Marija Kulaš

A practical view on substitutions

Mathematik und Informatik

Informatik-Berichte
372 – 07/2016
A practical view on substitutions

Marija Kulaš
A practical view on substitutions

Marija Kulaš
FernUniversität in Hagen, Wissensbasierte Systeme, 58084 Hagen, Germany
marija.kulas@fernuni-hagen.de

Abstract
For logic program analysis or formal semantics, the issue of renaming terms and generally handling substitutions is inevitable. We revisit substitutions 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 and allows for extensibility. A novel algorithm for term matching is proposed, which also solves the problem of substitution generality (and thus equivalence), using witness term technique. The technique alleviates the problem of ad-hoc proofs involving generality.

Categories and Subject Descriptors F.4.1 [Logic and constraint programming]

Keywords substitution, renaming, term matching, generality

1. Introduction
The image of substitutions in logic programming research is a somewhat tainted one. First, it has been pointed out by H.-P. Ko (Shepherdson 1994, p. 148) that the original claim of strong completeness of SLD-resolution needs to be rectified, because of a counter-example using the fact that \( f(x, a) \), which also solves the problem of substitution generality (and thus equivalence), using witness term technique. The technique alleviates the problem of ad-hoc proofs involving generality.

In Section 6, we discuss some other notions about substitutions that were needed in the course of work on operational semantics. A novel algorithm for term matching, as well as rooted in the relaxed core idea, is proposed (Algorithm 6.1). It solves the problem of substitution generality (and thus equivalence) as well, using witness term (Theorem 6.8). Witness term technique was also used for a direct proof of Legacy 6.27.

[Copyright notice will appear here once ‘preprint’ option is removed.]
2. Substitution

First we need a bit of notation. Assume two disjoint sets: the set of functors, Fun, and the countably infinite set of variables, V. If \( W \subseteq V \), any mapping \( F : W \rightarrow V \) shall be called variable-pure on \( W \). A mapping variable-pure on the whole set of variables \( V \) shall be called all-vars mapping. If \( W \setminus W \) is finite, \( W \) is said to be co-finite. A mapping \( F \) is injective on \( W \), if whenever \( F(x) = F(y) \) for \( x, y \in W \) we have \( x = y \).

Associated with every functor \( f \) shall be a natural number \( n \) denoting its number of arguments, arity. To emphasize this, the notation \( f/n \) will be used. Functions of arity 0 are called constants. Starting from Fun and Fun we build data objects, terms. In Prolog, everything is a term, and so shall term be here the topmost syntactic concept. 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 \( s \in t \).

A special kind of term is dotted pair, introduced under the name S-expression in [McCarthy 1960] and written [h,t] as h-t, where h is called the head and t is called the tail of the pair. A special dotted pair is non-empty list, distinguished by its tail being a special term nil called empty list, or a non-empty list itself. In Edinburgh Prolog notation, dotted pair would be written [h|t] and empty list as [].

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 \( t \). A term without variables is called a ground term. If the terms \( s \) and \( t \) share a variable, that shall be written \( s \parallel t \). Otherwise, we say \( s, t \) are variable-disjoint, written as \( s \not\parallel t \). The list of all variables of \( t \), in order of appearance, shall be denoted as \( \text{VarList}(t) \).

A recurrent theme in this paper shall be relevance, meaning "no extraneous variables" (relative to some term or terms). The name appears in [ Apt 1997] p.38, with the unary meaning, i.e. no extraneous variables relative to (one) term. This usage shall be reflected in the text as follows.

- A unifier \( \sigma \) of a set of equations \( E \) is a relevant unifier, if \( \text{Vars}(\sigma) \subseteq \text{Vars}(E) \). A renaming \( \rho \) embedding a preening \( \alpha \) is a relevant embedding, if \( \text{Vars}(\rho) \subseteq \text{Vars}(\alpha) \).

Additionally, a binary version of relevance, handling two terms, shall also be needed [Algorithm 5.1][Lemma 7.1]:

- A mapping \( F \) is relevant for \( t_1 \) to \( t_2 \), if \( \text{Dom}(F) \subseteq \text{Vars}(t_1) \) and \( \text{Range}(F) \subseteq \text{Vars}(t_2) \).

Definition 2.1 (substitution). A substitution is a function mapping variables to terms, which is identity almost everywhere. In other words, a function \( \theta \) with domain \( \text{Dom}(\theta) = V \) such that the following requirement holds:

**finite action**\(^2\) The set \( \{x \in V \mid \theta(x) \neq x \} \) is finite.

The set \( \text{Core}(\theta) := \{x \in V \mid \theta(x) \neq x \} \) shall be called the active domain, if \( \text{Core}(\theta) \subseteq \text{Core}(\theta) \) for all \( \theta \), and its elements active variables of \( \theta \). The set \( \text{Ran}(\theta) := \{(x, \theta(x)) \mid x \in \text{Core}(\theta) \} \) is called the active range of \( \theta \). The set \( \text{Vars}(\theta) := \{x \in V \mid \theta(x) \neq x \} \) is called the active variable range of \( \theta \). For completeness, a variable \( x \) such that \( \theta(x) = x \) shall be called a passive variable, or a 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 \( \text{Core}(\theta) = \{x_1, \ldots, x_n\} \), where \( x_1, \ldots, x_n \) are pairwise distinct variables, and \( \theta \) maps each \( x_i \) to \( t_i \), then \( \theta \) shall have the core representation \( \{x_1/t_1, \ldots, x_n/t_n\} \), or the perhaps more visual \( \{x_1 \cdots x_n/t_1 \cdots t_n\} \). Hence, the above requirement shall also be called finite core. Each pair \( x_i/t_i \) is called the binding for \( x_i \) via \( \theta \), denoted by \( x_i/t \in \theta \).

Often we identify a substitution with its core representation, and thus regard it as a syntactical object, a term. 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 restriction of \( \sigma \), and \( \sigma \) is an 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 a substitution \( \theta \) upon the variables of the term \( t \) shall be abbreviated as \( \theta|t := \theta|\text{Vars}(t) \). We also write \( \theta|\{t\} \) to denote the restriction of \( \theta \) to variables outside of \( t \), like \( \theta|\{t\} := \theta|\text{Core}(\theta) \cup \text{Vars}(t) \).

Definition of substitution is extended 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 \( s \) is a term, \( \theta(s) \) is an instance of \( s \) via \( \theta \).

The composition \( \theta \circ \sigma \) of substitutions \( \theta \) and \( \sigma \) is defined by \( \theta \circ \sigma(t) := \theta(\sigma(t)) \). Composition may be iterated, written as \( \theta_1 \circ \cdots \circ \theta_n \) for \( n \geq 1 \), and \( \theta^0 \) := \( () \). Here \( \theta := () \) is the identity function on \( V \). In case an all-vars substitution \( \rho \) is bijective, its inverse shall be denoted as \( \rho^{-1} \). A substitution \( \theta \) satisfying the equality \( \theta \circ \theta = \theta \) is called idempotent.

Example 2.2. \( (x w w u v) \cdot (u w w u v z w z y z w) \cdot (x y w w z z) = (\# x y x w z w # w z y z w \# x y w w \# w) \).

3. Renaming

Definition 3.1 (renaming). A renaming of variables is a bijective all-vars substitution.

In [Eder 1985], it is synonymously called "permutation". We shall reserve the word for the general case where infinite movements like translation are possible. Here we shall synonymously speak of finite permutation due to the fact that, being a substitution, any renaming has a finite core, and [Lemma 3.4] holds.

From the definition of substitution, we know: If \( s \in t \), then \( \sigma(s) \in \sigma(t) \). For bijective substitutions (i.e. renamings), a complementary property holds as well:

Lemma 3.2 (renaming stability of not-in). Let \( \rho \) be a renaming and \( s, t \) be terms. If \( s \not\in t \), then \( \rho(s) \not\in \rho(t) \).

Proof. Assume \( \rho(s) \in \rho(t) \). Then \( \rho^{-1}(\rho(s)) \in \rho^{-1}(\rho(t)) \). ◯

Corollary 3.3 (renaming stability of ",", "\#", "\##"). Let \( \rho \) be a renaming and \( s, t \) be terms. Then \( s = t \iff \rho(s) = \rho(t) \), and also \( s \in \rho(t) \iff \rho(s) \not\in \rho(t) \). As a consequence, \( s \not\in \rho(t) \iff \rho(s) \not\in \rho(t) \).

Legacy 3.4 ([Lassez et al. 1988]). A substitution \( \rho \) is a renaming iff \( \rho(\text{Core}(\rho)) = \text{Core}(\rho) \).

Legacy 3.5 ([Eder 1985]). Every injective all-vars substitution is a renaming.

So composition of renamings is a renaming. The next property is about cycle decompositions of a finite permutation.
Lemma 3.6 (cycles). Let \( \sigma \) be an all-vars substitution. It is injective iff for every \( x \in V \), there is a \( n \in N \) 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, \( \sigma^i(x_0) \neq \sigma^i(x_0) \) for all \( i \geq 1 \), due to injectivity. Hence, \( \sigma^{m-1}(x_0) \in \text{Core}(\sigma) \) 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, \( \sigma^{m-n}(x_0) = \sigma^{k-n}(x_0) \).

By iteration we get \( n := m - k \).

