INFORMATIK
BERICHTE

372 – 07/2016

A practical view on substitutions

Marija Kulaš

FernUniversität in Hagen

Fakultät für Mathematik und Informatik
D-58084 Hagen

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 aleviates 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 $\binom{x}{f(y,z)}$ is not more general than $\binom{x}{f(a,a)}$, which admittedly does look rather counter-intuitive, but complies with the definition of substitution generality (Example 6.10). Second, by composing substitutions, properties like equivalence (Remark 6.13), idempotency (Remark 6.24), or restriction (Remark 6.28) are not preserved. Third, due to group structure of renamings, permuting any number of variables amounts to "doing nothing", as in $\binom{x\ y}{y\ x} \sim \varepsilon$ (Example 6.12). Such equivalences are also felt to be counter-intuitive. Hence the prevalent sentiments that substitutions are "quite hard matter to deal with" (Palamidessi 1990) or "very tricky" (Shepherdson 1994). In an effort to avoid substitutions as much as possible, *resultant*s were proposed (Lloyd and Shepherdson 1991). Still, for almost anyone embarking on a journey of logic program analysis or formal semantics, sooner or later the need for renaming terms and generally handling substitutions arises.

In the case of this author, the need arose while trying to prove completeness of an operational semantics for pure Prolog, S1:PP (Kulaš 2005), and facing the following *extensibility problem*. Starting from a pair of variant queries, their respective formal derivations proceed to develop. At each step, new variables may crop up, but the status of being variant should be maintained. This setup is known from the classical *variant lemma* (Legacy 7.3). A new aspect here is that we need to *collect* the variables, obtaining at each step the temporary variance between the derivations.

How to model this process of accumulating new variables and their correspondence between derivations? Surely by renamings? Assume the first query is $p(z,u)$ and the second $p(y,z)$. There is only one relevant renaming, $\rho = \binom{z\ u\ y}{y\ z\ u}$. Now assume in the next step the first derivation acquires the variable y, and the second x. The relevant renaming this time would be $\rho' = \binom{z\ u\ y\ x}{y\ z\ x\ u}$. Clearly, ρ' is not an extension of ρ, which makes it seem unsafe to proceed: are some properties of the previous step now in danger?

For this reason, in Section 5 we introduce a slight generalization of renaming, called *prenaming*, which can handle extensibility (Theorem 5.17). As a bonus, 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 that would be $z \mapsto y, u \mapsto z$. Prenaming provides finer control of term variance, owing to *relaxed core representation*, which is nothing else than allowing some x/x pairs alongside "real" bindings, as placeholders. Prenamings relate to and are inspired by previous work as follows: A *safe* prenaming is more general than *renaming for a term* from (Lloyd 1987), and it maximizes W in the notion of *W-renaming* from (Eder 1985) (subsection 5.2). Also, prenamings can generalize *substitution renaming* from (Amato and Scozzari 2009).

In Section 7, an application on a nontrivial example is shown (Lemma 7.1), where a propagation claim for logic programming systems has been proved, in a constructive way. As a corollary, a variant lemma (Theorem 7.4) for Prolog is obtained. 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 (Urban et al. 2004), for logic programming systems or their compilation such are still few and far between, a notable exception being (Pusch 1996). New concepts like prenaming may help with this. In subsection 7.1 we discuss the question of renaming-compatibility and resolution-compatibility of unification for logic programming systems. Such properties are important for formalizing operational behaviour of Prolog in a compositional way.

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*, also 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.

2. Substitution

First we need a bit of notation. Assume two disjoint sets: the set of *functor*s, **Fun**, and the countably infinite set of *variable*s, **V**. If $W \subseteq \mathbf{V}$, any mapping F with $F(W) \subseteq \mathbf{V}$ 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 $\mathbf{V} \setminus W$ is finite, W is said to be *co-finite*. A mapping F is *injective on* W, if whenever $F(x) = F(y)$ for $x, y \in W$ 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. Functors of arity 0 are called *constants*. Starting from **V** 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 $x \in \mathbf{V}$ is a *term*. If $t_1, ..., t_n$ are terms and $f/n \in \mathbf{Fun}$, then $f(t_1, ..., t_n)$ 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[1] as $h \cdot t$, 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 $[t_1|[t_2|[...[t_n|[]]]]]$, conveniently written as $[t_1,...,t_n]$.

