultant of step $i$ is $R_i := \sigma_i \cdot \ldots \cdot \sigma_1(G) \leftarrow G$, we obtain

$$R_i' = \sigma_i' \cdot \ldots \cdot \sigma_1'(G') \leftarrow G_i' = \beta_i(\sigma_i \cdot \ldots \cdot \sigma_1)(\alpha(G)) \leftarrow \beta_i(G_i), \text{ by (14), (9) and (11)}$$

$$= \beta_i(\sigma_i \cdot \ldots \cdot \sigma_1)(\beta_i(G_i)) \leftarrow \beta_i(G_i) = \beta_i((\sigma_i \cdot \ldots \cdot \sigma_1)(G)) \leftarrow \beta_i(G_i), \text{ by (12) and Theorem 5.25}$$

$$= \beta_i(R_i)$$

So partial answers of step $i$ (and in consequence c.a.s.es and resultants) are variant via $\beta_i$. \hfill \diamond

**Example 6.6** (similarity). Let the logic program be

$$\text{son}(X) \leftarrow \text{male}(X, A). \quad \text{\% } \mathcal{K}_1$$

$$\text{male}(c), \text{male}(d). \text{child}(d, a). \quad \text{\% } \mathcal{K}_2, \mathcal{K}_3, \mathcal{K}_4$$

An interpreter for LD-resolution may produce the following two derivations:

$$\text{son}(A) \leftarrow_{\mathcal{K}_1} \sigma_1 \text{male}(A), \text{child}(A, C) \leftarrow_{\mathcal{K}_2} \sigma_2 \text{child}(d, C)$$

$$\text{son}(B) \leftarrow_{\mathcal{K}_1'} \sigma_1' \text{male}(B), \text{child}(B, D) \leftarrow_{\mathcal{K}_2'} \sigma_2' \text{child}(d, D)$$

They are obviously similar, with $\mathcal{K}_1 = \mathcal{K}_1'[X, C], \mathcal{K}_1' = \mathcal{K}_1'[Y, D], \mathcal{K}_2 = \mathcal{K}_2' = \mathcal{K}_3$. The variables $X, Y$ stand for actually used variables, which are not discernible from the form of derivations. From the queries, input clauses and resolvents we can further deduce which relevant mgus were used: $\sigma_1 = \left( \begin{array}{l} X \\ A \end{array} \right)$, $\sigma_1' = \left( \begin{array}{l} Y \\ B \end{array} \right)$, $\sigma_2 = \left( \begin{array}{l} A \\ B \end{array} \right)$ and $\sigma_2' = \left( \begin{array}{l} B \\ C \end{array} \right)$. The mappings are $\alpha = \left( \begin{array}{l} A \\ B \end{array} \right)$, $\lambda_1 = \left( \begin{array}{l} X \\ C \end{array} \right)$ and $\lambda_2 = \varepsilon$.

Clearly, they fulfill $(\alpha \uplus \lambda_1)(\text{male}(A), \text{child}(A, C)) = \text{male}(B), \text{child}(B, D)$ and $(\alpha \uplus \lambda_1)(\left( \begin{array}{l} B \\ A \end{array} \right)) = \left( \begin{array}{l} C \\ B \end{array} \right)$, as well as $(\alpha \uplus \lambda_1 \uplus \lambda_2)(\text{child}(d, C)) = \text{child}(d, D)$, and so forth.

### 7 Outlook

Concepts relating to variable renaming have been reviewed and built upon, with the aim of providing for practical needs of program analysis and formal semantics in logic programming. By relaxing the core representation and forgoing permutation requirement for renaming, the concept of prenaming is obtained. It is a mathematical underpinning of the intuitive practice of renaming terms by just considering the necessary bindings, where now $x/x$ may be necessary. In other words, a prenaming is a variable-pure substitution with mutually distinct variables in range, possibly including some passive bindings.

Prenamings enable incremental claims like variance propagation in implemented Horn clause logic [Lemma 6.3][Theorem 6.5]. There, prenamings made it possible to keep track of new (local) variables.
References

[1] G. Amato & F. Scozzari (2009): Optimality in goal-dependent analysis of sharing. Theory and Practice of Logic Programming 9(5), pp. 617–689, doi:10.1017/S1471068409990111

[2] K. R. Apt (1997): From logic programming to Prolog. Prentice Hall.

[3] K. R. Apt & M. H. van Emden (1982): Contributions to the theory of logic programming. J. of ACM 29(3), pp. 841–862, doi:10.1145/322326.322339

[4] R. N. Bol (1992): Generalizing completeness results for loop checks in logic programming. Theor. Comp. Sci. 104(1), pp. 3–28, doi:10.1016/0304-3975(92)90164-B

[5] K. Doets (1993): Leviationis laus. J. Logic and Computation 3(5), pp. 487–516, doi:10.1093/logcom/3.5.487

[6] E. Eder (1985): Properties of substitutions and unifications. J. Symbolic Computation 1(1), pp. 31–46, doi:10.1016/S0747-7171(85)80027-4

[7] J. Gallier (2015): Logic for computer science: Foundations of automatic theorem proving, 2. edition. Dover.

[8] D. Jacobs & A. Langen (1992): Static analysis of logic programs for independent AND parallelism. J. of Logic Programming 13(2-3), pp. 291 – 314, doi:10.1016/0743-1066(92)90034-Z

[9] M. Kulaš (2005): Toward the concept of backtracking computation. In L. Aceto, W. J. Fokkink & I. Ulidowski, editors: Proc. of the Workshop on Structural Operational Semantics (SOS’04), London, ENTCS 128, Elsevier, pp. 39–59, doi:10.1016/j.entcs.2004.10.026

[10] J. Lassez, M. J. Maher & K. Marriott (1988): Unification revisited. In M. Boscarol, L. Carlucci Aiello & G. Levi, editors: Foundations of Logic and Functional Programming, LNCS 306, Springer Berlin Heidelberg, pp. 67–113, doi:10.1007/3-540-19129-1_4

[11] J. W. Lloyd (1987): Foundations of logic programming, 2. edition. Springer-Verlag. doi:10.1007/978-3-642-83189-8

[12] J. W. Lloyd & J. C. Shepherdson (1991): Partial evaluation in logic programming. J. of Logic Programming 11(3-4), pp. 217–242, doi:10.1016/0743-1066(91)90027-M

[13] C. Palamidessi (1990): Algebraic properties of idempotent substitutions. In: Proc. 17th ICALP, LNCS 443, Springer-Verlag, pp. 386–399, doi:10.1007/BFb0032046

[14] G. D. Plotkin (1971): Automatic methods of inductive inference. Ph.D. thesis, U. of Edinburgh. Available at http://homepages.inf.ed.ac.uk/gdp

[15] C. Pusch (1996): Verification of compiler correctness for the WAM. In: Proc. TPHOLs, LNCS 1125, Springer Berlin Heidelberg, pp. 347–361, doi:10.1007/BFb0105415

[16] J. C. Shepherdson (1994): The role of standardising apart in logic programming. Theor. Comp. Sci. 129(1), pp. 143–166, doi:10.1016/0304-3975(94)90084-1

[17] C. Urban, A. Pitts & M. Gabbay (2004): Nominal unification. Theoretical Computer Science 323(1-3), pp. 473–497, doi:10.1016/j.tcs.2004.06.016 Proof on http://www.inf.kcl.ac.uk/staff/urbanc/Unification/