For the other direction, assume \( \sigma(x) = \sigma(y) \), and minimal \( m, n \) such that \( \sigma^m(x) = x, \sigma^n(y) = y \). Consider the case \( m \neq n \), say \( m > n \). Then \( \sigma^{m-n}(y) = \sigma^{m-n}(x) = \sigma^{m-n}(\sigma^n(x)) = \sigma^{m-n}(\sigma^n(y)) = \sigma^m(y) = y \), contradicting minimality of \( m \).

Hence \( m = n \), and so \( x = \sigma^n(x) = \sigma^n(y) = y \).

4. Relaxed core representation

In [Lemma 7.1] we shall have to deal with mappings between two terms. There, it is possible that a variable stays the same, so \((x, x)\) would have to be tolerated as a “binding”, since we need our mapping to cover all variables in the two terms. Therefore, we allow the set \( C \) to contain some passive variables, raising those above the rest, as it were.

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 \((x_1, \ldots, x_n, \sigma(x_1), \ldots, \sigma(x_n))\) shall be called a relaxed core representation for \( \sigma \).

If \( \sigma \) is a relaxed core for \( \sigma \), shall be denoted \( \text{C}(\sigma) \) := \((x_1, \ldots, x_n)\). The associated range \( \text{C}(\sigma) \) we denote as \( \text{R}(\sigma) \).

The set of variables of \( \sigma \) being is as expected, \( \text{V}(\sigma) := \text{Vars}(\text{C}(\sigma)) \cup \text{Vars}(\text{R}(\sigma)) \). To get back to the traditional core representation, we denote by \( \sigma \) the core representation of \( \sigma \).

For extending substitution, we shall employ disjoint union.

Definition 4.2 (sum of substitutions). If \( \sigma = (x_1 \ldots x_m, \theta) \) and \( \psi = (y_1 \ldots y_n, \theta) \) are substitutions in relaxed representation such that \( \{y_1, \ldots, y_n\} \not\supset \{x_1, \ldots, x_m\} \), then \( \sigma \oplus \psi := (x_1 \ldots x_m, y_1 \ldots y_n, \sigma(x_1), \ldots, \sigma(x_m), \theta) \) is the sum of \( \sigma \) and \( \psi \).

In case \( \{y_1, \ldots, y_n\} \supset \{x_1, \ldots, x_m\} \) but with \( \sigma(x_i) = \theta(y_j) \) on any common variables \( x_i = y_j \), we shall simply write \( \sigma \cup \theta \). Also, we shall not be introducing special symbols to denote that \( \sigma \) is an extension of \( \theta \), but simply write \( \sigma \geq \theta \).

In subsection 7.2 we shall need backward compatibility of an extension. A first stab might be:

Lemma 4.3. If \( \beta(t) = t \), then \( (\alpha \oplus \beta)(t) = \alpha(t) \).

Proof. For any \( x \in \text{Vars}(t) \cap \text{C}(\alpha) \) by definition \( (\alpha \oplus \beta)(x) = \alpha(x) \). Assume now \( x \in \text{Vars}(t) \cap \text{C}(\beta) \). From the condition, \( \beta(x) = x \), and by definition of extension, \( x \not\in \text{C}(\alpha) \), hence \( (\alpha \oplus \beta)(x) = \beta(x) = x = \alpha(x) \). Clearly, if \( x \in \text{Vars}(t) \backslash \text{C}(\alpha \oplus \beta) \) the claim also holds.

As an immediate consequence, if a substitution \( \sigma \) is complete for a term \( t \), there is no danger that an extension of \( \sigma \) might map \( t \) differently from \( \sigma \).

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

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

5. Prenaming

In practice, one would like to change the variables in a term, without bothering to check whether this change is a permutation or not. For example, the term \( p(z, u, x) \) can be mapped on \( p(y, z, x) \) via \( z \mapsto y, u \mapsto z, x \mapsto x \).

Let us call such a mapping prenaming 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 \( \text{C}(\alpha) \) is fixed by the terms to map. Obviously, injectivity is important for such a mapping, since \( p(z, u, x) \) cannot be mapped on \( p(y, y, y) \) without losing a variable. Hence:

Definition 5.1 (prenaming). A prenaming \( \alpha \) is an all-vars substitution injective on a finite set of variables \( \text{C}(\alpha) \supseteq \text{Core}(\alpha) \).

Obviously, any renaming is a prenaming. For [Theorem 7.4] we need a possibility to extend a given prenaming by new bindings.

Lemma 5.2 (extension of prenaming). Let \( \alpha = (x_1 \ldots x_n, \theta) \) and \( \beta = (u_1 \ldots u_k, \theta) \) be prenaminings such that \( \{u_1, \ldots, u_k\} \not\supset \{x_1, \ldots, x_n\} \) and \( \{x_1, \ldots, x_n\} \not\subseteq \{v_1, \ldots, v_k\} \). Then \( \alpha \oplus \beta = (x_1 \ldots x_n, u_1 \ldots u_k, \theta) \) is also a prenaming.

Clearly, \( \text{C}(\alpha \oplus \beta) = \text{C}(\alpha) \cup \text{C}(\beta) \) and \( \text{R}(\alpha \oplus \beta) = \text{R}(\alpha) \cup \text{R}(\beta) \).

5.1 The question of inverse

In practice, a renaming is more natural, but a “full” renaming is better mathematically tractable (inverse exists). Hence we want to know whether each prenaming can be embedded in a renaming.

The next property shows how to extend a prenaming \( \alpha \) to obtain a renaming, and a relevant one at that, i.e. acting only on the variables from \( V(\alpha) \). The claim is essentially given in [Lloyd and Shepherdson 1991], [Apt 1997] and [Amato and Scrocci 2009] with the emphasis on the existence of such an extension. In [Eder 1985], the emphasis is on the actual reach of the extension. The latter is our concern as well. We formulate the claim around the notion of prenaming, and provide a constructive proof based on [Lemma 3.6].

Theorem 5.3 (embedding). Let \( \alpha \) be a prenaming. Then there is a renaming \( \rho \) which coincides with \( \alpha \) on \( V \setminus (R(\alpha) \setminus C(\alpha)) \) such that \( \text{Vars}(\rho) \subseteq V(\alpha) \).

Additionally, if \( \alpha(x) \neq x \) on \( C(\alpha) \), then \( \rho(x) \neq x \) on \( V(\alpha) \).

Proof. If \( \alpha \) is a prenaming, then \( C(\alpha) = C \) and \( R(\alpha) = R \) sets of \( n \) distinct variables each. We shall construct the wanted renaming in [Algorithm 5.1] where it is named \( \tau \). The idea is to close any open chains \( \alpha(x), \alpha^2(x), \ldots \).

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

Algorithm 5.1: Closure, the natural relevant embedding

\[ 5 \] Finding an appropriate name can be a struggle. Shortlisted were pre-renaming and proto-renaming.

[Apr 1997] 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.”

[Eder 1985] p. 35): “Let \( W \) be a co-finite set of variables (...) and let \( \alpha \) be a W-renaming. Then there is a permutation \( \tau \) which coincides with \( \sigma \) on the set \( W \)”
Let us see if for every $x$ there is a $j$ such that $\overline{\alpha}^j(x) = x$. If $x \in C$, we start as in the proof of Lemma 3.6 and consider the sequence $\alpha(x)$, $\alpha^2(x)$, ..., Since $C$ is finite, either we get two equals (and proceed as there), or we get $\alpha(x) \notin C$ and are stuck. For $y := \alpha^m(x)$ 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 $\alpha^{m+1}(x) = x$.

The cases $x \in R \cup C$ or $x \notin C \cup R$ are easy. By Lemma 3.6 $\overline{\alpha}$ is injective. By Lemma 3.5 $\overline{\alpha}$ is a renaming. The discussion of the case $\alpha(x) \neq x$ on $C(\alpha)$ is straightforward. \H

Definition 5.4 (closure of a renaming). The renaming $\overline{\alpha}$ constructed in Algorithm 5.1 shall be called the closure of $\alpha$.

Remark 5.5 (relevant embedding is not unique). Let $\alpha = \{ (x, z), (y, w) \}$, and let us embed it in a relevant renaming. The Algorithm 5.1 gives $\overline{\alpha} = \{ (x, y), (z, w) \}$. But $\rho = \{ (x, z), (y, w) \}$ is also a relevant renaming which is embedding $\alpha$. In the usual notation for cycle decomposition, $\rho = \{ (x, w), (z, y), (u, z), (w, y) \}$ and $\overline{\alpha} = \{ (x, u, z, y), (w, y, z) \}$.

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

Lemma 5.6 (reverse renaming). Let $\alpha := \{ (x_1, z_1), \ldots, (x_n, z_n) \}$ and $\beta := \{ (y_1, z_1), \ldots, (y_n, z_n) \}$. Then $\beta = \overline{\alpha}^{-1}$.

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

Remark 5.8 (closure is not monotone). If $\alpha \supseteq \alpha'$, then not always $\overline{\alpha} \supseteq \overline{\alpha}'$. To see this, let $\alpha = \{ (x, y) \}$ and $\alpha' = \{ (x, y) \}$. Then $\overline{\alpha}' = \{ (x, y) \}$ and $\overline{\alpha} = \{ (x, y) \}$.

5.2 Staying safe

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

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

Remark 5.8 means: if we want to use $\alpha$ on a larger set than $C(\alpha)$, then the set $\Pi(\alpha) := R(\alpha) \setminus C(\alpha)$ is dangerous to touch. But, luckily, its complement is not:

Lemma 5.10 (larger set). A renaming $\alpha$ is injective on the finite set $V \setminus \Pi(\alpha)$. The set is maximal containing $C(\alpha)$.

Proof. Let $x, y \in V \setminus \Pi(\alpha)$. Is it possible that $\alpha(x) = \alpha(y)$? Possible cases: If $y, x \in C(\alpha)$, then by definition of renaming $\alpha(x) \neq \alpha(y)$. If $x, y \in C(\alpha)$, then $\alpha(x) = \alpha(y)$. It remains to consider the mixed case $x \in C(\alpha)$, $y \notin C(\alpha)$. 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 $y \notin C(\alpha)$, that would mean $y \in \Pi(\alpha)$. Contradiction.

The set cannot be made larger: if $y \in \Pi(\alpha)$, then there is $x \in C(\alpha)$ with $x \neq y$ and $\alpha(x) = y = \alpha(y)$, so injectivity is compromised. \H

Definition 5.11 (injectivity domain). For a renaming $\alpha$, let $\text{InDom}(\alpha) := V \setminus \Pi(\alpha)$. Since $\text{InDom}(\alpha)$ is the largest cofinite set containing $C(\alpha)$ on which $\alpha$ is injective, it shall be called the injectivity domain of $\alpha$.

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

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

Clearly, $\text{InDom}(\alpha) = C(\alpha) \cup (V \setminus R(\alpha))$, so $\alpha$ is safe for its relaxed core. Hence, if $\alpha$ is complete for a term, it is safe for that term. For a renaming $\alpha$ 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 renaming behaves like a renaming on its injectivity domain, since it coincides with its closure there. This follows immediately from Theorem 3.3.

Corollary 5.13 (injectivity domain). Let $x \in \text{InDom}(\alpha)$. Then $\alpha(x) = \overline{\alpha}(x)$.

Corollary 5.14 (premaining stability). A generalization of Corollary 3.4 holds: Let $s, t$ be terms and $\alpha$ be a renaming safe for $s, t$. Then $s = t \iff \alpha(s) = \alpha(t)$ and also $s \neq t \iff \alpha(s) \neq \alpha(t)$.

As a consequence, $s \neq t \iff \alpha(s) \neq \alpha(t)$.

Our definition of renaming was inspired by the following more general notion from [Eder 1985].

Definition 5.15 (W-renaming). 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 $t$, $\alpha \cup \beta$ does not have to be, even if $\beta(t) = t$, as the following example shows: $\alpha := \{ (x) \}$, $\beta := \{ (y) \}$, $t := p(x)$. The following two claims try to redress that issue:

Lemma 5.16 (monotonicity). Assume $\alpha \cup \beta$ is defined. Then

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

Proof. Since $(V \setminus A) \cup (V \setminus B) = V \setminus (A \cap B)$, and $\Pi(\alpha) \notin \Pi(\beta)$, we obtain $\text{InDom}(\alpha) \cup \text{InDom}(\beta) = V$.

Further, $(V \setminus A) \cap (V \setminus B) = V \setminus (A \cup B)$ and so $\Pi(\alpha \cup \beta) = (R(\alpha) \cup R(\beta)) \setminus (C(\alpha) \cup C(\beta)) \subseteq (R(\alpha) \setminus C(\alpha)) \cup (R(\beta) \setminus C(\beta)) = \text{InDom}(\alpha) \cup \text{InDom}(\beta)$.

In Remark 5.8 $\Pi(\alpha') = \{ y \}$, $\Pi(\beta') = \{ x \}$, and $\Pi(\alpha) = \{ x \}$; 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 $\text{InDom}(\alpha \cup \beta)$. 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 $\alpha \cup \beta$. This shall be used in Section 7.

Theorem 5.17 (extensibility). Assume $\alpha \cup \beta$ is defined. Then

1. If $\alpha$ is safe for $t$ and $\beta$ is safe for $t$, then $\alpha \cup \beta$ is safe for $t$.\footnote{Our definition of safe renaming is more general than the definition of renaming for a term in [Lloyd 1987] 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 [Amato and Scozzari 2009]. For this, substitution shall 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, as in subsection 7.2.

5.3.1 Term variant

Definition 5.18 (term variant). If α is a prenaming safe for t, then we also call α(t) a variant of t, and write α(t) ≃ t. The particular variance and the direction of its application may be explicited: α ≃ t iff s = α(t).

If α ≃ t, then there is a unique α mapping s to t in a complete relevant manner, i.e. mapping each variable pair and nothing else, as computed by Algorithm 5.2. The algorithm makes do with only one set for equations and bindings, thanks to different types. Termination can be seen from the tuple (fun(E), card(E)) decreasing in lexicographic order with each rule application, where fun(E) is the number of function symbols in equations in E, and card(E) is the number of equations in E.

Algorithm 5.2: Computing the prenaming of s to t

Notation 5.19 (episoid). The prenaming constructed in Algorithm 5.2 shall be simply called the prenaming of s to t, and denoted Pren(s, t). It is complete for s and relevant for s to t.

In case s = t, we obtain for Pren(s, t) essentially the identity substitution. However, regarded as prenaminings, Pren(t, t) and ε are not the same. A prenaming α with relaxed core W mapping each variable on itself (in other words, C(α) = W and ε = ε) shall be called the W-episoid and denoted εW. For a term t, we abbreviate εt := εVars(t).

Regarding composition, an episoid behaves just like ε. Its use is for providing completeness, and hence extensibility, by means of placeholdering pairs x/x.

2. If α is complete for t, then α ⊔ β is safe for t and (α ⊔ β)(t) = α(t).

The first part follows from Lemma 5.16 and the second from Corollary 4.2. Observe the importance of relaxed core for this to work; otherwise, passive bindings x/x would not be accounted for.

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 ρ is a renaming and σ is a substitution, [Amato and Scozzari 2009] define substitution renaming by ρ(σ) := {ρ(x)/ρ(σ(x)) | x ∈ Core(σ)}. It is easy to see that ρ(σ) is a substitution in core representation. For this we only need two properties of ρ: variable-pure on Vars(σ) and injective on Vars(σ). These requirements are clearly fulfilled by prenaminings safe on σ as well. Hence,

Definition 5.20 (substitution variant). Let σ be a substitution and let α be a prenaming safe for σ, i.e. Vars(σ) ⊆ InDom(α). Then a variant of σ by α is

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

(1)

We may write ρ = α σ if ρ = α(σ), 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 with α = (x \ y), σ = (x \ y).

Lemma 5.21 (well-defined). Substitution variant is well-defined, i.e. (1) is a core representation of a substitution, and α does not introduce aliasing.

Proof. Let Core(σ) = {x1, ..., xn}. Due to injectivity of α on Vars(σ), if σ(xi) = σ(xj), then xi = xj, so i = j. To finish the proof that (1) is a core representation, observe x ∈ Core(σ) if x = σ(x) (i.e. α(x) = σ(x)), due to injectivity again.

Next, by Corollary 5.14 if α(σ(xj)) ∼ α(σ(xj)), then α(σ(xj)) ∼ α(σ(xj)), meaning that α does not introduce aliasing. ♦

From Definition 5.20 and Corollary 5.13 follows

Lemma 5.22. Let σ be a substitution and α, β be prenaminings such that α(σ) and (α · β)(σ) are defined. Then
1. (α · β)(σ) = α(β(σ))
2. α(σ) = π(α)

For the case of "full" renaming, there is a way to dissolve the new expression[9].

Legacy 5.23 ([Amato and Scozzari 2009]). For any renaming ρ and substitution σ

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

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

Theorem 5.24 (substitution variant). Let σ be a substitution and α be a prenaming safe for σ. Then
1. $$\alpha(\sigma) \cdot \alpha = \alpha \cdot \alpha$$
2. $$\alpha(\sigma) = \pi \cdot \sigma \cdot \pi^{-1}$$

Proof. First part: According to Definition 5.20 for every x ∈ 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 π(α(σ)) = π(σ), hence π(σ) = π(σ) · π^{-1}. By Corollary 5.13 holds α(σ) = π(σ), which completes the proof. ♦

It is known that idempotence and equivalence of substitutions are not compatible with composition [Edler 1985]. Luckily, the concept of variance, with constant prenaming, does not share this handicap:

[9] An immediate consequence of which is ρ(σ) = ρ · σ.
Theorem 5.25 (compositionality). Let \( \sigma, \theta \) be substitutions and \( \alpha \) be their safe prenaming. 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 5.24 \( \alpha(\sigma) \cdot \alpha(\theta) = \alpha(\sigma) \cdot \alpha(\theta) = \alpha(\sigma \cdot \theta) \cdot \alpha(\theta) = \alpha(\sigma \cdot \theta) \cdot \alpha(\theta) \). \( \Box \)

6. Further topics
Here is a brief overview of other substitution properties that we found useful for analysing the operational semantics S1-PP.

For some properties like [Lemma 7.1 and Theorem 7.4] we need the concept of SLD-derivations. Regarding SLD-derivations, we shall for the most part assume traditional concepts as given in [Apt 1997], but with some changes and additions outlined below. The variable names in actual logic programs shall be capitalized, as in Prolog.

**Notation 6.1** (adapting SLD-derivation). Assume an SLD-derivation for \( G \) like \( G \leftarrow \varphi_{K_1} : \sigma_1, G \leftarrow \varphi_{K_2} : \sigma_2, \ldots, \varphi_{K_n} : \sigma_n, G. \)

- \( K_i \) is the **currently used** variant of a program clause (i.e., the current input clause) and not the program clause itself.
- The substitution \( \sigma_1, \ldots, \sigma_n \) shall be called the **partial answer at step** \( n \) of the derivation.
- Recall that a **computed answer substitution** (c.a.s.) for \( G \) is defined as \( (\sigma_1, \ldots, \sigma_n) \subseteq \text{vars} \), whenever \( G_n = \varnothing \). For our purposes, the restriction on the variables of \( G \) is not urgent. As an interim step, we define a complete answer for \( G \) to be a final partial answer, \( \sigma_1, \ldots, \sigma_n \). A c.a.s. is then a complete answer made relevant by restricting it to query variables.

An input clause \( K \) obtained from a program clause \( \tilde{K} \) by replacing the variables in order of appearance with \( A_1, \ldots, A_n \), may be denoted as \( K_i = \tilde{K}[A_1, \ldots, A_n] \). We also say that \( \tilde{K} \) is **applicable** on \( G_i \) with **effector clause** \( K_i \), and that \( K_i \) is **effective** on \( G_i \) with \( \sigma_i \).

Showing the actually used variants of program clauses (instead of program clauses themselves) enables a simple definition of derivation variables.

**Definition 6.2** (variables of a derivation). Assume \( D \) to be an SLD-derivation of \( G \leftarrow \varphi_{K_1} : \sigma_1, \varphi_{K_2} : \sigma_2, \ldots, \varphi_{K_n} : \sigma_n, G. \) We shall define the set of variables of \( D \) as would be natural for a term, i.e., we regard the annotations \( K_i : \sigma_i \) as part of the derivation. Hence, \( \text{Vars}(D) := (\text{Vars}(G) \cup \ldots \cup \text{Vars}(G_n)) \cup (\text{Vars}(\sigma_1) \cup \ldots \cup \text{Vars}(\sigma_n)) \cup (\text{Vars}(K_1) \cup \ldots \cup \text{Vars}(K_n)). \)

One last piece of introductory notation: \( \text{Head}(H \leftarrow \text{B}) := H \).

6.1 Term matching and subsumption
Consider \( f(x, y) \) and \( f(z, x) \). Intuitively, they "match" each other, while \( f(x) \) and \( g(x) \) do not. If asked about \( f(x, x) \) and \( f(x, y) \), we may consent that they "match" only in one direction.

**Definition 6.3** (term matching). Let \( g \) and \( s \) be two terms. If there is a substitution \( \sigma \) such that \( g \sigma = s \), then we say \( g matches s \), and also that \( s \) is an instance of \( g \) (as already defined in [Definition 2.1]).

The substitution \( \sigma \) is then a **matcher** of \( g \) on \( s \).

Moreover, if \( g \sigma = s \), then we say \( g subsumes s \). The substitution \( \sigma \) is then a **subsumer** of \( g \) by \( s \).

Example: \( f(x) \) matches \( f(g(z)) \), but does not subsume it, while \( f(x, y) \) subsumes \( f(x, x) \). For relation to Prolog see [Neunheisel 2010].

Term matching can be seen as a special case of unification, where any variables on the right-hand side are inactivated by replacing them with new constants (hence the synonym "one-sided unification"). For parallel approach, see e.g. [Dwork et al. 1984]. We propose a one-pass algorithm with a stress on simplicity. [Algorithm 6.1] It decides generality and equivalence of substitutions as well.

6.1.1 Subterm
**Definition 6.4** (subterm, occurrence). A character subsequence of the term \( t \) which is itself a term, \( s \), shall be called an **occurrence** of **subterm** \( s \) of \( t \), denoted non-deterministically by \( s \in t \). This may also be pictured as \( t = \tilde{s} \).

Note that there may be several occurrences of the same subterm in a term. Unlike its term representation, the position (Definition 6.5) of an occurrence determines it uniquely. For disambiguation, the \( n \)-th occurrence of \( s \) in \( t \) may be denoted as \( (s \in t)_n \).

Terms have a **tree representation** as follows. A variable \( x \) is represented by the root labeled \( x \). A term \( f(t_1, \ldots, t_n) \) is represented by the root labeled \( f \) and by trees for \( t_1, \ldots, t_n \) as subtrees, ordered from left to right. Thus, the **root label** for a term \( t \) itself, if it is a variable, otherwise the constructor of \( t \).

**Access path** shall be defined as a variation of [Apt 1997, p. 27], and used to define **pendants**, which shall be needed for matching, and include disagreement pairs from [Robinson 1965].

**Definition 6.5** (access path and position of subterm). Let \( t \) be a term and consider an occurrence of its subterm \( s \), denoted as \( s \in t \). The **access path** of \( s \in t \) is defined as follows. If \( s = t \), then \( AP(s \in t) \) is the root label for \( t \). If \( t = f(t_1, \ldots, t_k) \) and \( s \in t_k \), then \( AP(s \in t) := f(k) \cdot AP(t_k) \).

By extracting the integers, we obtain the position of \( s \in t \). By extracting the labels, save for the last one, we obtain the **ancestry of \( s \in t \)**. If \( s_1 \in t_1 \) has the same position and ancestry as \( s_2 \in t_2 \), then we say \( s_1 \in t_1 \) and \( s_2 \in t_2 \) are **pendants** in \( t_1 \) and \( t_2 \). A **disagreement pair** between \( t_1 \) and \( t_2 \) is a pair of pendants therein differing in the last label.

For example, let \( t := [f(y), z] \) and \( s := z \). There is only one occurrence \( s \in t \). According to list definition, \( [f(y), z] = ([f(y), \cdot], (z, \text{nil})) \). Hence, \( AP([f(y), z]) = ([f(y), \cdot]) \cdot 2 \cdot (\cdot) \cdot 1 \cdot z \), so the position of \( s \in t \) is \( 2 \cdot 1 \) and its ancestry is \( (\cdot) \cdot (\cdot) \). An example of pendants: \( f([f(y), z]) \) and \( (g(a), b) \in [g(a, b), h(x)] \). This is also a disagreement pair.

6.1.2 A matching algorithm
Owing to the placeholder facility of relaxed core representation, the following algorithm is linear and rather succinct. In fact, without the placeholder facility it would be difficult to capture the error in matching \( f(x, z) \) on \( f(x, y) \) in just one pass along the terms and without auxiliary registers.

variable Let \( L \) be a variable. If \( L/S \in L \) and \( S \neq R \), then stop with FAILURE("divergence"). Otherwise, \( Match(L, R, \delta) := L \cup R \).

failure: shrinkage If \( L \) is a non-variable, but \( R \) is a variable, stop with FAILURE("shrinkage").

failure: clash If \( L \) and \( R \) are non-variables of different shape, stop with FAILURE("clash").

decomposition Let \( L = f(s_1, \ldots, s_n) \) and \( R = f(t_1, \ldots, t_n) \).
If there are \( \delta_1 := Match(s_1, t_1, \delta) \) and \( \ldots \) and \( \delta_n := Match(s_n, t_n, \delta_n-1) \), then \( Match(L, R, \delta) := \delta_n \).

Algorithm 6.1: One-pass term matching \( Match(L, R, \delta) \)
Theorem 6.6 (matching). Algorithm 6.1 solves the problem of matching $L$ on $R$: If Match$(L, R, \delta)$ stops with failure, then $L$ does not match $R$; otherwise, it stops with a substitution $\delta$ such that $[\delta]$ is a relevant matcher of $L$ on $R$. This follows from:

1. If Match$(L, R, \delta)$ stops with failure, there is no $\mu$ with $\mu(L) = R$ and $\mu \supseteq \delta$.

2. If Match$(L, R, \delta) = \delta'$, then $\delta'(L) = R$ and $\delta' \supseteq \delta$. In other words, Match$(L, R, \delta)$ is a matcher of $L$ on $R$ containing $\delta$. Additionally, $\delta'$ is complete for $L$ and $\text{Vars}(\delta') \subseteq \text{Vars}(L, R, \delta)$.

**Proof.** This algorithm clearly always terminates. For the Proof of Theorem 6.6 we need two observations, readily verified by structural induction:

- If $\sigma$ maps $L$ on $R$, then it maps any $s \in L$ on its pendant $t \in R$.
- Each time the algorithm visits one of the cases, the registers $L$ and $R$ denote either the original terms, or some pendants therein.

Thus, in the two middle non-variable cases there can be no matcher for the original terms, notwithstanding $\delta$.

In the variable case, the purported matcher $\sigma$ would have to map one variable on two different terms (Figure 1).

![Figure 1. Divergence: one variable, two terms](image)

**Proof of 2**. By structural induction. In case of variable, the claim holds. Assume we have a case of decomposition and the claim holds for the argument terms, i.e. $\delta_1(s_1) = t_1$, $\delta_1 \supseteq \delta$, $\delta_2(s_2) = t_2$, $\delta_2 \supseteq \delta_1$, ..., $\delta_n(s_n) = t_n$, $\delta_n \supseteq \delta_{n-1}$, and each $\delta_i$ is complete for $s_i$ as well as relevant. Due to completeness and Corollary 4.5 from $\delta_n \supseteq \ldots \supseteq \delta_2 \supseteq \delta_1$ follows $\delta_n(s_1) = \ldots = \delta_2(s_2) = t_1$ and so forth. Hence, $\delta_n(L) = R$. Clearly, $\delta_n \supseteq \delta$ and $\delta_n$ is complete for $L$ and relevant. As a final detail, recall that $\delta_n$ may contain passive pairs $x/x$, which are eliminated in $[\delta_n]$.

6.2 Generality

As an application of the matching algorithm [Algorithm 6.1] we can solve the problem of generality and equivalence between two substitutions.

**Definition 6.7** (more general). A substitution $\sigma$ is more general (or less instantiated) than a substitution $\theta$, written as $\sigma \leq \theta$[12], if $\sigma$ is a right-divisor of $\theta$, i.e. if there exists a substitution $\delta$ with the property $\theta = \delta \cdot \sigma$.

To check whether $\sigma \leq \theta$? One possibility would be to look for a counter-example, i.e. try to find a term $w$ such that for no renaming $\delta$ holds $\delta(\sigma(w)) = \theta(w)$. Let us call such a term a witness term. How to obtain a witness term? Intuitively, we may take $w$ to be the list of all variables of $\sigma, \theta$, denoted $w := \text{VarList}((\sigma, \theta))$, and see if we can find an impasse, i.e. some parts of $\sigma(w)$ that cannot possibly simultaneously be mapped on the respective parts of $\theta(w)$. It turns out this is sufficient.

**Theorem 6.8** (witness). $\sigma \leq \theta$ if and only if for some $w$ with $\text{Vars}(w) = \text{Vars}(\sigma, \theta)$ holds that $\sigma(w)$ matches $\theta(w)$.

---

1. It has also been said that $\sigma$ schematizes $\theta$ (Huet 1976).
2. Some authors like Jacobs and Langen (1992) and Amato and Scozzari (2009) turn the symbol $\leq$ around. Indeed the choice may appear to be arbitrary. But we shall stick to the notion that a more general object is "smaller", because it correlates with the "smallness" of the substitution stack.

---

**References**


---

Proof. If $\delta \cdot \sigma = \theta$, then surely $\delta(\sigma(w)) = \theta(w)$.

For the other direction, assume there is $\mu$ with $\mu(\sigma(w)) = \theta(w)$. By [Theorem 6.6] we can choose the matcher $\mu$ to be relevant, so $\text{Vars}(\mu) \subseteq \text{Vars}(\sigma, \theta)$. If for some $x \in V$ holds $\mu(\sigma(x)) \neq \theta(x)$, then clearly $x \notin \text{Vars}(\sigma, \theta)$, hence the inequality becomes $\mu(x) \neq x$, meaning $x \in \text{Core}(\mu)$, which is impossible. 

As a consequence, we obtain a simple visual criterion.

**Corollary 6.9** (witness). The relation $\sigma \leq \theta$ does not hold, iff for some $w$ with $\text{Vars}(w) \subseteq \text{Vars}(\sigma, \theta)$ any of the following holds:

1. At some corresponding positions, $\sigma(w)$ exhibits a non-variable, and $\theta(w)$ exhibits a variable ("shrinkage"), or a non-variable of a different shape ("clash").

2. $\sigma(w)$ exhibits two occurrences of variable $x$, but at the corresponding positions in $\theta(w)$ there are two mutually distinct terms ("divergence").

For the search for an impasse can be performed by [Algorithm 6.1] via $\text{Match}(\sigma(w), \theta(w), \varepsilon)$, where $\varepsilon := \text{VarList}((\sigma, \theta))$.

If no impasse is found, the algorithm produces $\delta$ such that $\theta = \delta \cdot \sigma$. Some test runs are in [Figure 2] and [Figure 3].

**Example 6.10.** The subtlety of the relation "more general" is illustrated in [Apt 1997] with the following example: $\sigma := (x / a)$ is more general than $(a / a)$, but not more general than $\theta := (x / a)$. The former claim is justified by $(x / a) = (x) \cdot (a)$. The matcher was here not difficult to guess, but in general may be, and can always be found by [Algorithm 6.1] (Figure 2).

The latter is a simplified form of a counter-example by Haiping Ko (reported in [Apt and Seldin 1994]), which was pivotal in showing that the strong completeness theorem for SLD-derivation (Lloyd 1987) needed a revision. The Ko example purports that $\sigma := (x / f(y, a))$ is not more general than $\theta := (x / f(y, a))$, where $y, z$ are distinct variables. For proof, it was observed: if $\delta \cdot (x / f(y, a)) = (x / f(y, a))$, then $y/a, z/a \in \delta$, therefore even if one of $y, z$ is equal to $x$, at least one of bindings $y/a, z/a$ has to be in $\delta \cdot (x / f(y, a))$. To alleviate the need for such "ad-hoc", custom-made proofs, [Algorithm 6.1] could be used, giving divergence (Figure 3).

![Figure 2. Successfull check on \(\leq\)](image)
Figure 3. Failed check on ϵ

Legacy 6.11 (equivalence). \( \theta \) is more general than \( \theta' \) and \( \theta' \) is more general than \( \theta \) iff for some renaming \( \rho \) such that \( \text{Vars}(\rho) \subseteq \text{Vars}(\theta) \cup \text{Vars}(\theta') \) holds \( \rho \cdot \theta = \theta' \).

With some practice, such a renaming \( \rho \) can be guessed, or simply constructed by Algorithm 6.1.

Example 6.12. Since \( \begin{pmatrix} y & z \\ x & y \end{pmatrix} \) is a renaming and \( \begin{pmatrix} y & z \\ x & y \end{pmatrix} \cdot \begin{pmatrix} y & z \\ x & y \end{pmatrix} = \epsilon \), we have \( \begin{pmatrix} y & z \\ x & y \end{pmatrix} \sim \epsilon \). In other words, if we permute two variables, that amounts to "doing nothing". This is a much-cited example of counter-intuitive character of equivalence. In fact, due to group structure of finite permutations, any two renamings are bound to be equivalent, so permuting any number of variables amounts to doing nothing.

Remark 6.13 ("\( \sim \)" is not compositional). Equivalence is not compatible with composition, as shown in [Eder 1985]. Let \( \sigma := \begin{pmatrix} y & z \\ x & y \end{pmatrix} \), \( \sigma' := \begin{pmatrix} y & z \\ x & y \end{pmatrix} \) and \( \theta := \begin{pmatrix} y & z \\ x & y \end{pmatrix} \). Then \( \sigma \sim \sigma' \), but \( \theta \cdot \sigma \neq \begin{pmatrix} y & z \\ x & y \end{pmatrix} \). The non-equivalence is verified by Algorithm 6.1.

6.4 Unification

Definition 6.14 (unification). Let \( s \) and \( t \) be terms. If there is a substitution \( \theta \) such that \( \theta(s) = \theta(t) \), then \( s \) and \( t \) are said to be unifiable; and \( \theta \) is their unifier, the set of all such being \( \text{Unif}(s, t) \). It is a relevant, if \( \text{Vars}(\theta) \subseteq \text{Vars}(s) \cup \text{Vars}(t) \). A unifier \( \theta \) of \( s \) and \( t \) is their most general unifier (mgu), if it is more general than any other unifier; the set of all such is \( \text{Mgu}(s, t) \) := \{ \theta \in \text{Unif}(s, t) \mid \text{for every } \alpha \in \text{Unif}(s, t) \text{ holds } \theta \leq \alpha \} \).

A set of equations \( \{ a_1 = b_1, ..., a_n = b_n \} \) may be condensed to one equation like \( f(a_1, ..., a_n) = f(b_1, ..., b_n) \), and vice versa, which shows that unifying two terms and unifying arbitrarily many terms are the same task. So the notions of unifier and mgu can be extended from a single equation to a set of equations \( E \) by defining \( \text{Unif}(E) := \{ \theta \mid \text{for every } (s=t) \in E \text{ holds } \theta(s) = \theta(t) \} \). Similarly for \( \text{Mgu}(E) \). A set of equations is in solved form if it is of the form \( \{ x_1 = t_1, ..., x_n = t_n \} \) where all \( x_i \) are distinct and none of them occurs in any \( t_j \).

If \( \sigma \in \text{Mgu}(s, t) \), then for any renaming \( \rho \) by Corollary 3.3 \( \rho \cdot \sigma \in \text{Mgu}(s, t) \). In fact, any element from \( \text{Mgu}(s, t) \) has this form, as a consequence of Legacy 6.11.

Legacy 6.15 (equivalence of mgus). Let \( \mu \in \text{Mgu}(E) \). Then \( \mu' \in \text{Mgu}(E) \) if there is a renaming \( \rho \) such that \( \text{Vars}(\rho) \subseteq \text{Vars}(\mu) \cup \text{Vars}(\mu') \) and \( \rho \cdot \mu = \mu' \).

Thus, the set \( \text{Mgu}(s, t) \) is either empty or infinite.\(^{14}\) As a meta-function, \( \text{Mgu} \) has two pleasing properties: it is compatible with renaming and it is compatible with LD-resolution.

Lemma 6.16 (renaming compatibility of \( \text{Mgu} \)). For every \( \rho \) and \( E \) holds \( \text{Mgu}(\rho(E)) = \rho(\text{Mgu}(E)) \).

Proof. This follows from Theorem 5.24 and Corollary 3.3. Assume \( \sigma \in \text{Mgu}(s, t) \), then \( \rho(\sigma)(\rho(s)) = \rho(\sigma(s)) = \rho(\sigma(t)) = \rho(\sigma)(\rho(t)) \). 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 \( \theta = \delta \cdot \sigma \cdot \rho^{-1} = \delta \cdot \rho^{-1} \cdot \rho \cdot \sigma \cdot \rho^{-1} = (\delta \cdot \rho^{-1}) \cdot \rho(\sigma) \), meaning \( \rho(\sigma) \in \text{Mgu}(\rho(E)) \). For the other direction, observe \( \theta = \rho \cdot \rho^{-1} \cdot \delta \cdot \sigma \cdot \rho^{-1} = (\rho \cdot \rho^{-1} \cdot \delta \cdot \sigma) \).

Compatibility of \( \text{Mgu} \) with LD-resolution, also called iteration property, is proved in [Apel 1997].

Legacy 6.17 (iteration for \( \text{Mgu} \)). 1. Let \( E_1, E_2 \) be sets of equations. If \( \sigma \) is a mgu of \( E_1 \) and \( \mu \) is a mgu of \( E_2 \), then \( \sigma \cdot \mu \) is a mgu of \( E_1 \cup E_2 \).

2. Moreover, if \( E_1 \cup E_2 \) is unifiable, then there exists a mgu \( \sigma \) of \( E_1 \), and for each such \( \sigma \) there exists a mgu \( \theta \) of \( E_2 \).

6.4.1 Unification by algorithm

For any two unifiable terms \( s, t \) holds that \( \text{Mgu}(s, t) \) is an infinite set. On the other hand, any particular unification algorithm \( \Xi \) 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 \( \Xi(s, t) \), the algorithmic (or concrete) mgu of \( s \) and \( t \), produced by algorithm \( \Xi \).

The task of unification was introduced and solved in [Robinson 1965]. Another classical unification algorithm is [Martelli and Montanari 1982], based on [Herbrand 1930]. The algorithm is usually given in non-deterministic form, here denoted as \( \Xi_{\text{mut}} \), but it can be made deterministic using sequences instead of sets and picking the leftmost equation eligible for a rule application, as observed in [Apel 1997] p. 36. The resulting algorithm shall be denoted \( \Xi_{\text{mut}} \).

6.4.2 ... and iteration property

As opposed to \( \text{Mgu} \), any particular unification algorithm like \( \Xi_{\text{mut}} \) does not have to satisfy the iteration property. But the deterministic version \( \Xi_{\text{mut}} \) does.

Example 6.18 (no iteration for \( \Xi_{\text{mut}} \)). Let \( E_1 := \{ (x, y, z) \} \) and \( E_2 := \{ (z = f(x)) \} \). If we pick the equation \( y = x \) for binding (denoted by underlining), we obtain \( E_1 = \{ x = y, y = z \} \Rightarrow \{ x = y, y = z \} \Rightarrow \{ y = x \} \Rightarrow \xi \Rightarrow \mu \Rightarrow \theta \). However, for unifying \( E_1 \cup E_2 \) we may as well pick \( x = y \) for binding, and get \( E_1 \cup E_2 = \{ x = y, y = z, z = f(x) \} \Rightarrow \{ x = y, y = z, z = f(x) \} \Rightarrow \xi \Rightarrow \mu \Rightarrow \theta \).

Lemma 6.19 (iteration for \( \Xi_{\text{mut}} \)). Assume \( \sigma = \Xi_{\text{mut}}(E') \) and \( \theta = \Xi_{\text{mut}}(E'') \). Then \( \Xi_{\text{mut}}(E', E'') \) = \( \theta \).

Proof. 1. From Legacy 6.17, we know that \( E', E'' \) is unifiable.

The deterministic version of \( \Xi_{\text{mut}} \) transforms an equation sequence from left to right. This has the nice consequence that \( \Xi_{\text{mut}} \) chooses the same equations to transform in \( E' \) as in \( E', E'' \) for so
long as $E'$ has not reached its solved form. The only interesting steps here are binding steps. Underlined is the next candidate for binding, which is afterwards shaded, to signify that this pair now went "passive", i.e. it cannot be elected for further transformations and may merely get its right-hand side further instantiated.

\[ \mathcal{A}(E', E'') = \mathcal{A}(E, E'') \]

1. there is $\mathcal{A}(G = \text{Head}(L)) = \emptyset$ iff there is $\mathcal{A}(G = \text{Head}(L)) = \lambda((\emptyset))$

2. $\mathcal{K}$ is effective on $G$ with some mgu $\sigma$ iff there is renaming $\rho$ with $\theta = \rho \cdot \tilde{X}(\sigma)$

Additionally, if $\sigma$ is relevant, then $\tilde{X}(\sigma) = \lambda(\sigma)$. If furthermore $\mathcal{K}$ produces relevant mgus, then $\text{Vars}(\rho) \subseteq \text{Vars}(G) \cup \text{Vars}(L)$.

Proof. For the first part, $\tilde{X}(G = \text{Head}(L)) = \mathcal{A}(G = \text{Head}(L))$ is due to $\mathcal{A}(G = \text{Head}(L)) = \mathcal{A}(G = \text{Head}(L))$, which is due to Axiom 6.20. $\mathcal{X}(G = \text{Head}(L)) = \mathcal{A}(G = \text{Head}(L))$, hence if $\theta := \mathcal{A}(G = \text{Head}(L))$, then $\tilde{X}(\sigma) = \lambda^{-1}(\emptyset)$. For the second part, note that $\sigma$ and $\mathcal{A}(G = \text{Head}(L))$ are two mgus for the same unification task. Hence, by [Lemma 6.15] there is renaming $\delta$ with

\[ \text{Vars}(\delta) \subseteq \text{Vars}(\sigma) \cup \text{Vars}(\mathcal{A}(G = \text{Head}(L))) \]

such that $\mathcal{A}(G = \text{Head}(L)) = \theta \cdot \sigma$. From $\lambda^{-1}(\emptyset) = \delta \cdot \sigma$ we get $\theta = \tilde{X}(\delta) = \lambda(\sigma)$. By assigning $\rho := \mathcal{A}(\delta)$ we obtain the claim.

Due to this consideration. If $\sigma$ is relevant, $\mathcal{A}(G = \text{Head}(L)) \subseteq \text{Vars}(G) \cup \text{Vars}(G)$. On the other hand, $\tilde{X}(\sigma) = \mathcal{A}(G = \text{Head}(L)) \setminus G \not\subset L$, hence $\tilde{X}(\lambda(\sigma))$ is relevant, and $\tilde{X}(\sigma)$ is in $\mathcal{A}(\lambda(\sigma))$. Lastly, if both $\sigma$ and $\mathcal{A}(G = \text{Head}(L))$ are relevant, from (2) follows $\mathcal{A}(\rho) \subseteq \text{Vars}(G) \cup \text{Vars}(L)$.

Notation 6.22 (equivalewid modulo prenaming). If for some renaming $\lambda$ holds $\emptyset \sim \lambda(\sigma)$, then we also write $\emptyset \sim \lambda(\sigma)$.

With this notation, in the previous claim we would have had $\emptyset \sim \sigma$. Assuming relevant mgus, this can be further simplified to $\emptyset \sim \sigma$. Remark 6.23. The relationship from Lemma 6.21 is symmetrical. Let $\mu := \text{Pre}(L, K)$, then $\tilde{P}(\lambda)$ is $\text{L}(\lambda)$. From $\theta := \rho \cdot \tilde{X}(\sigma)$ follows $\lambda^{-1}(\rho \cdot \theta) = \sigma$. By assigning $\delta := \lambda^{-1}(\rho \cdot \theta)$ we obtain $\sigma = \delta \cdot \tilde{X}(\theta)$, or $\sigma = \rho \cdot \tilde{X}(\theta)$. If $\theta$ is relevant, $\mathcal{A}(\theta) \setminus \mathcal{A}(L, G)$, so due to $G \not\subset L$ and $\tilde{X}(\mathcal{A}(\lambda(\mu)) \setminus \text{Vars}(L)$ we have $\theta \not\subset \mathcal{A}(\lambda(\mu))$, thus $\tilde{P}(\mu) = \lambda(\mu)$, and $\sigma \sim_{\mu} \theta$.

### 6.5 Idempotence

Recall that a substitution $\theta$ satisfying the equality $\theta \cdot \theta = \theta$ is called idempotent. Any two unifiable terms have an idempotent (and relevant) most general unifier, as provided by classical unification algorithms.

Remark 6.24 (idempotence is not compositional). As illustrated by [Edel 1985], composition of two idempotent substitutions does not have to be idempotent — not even equivalent to an idempotent substitution. Example: $\sigma := (x f(y))$ and $\theta := (y f(z))$ give $\sigma \cdot \theta = (x f(y) y f(z))$.

However, $\theta \cdot \sigma = (y f(z))$ is idempotent. This is an instance of a useful property from [Apollonio 1997].

Legacy 6.25 (two idempotence criteria). Let $\sigma, \theta \in \text{substitutions}$.

1. $\theta$ is idempotent iff $\text{Core}(\theta) \cap \mathcal{A}(\text{Ran}(\theta)) = \emptyset$.

2. Let $\sigma$ and $\theta$ be idempotent. If $\text{Vars}(\text{Ran}(\sigma)) \not\subset \text{Core}(\sigma)$, then $\theta \cdot \sigma$ is also idempotent.

The first criterion is quite intuitive: if the active domain and the active range of a substitution $\sigma$ have no variables in common, then all the variables from the active domain shall be released from the term $t$ after the application of $\sigma$ upon $t$. Therefore, a repeated application of $\sigma$ cannot change anything.
By the unruly charm of substitutions, \((\varepsilon\ y)\) is an mgu for \(t = t\), yet surely not the expected one. It is lucky that idempotency prevents such surprises:

**Lemma 6.26** (pertinence). If \(\sigma\) is idempotent and \(\sigma \in \text{Mgu}(t, t)\) for some \(t\), then \(\sigma = \varepsilon\).

**Proof.** Since we know that \(\varepsilon \in \text{Mgu}(t, t)\), by [Legacy 6.15] there has to hold \(\varepsilon \sim \sigma\), so there is a renaming \(\rho\) such that \(\rho \circ \varepsilon = \sigma\). If \(\sigma\) is idempotent, by [Legacy 6.25] we know that \(\text{Core}(\sigma) \cap \text{Vars}\text{Ran}(\sigma) = \emptyset\). The only renaming with this property is \(\varepsilon\), since for a renaming always holds \(\text{Core} = \text{Ran}\) [Legacy 3.4].

It turned out that relevance is a mandatory property of an idempotent mgu [Apr1997]. This shall come in handy for **subsection 7.2**. We give a direct proof using witness term.

**Legacy 6.27** (relevance). Every idempotent mgu is relevant.

**Proof.** Assume \(\sigma \in \text{Mgu}(E)\) is idempotent, but not relevant, i.e. there is \(z \in \text{Vars}(\sigma)\) such that \(z \not\in \text{Vars}(E)\). Let us show that \(\sigma\) cannot be an mgu, by finding a unifier \(\theta\) of \(E\) such that \(\sigma \not\subseteq \theta\). Technically, we construct \(\theta\) and a witness term \(w\) such that for no \(\delta\) can hold \(\delta(\sigma(w)) = \theta(w)\), as outlined in **Corollary 6.9**.

Case 1: \(z \in \text{Core}(\sigma)\): Here we choose \(\theta := \sigma[z/\varepsilon]\). If \(\sigma\) is an idempotent unifier of \(E\), then so is \(\theta\).

Subcase 1: \(\sigma(z) = \emptyset\) is ground. Since \(\theta(z) = \varepsilon\), the witness can be \(w := z\) (shrinkage). Subcase 2: \(\sigma(z)\) contains a variable, say \(x\), pictured as \(\sigma(z) = [x/z]\). Due to idempotency of \(\sigma\), holds \(x \not\in \text{Core}(\sigma)\), so \(x \neq z\). We get \(\sigma([x, z]) = [x, \varepsilon]\), whereas \(\theta([x, z]) = [x, \varepsilon]\). So with \(w := [x, z]\) we have divergence (if \(\sigma(z) = x\)) or shrinkage (otherwise).

Case 2: \(z \notin \text{Core}(\sigma)\), but \(z \in \text{Ran}(\sigma)\): There is \(x \in \text{Core}(\sigma)\) (and therefore \(x \neq z\)) such that \(\sigma(x) = [x]\). Here we take \(\theta\) to be an idempotent and irrelevant mgu of \(E\) (e.g. the outcome of a classical unification algorithm). Due to relevance, \(z \not\in \text{Vars}(\theta)\). We get \(\sigma([x, z]) = [z, \varepsilon]\), whereas \(\theta([x, z]) = [z, \varepsilon]\) (divergence). \(\diamondsuit\)

### 6.6 Restriction

**Remark 6.28** (restriction is not compositional). In general, \((\sigma \cdot \theta)[W] = \sigma[W] \cdot \theta[W]\) does not hold. Take \(\theta := \langle u \rangle, \sigma := \langle y \rangle\).

\(W := \{x\}\). Then \(\sigma \cdot \theta = \langle y \rangle \cdot \langle x \rangle = \langle y \rangle\), \((\sigma \cdot \theta)[W] = \langle y \rangle\),

but on the other hand, \(\sigma[W] = \varepsilon, \theta[W] = \theta, \sigma[W] \cdot \theta[W] = \langle y \rangle\).

On the plus side, restriction is compatible: \(\rho(\sigma[W]) = \rho(\sigma) \cdot \rho(W)\). Also, by [Legacy 6.25] any restriction of an idempotent substitution is itself idempotent.

### 7. Claims for logic programming systems

Looking for the meaning of logic programming, there are three levels to consider: logical level, which is Horn-clause logic (HCL) and its extensions; proof method, based on SLD-resolution, for handling the question of logical consequence for HCL; and implementation level, which uses fixed algorithms for mgu, standardization-apart and search. The first two levels have been extensively studied and it is known that SLD-resolution and its special case LDR-resolution are handling the question of logical consequence in a sound and complete way for HCL. The third level is by its nature more a subject of technical than of theoretical interest. The latter seems to have culminated in the assertion that the usual search strategy of Prolog, depth-first search, is incomplete. Yet, it is our belief that there is an interesting theoretical side to the logic programming systems as well.

### 7.1 Compatibility claims: partly preserved

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

- most general unifier is provided by a fixed algorithm \(\mathfrak{A}\)
- standardization-apart is provided by a fixed algorithm \(\mathfrak{S}\)

#### 7.1.1 Renaming-compatibility

If we have an SLD-derivation, it is now not possible to just rename it wholesale (the resolvents, the mgu, the input clauses), which was possible in Horn clause logic, by virtue of [Corollary 3.3]. This is because the two fixed algorithms do not have to be renaming-compatible – in fact, the second one cannot be.

To see this, assume a standardization-apart algorithm has for the query \(G := p(X, Y)\) at the tip of a derivation \(D\) assigned the input clause \(K := \langle p(U, V), q(U, W)\rangle\). With renaming \(\rho = \langle W, U, V \rangle\), and assuming the algorithm is renaming-compatible, the query \(\rho(G) = G\) at the tip of the derivation \(\rho(D) = D\) would need to be assigned the input clause \(\rho(K) = \langle p(W, V), q(W, U)\rangle\). But already something else has been assigned to it.

As a consequence, the handling of local variables in **subsection 7.2** shall require some more attention.

#### 7.1.2 Resolution-compatibility ("iteration property")

As seen in **subsubsection 6.4.1**, \(\mathfrak{R}\) does not but \(\mathfrak{A}\) does satisfy the iteration property. The iteration property (or resolution compatibility) states that the particular unification algorithm works the same as the LD-resolution algorithm on a sequence of equations represented via predicate \(eq/2\) defined as \("eq(X, X)\"."

Iteration property is important for compositional formal semantics of logic programming like S1:PP.

### 7.2 Variant lemma revisited

For logic programming implementations complying with [Axiom 6.20] and yielding relevant mgu, that is to say for all of them\(^1\) a propagation result can be proved, which leads to a constructive and incremental version of the variant lemma.

Assume the program 
\[
\text{son}(S) \leftarrow \text{male}(S), \text{child}(S, P).,\]

and let us enquire about \text{son} in two derivations [Table 1]. If we know that one query, \text{son}(X), is a variant of the other, \text{son}(A), does the same connection hold between the resolvents as well?

As can be seen from [Table 1] in a resolution some new variables may crop up, originating from standardization-apart in cases where the clause body has variables not present in the head. For the purposes of this paper let us call them **local variables**, as opposed to **query variables**. Were it not for local variables, the resolvents in both derivations would clearly be variants, with the same pre-naming as for the original queries. Yet, even though the variables new in one derivation do not have to be new in the other [Table 1], at least the pre-naming can be extended to accommodate those local variables. The claim is proved in a constructive manner.

<table>
<thead>
<tr>
<th>query</th>
<th>input clause</th>
<th>resolvent</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{son}(X)</td>
<td>\text{son}(B) \leftarrow \text{male}(B), \text{child}(B, A).</td>
<td>\text{male}(X), \text{child}(X, A)</td>
</tr>
<tr>
<td>\text{son}(A)</td>
<td>\text{son}(X) \leftarrow \text{male}(X), \text{child}(X, B).</td>
<td>\text{male}(A), \text{child}(A, B)</td>
</tr>
</tbody>
</table>

\(^1\)Classical unification algorithms not only satisfy [Axiom 6.20] but also yield idempotent mgu. Idempotent mgu are always relevant [Legacy 6.27].

10 2016/6/30
Lemma 7.1 (propagation of variance). Assume a unification algorithm \( \mathcal{A} \) satisfying Axiom 6.20. Assume an SLD-derivation \( D \) ending with \( G \) and an SLD-derivation \( D' \) ending with \( G' \) such that \( \alpha(G) = G' \) for some prenames \( \alpha \) which is complete for \( G \) and relevant for \( D \) and \( D' \).

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

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

The claim can be summarized in Figure 4 and the role of relevance in Figure 5.

\[
\begin{align*}
G & \nrightarrow_{D} H \\
\alpha & \nleftarrow_{\alpha \uplus \lambda} \alpha \uplus \lambda \\
G' & \nrightarrow_{D} H'
\end{align*}
\]

Figure 4. Propagation of variance

\[
\begin{align*}
\text{son}(X) & \nrightarrow_{D} \text{male}(X), \text{child}(X, A) \\
\alpha & = (\times, c) \quad \alpha \uplus \lambda \\
\text{son}(A) & \nrightarrow_{D} \text{male}(A), \text{child}(A, B)
\end{align*}
\]

Figure 5. ...is not always possible

Proof of 3: By definition, \( C(\lambda) \subseteq \text{Vars}(K) \) and \( R(\lambda) \subseteq \text{Vars}(K') \). Hence, and due to relevance of \( \alpha \), \( C(\alpha \uplus \lambda) = C(\alpha) \cup C(\lambda) \subseteq \text{Vars}(D') \cup \text{Vars}(K') \subseteq \text{Vars}(D \nleftarrow_{K, \sigma} K' \nleftarrow_{\sigma} H' \nleftarrow_{\alpha \uplus \lambda} H) \). Similarly, \( R(\alpha \uplus \lambda) \subseteq \text{Vars}(D' \nleftarrow_{K', \sigma'} H' \nleftarrow_{\alpha \uplus \lambda} H) \), therefore \( \alpha \uplus \lambda \) is relevant for \( D \nleftarrow_{K, \sigma} H \) to \( D' \nleftarrow_{K', \sigma'} H' \).

Proof of 7: By (5) and (6), \( \text{Vars}(H) \subseteq \text{Vars}(G) \cup \text{Vars}(K') \subseteq C(\alpha) \cup C(\lambda) = C(\alpha \uplus \lambda) \), meaning \( \alpha \uplus \lambda \) is complete for \( H \).}

Definition 7.2 (similarity). SLD-derivations of the same length of \( G \nleftarrow_{\lambda} G_1, G_2, ..., G_n \) 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 \( K_i \) and \( K'_i \) are variants of the same program clause.

That the name “similarity” is justified, follows from the claim known as variant lemma (Lloyd 1987). Lloyd and Shepherson (1991), (Apr 1997), here in the formulation from (Doets 1993).

Legacy 7.3 (variant). Finite derivations which are similar and start from variant queries have variant resultants.

For logic programming systems obeying Axiom 6.20 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 10), yet the added conclusion has substance: first, renaming a query 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 7.4 (variant claim for logic programming systems). Assume a unification algorithm \( \mathcal{A} \) satisfying Axiom 6.20 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 (12). Then for every \( i = 1, ..., n \) holds \( G'_i = \beta_i(G_i) \). Thus \( \sigma' = \beta_i(\sigma_i') \) and \( \sigma_i' \) are similar to \( \sigma_i \), where \( \beta_i := \alpha \uplus \lambda_i \uplus \ldots \uplus \lambda_i \), \( \alpha := \text{Pren}(G, G') \) and \( \lambda_i := \text{Pren}(K_i, K'_i) \).
Proof. By assumption, $G$ and $G'$ are variants, so
\[
\alpha := \text{Pren}(G, G')
\] (13)
exists. Clearly, $\alpha$ is complete for $G$, since $\text{Var}(G) = C(\alpha)$. By
construction, $\alpha$ is also relevant for $D_0 := G$ to $D_0' := G'$.

We may iterate [Lemma 7.1] obtaining for every $i = 1, \ldots, n$
\[
\sigma_i' = (\alpha \cup \lambda_1 \cup \ldots \cup \lambda_i)(\sigma_i)
\] (14)
\[
G_i' = (\alpha \cup \lambda_1 \cup \ldots \cup \lambda_i)(G_1)
\] (15)
where $\lambda_i := \text{Pren}(K_i, K_i')$. Therefore, $\sigma_i' \cdot \sigma_{i-1}' \cdot \ldots \cdot \sigma_1' = (\alpha \cup \lambda_1 \cup \ldots \cup \lambda_i)(\sigma_i) \cdot (\alpha \cup \lambda_1 \cup \ldots \cup \lambda_{i-1})(\sigma_{i-1}) \ldots (\alpha \cup \lambda_1)(\sigma_1)$.

Let $k < i$. Since $\text{Var}(\sigma_k) \subseteq \text{Var}(G) \cup \text{Var}(K_k) \cup \ldots \cup \text{Var}(K_{a_k}) \subseteq C(\alpha) \cup C(\lambda_1) \cup \ldots \cup C(\lambda_k) = C(\alpha \cup \lambda_1 \cup \ldots \cup \lambda_k)$, by [Theorem 1.17] $\alpha \cup \lambda_1 \cup \ldots \cup \lambda_k \cup \lambda_k'$ is safe for $\sigma_i$ and
$(\alpha \cup \lambda_1 \cup \ldots \cup \lambda_k)(\sigma_k) = (\alpha \cup \lambda_1 \cup \ldots \cup \lambda_k)(\sigma_k)$. Hence,
\[
(\alpha \cup \lambda_1 \cup \ldots \cup \lambda_{i-1})(\sigma_{i-1}) = (\alpha \cup \lambda_1 \cup \ldots \cup \lambda_i)(\sigma_{i-1})
\]...
(16)
\[
(\alpha \cup \lambda_1)(\sigma_1) = (\alpha \cup \lambda_1 \cup \ldots \cup \lambda_i)(\sigma_1)
\] (17)
$G = (\alpha \cup \lambda_1 \cup \ldots \cup \lambda_i)(G)$

Let us abbreviate $\beta_i := (\alpha \cup \lambda_1 \cup \ldots \cup \lambda_i)$. Then from (14) and (16) by [Theorem 5.25]
\[
\sigma_i' \cdot \sigma_{i-1}' \ldots \sigma_1' = \beta_i(\sigma_i) \cdot \beta_{i-1}(\sigma_{i-1}) \ldots \beta_1(\sigma_1) = \beta_i(\sigma_i \cdot \sigma_{i-1} \ldots \sigma_1)
\] (18)
which is the promised connection between partial answers.

Clearly, variance of partial answers means variance of complete answers, and c.a.s. and results as well: For the cases when $G_n = \emptyset$, we obtain, by (13), the expected relationship between the respective complete answers: $\sigma_n' \cdot \ldots \cdot \sigma_1' = \beta_n(\sigma_n \ldots \sigma_1)$. A c.a.s. differs from our complete answer by the added restriction on the query variables. Due to renaming-compatibility of restriction, (18) and $\beta_n(\sigma_1) = \sigma_n \equiv \sigma'(\sigma_n \ldots \sigma_1)$, we obtain
\[
\text{R}(\alpha \cdot \sigma_1 \ldots \sigma_n)(G) = G
\] (13) and (15)
\[
= \beta_i(\sigma_i \ldots \sigma_1)(G) \equiv \beta_i(G),
\] (17) and
\[
= \beta_i((\alpha \cup \lambda_1 \cup \ldots \cup \lambda_i)(G)) \equiv \beta_i(G), \text{by [Theorem 5.24]}
\]

Example 7.5 (similarity). Assume the program
\[
\text{son}(S) \leftarrow \text{male}(S), \text{child}(S, P).
\%
K_1
\]
\[
\text{male}(c). \text{male}(d). \text{child}(a, d).
\%
K_2, K_3, K_4
\]
An interpreter for L.D-resolution may produce derivations
\[
\text{son}(A) \leftarrow K_1 \cdot \sigma_1 \text{male}(A), \text{child}(C, A) \leftarrow K_2 \cdot \sigma_2 \text{child}(C, d)
\]
\[
\text{son}(B) \leftarrow K_2 \cdot \sigma_1 \text{male}(B), \text{child}(D, B) \leftarrow K_3 \cdot \sigma_2 \text{child}(D, d)
\]

They are obviously similar, with $K_1 = K_3[X, C], K_1' = K_3[Y, D], K_2 = K_2'$ and $K_4 = K_3$. The variables $X, Y$ stand for actual
used variables, which cannot be deduced from the form of queries. From the queries, input clauses and resolvents we can further deduce relevant mngs $\sigma_1 = (C), \sigma_2 = (\bar{D}), \sigma_2 = (\bar{A})$ and $\sigma_2 = (\bar{C})$. The mappings are $\alpha = (C), \lambda_1 = (X), \lambda_2 = (\bar{X})$. Clearly, they fulfill $(\alpha \cup \lambda_1)(\text{male}(A), \text{child}(C, A)) = \text{male}(B), \text{child}(D, B)$ and $(\alpha \cup \lambda_1)(\text{male}(C, d)) = \text{child}(D, d)$, and so on.

Observe that in step 1' there is a relevant mgu $(\sigma_2')$ as well, but even without knowing the resolvent, we know that $(\sigma_2')$ couldn't have been employed, due to renaming compatibility of the interpreter's unification algorithm.

8. Outlook
There are two main contributions in this paper, the concept of pre

naming and an algorithm for term matching. By relaxing the core representation and forgoing permutation requirement for renaming, the concept of prenaming is obtained. Its use for incremental claims concerning implemented logic programming systems like propagation of variance is shown [Lemma 7.1, Theorem 7.4]. There, prenaming made it possible to keep track of local variables in an incremental fashion. Relaxed core representation is also used for a novel term matching algorithm, [Algorithm 6.1] that solves the problem of checking substitution generality.

References
Verzeichnis der zuletzt erschienenen Informatik-Berichte

[364] Güting, R.H., Behr, T., Düntgen, C.:
Book Chapter: Trajectory Databases; 5/2012

[365] Paul, A., Rettinger, R., Weihrauch, K.:
CCA 2012 Ninth International Conference on Computability and Complexity in Analysis (extended abstracts), 6/2012

[366] Lu, J., Güting, R.H.:
Simple and Efficient Coupling of a Hadoop With a Database Engine, 10/2012

[367] Hoyrup, M., Ko, K., Rettinger, R., Zhong, N.:
CCA 2013 Tenth International Conference on Computability and Complexity in Analysis (extended abstracts), 7/2013

[368] Beierle, C., Kern-Isberner, G.:
4th Workshop on Dynamics of Knowledge and Belief (DKB-2013), 9/2013

Symbolic Trajectories, 12/2013

[370] Bortfeldt, A., Hahn, T., Männel, D., Mönch, L.:
Metaheuristics for the Vehicle Routing Problem with Clustered Backhauls and 3D Loading Constraints, 8/2014

[371] Güting, R. H., Nidzwetzki, J. K.:
DISTRIBUTED SECONDO: An extensible highly available and scalable database management system, 5/2016