Let $Vars(t)$ 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 $s \bowtie t$. Otherwise, we say s, t are *variable-disjunct*, written as $s \not\bowtie t$. The list of all variables of t, in order of appearance, shall be denoted as $VarList(t)$.

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 σ of a set of equations E (Definition 6.14, Legacy 6.27) is a *relevant* unifier, if $Vars(\sigma) \subseteq Vars(E)$. A renaming ρ embedding a prenaming α (Algorithm 5.1) is a *relevant* embedding, if $Vars(\rho) \subseteq Vars(\alpha)$.

Additionally, a binary version of relevance, handling two terms, shall also be needed (Algorithm 5.2, Lemma 7.1):

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

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

finite action[2] The set $\{x \in \mathbf{V} \mid \theta(x) \neq x\}$ is finite.

The set $Core(\theta) := \{x \in \mathbf{V} \mid \theta(x) \neq x\}$ shall be called the *active domain*[3] or *core* of θ, and its elements *active variables*[4] of θ. The set $Ran(\theta) := \theta(Core(\theta))$ is called the *active range* of θ. The set $VarsRan(\theta) := Vars(\theta(Core(\theta)))$ is called the *active variable range* of θ. For completeness, a variable x such that $\theta(x) = x$ shall be called a *passive variable*, or a *fixpoint*, for θ. Also, we say that θ is *active* on the variables from $Core(\theta)$, and *passive* on all the other variables.

If $Core(\theta) = \{x_1, ..., x_k\}$, where $x_1, ..., x_k$ are pairwise distinct variables, and θ maps each x_i to t_i, then θ shall have the *core representation* $\{x_1/t_1, ..., x_k/t_k\}$, or the perhaps more visual $\binom{x_1 \ \dots \ x_k}{t_1 \ \dots \ t_k}$. Hence, the above requirement shall also be called *finite core*. Each pair x_i/t_i is called the *binding* for x_i via θ, 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. So the set of variables of a substitution is defined as $Vars(\theta) := Core(\theta) \cup VarsRan(\theta)$.

The notions of restriction and extension of a mapping shall also be transported to core representation: if $\theta \subseteq \sigma$, we say θ is a *restriction* of σ, and σ is an *extension* of θ. The *restriction* $\theta\restriction_W$ of a substitution θ on a set of variables $W \subseteq \mathbf{V}$ is defined as follows: if $x \in W$ then $\theta\restriction_W(x) := \theta(x)$, otherwise $\theta\restriction_W(x) := x$. The restriction of a substitution θ upon the variables of the term t shall be abbreviated as $\theta\restriction_t := \theta\restriction_{Vars(t)}$. We also write $\theta\restriction_{-t}$ to denote the restriction of θ to variables outside of t, like $\theta\restriction_{-t} := \theta\restriction_{Core(\theta) \setminus Vars(t)}$.

Definition of substitution is extended from variables to arbitrary terms in a structure-preserving way by $\theta(f(t_1, ..., t_n)) := f(\theta(t_1), ..., \theta(t_n))$. If s is a term, $\theta(s)$ is an *instance* of s via θ.

The *composition* $\theta \cdot \sigma$ of substitutions θ and σ is defined by $(\theta \cdot \sigma)(t) := \theta(\sigma(t))$. 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 an all-vars substitution ρ is bijective, its inverse shall be denoted as ρ^{-1}. A substitution θ satisfying the equality $\theta \cdot \theta = \theta$ is called *idempotent*.

Example 2.2. $\binom{x\ w\ u\ v}{u\ v\ x\ w} \cdot \binom{u\ v\ x\ y\ z\ w}{x\ w\ y\ u\ v\ z} = \binom{u\ v\ x\ y\ z\ w\ \not\in\ \not\in\ \not\in}{x\ w\ y\ u\ v\ z\ \not\in\ \not\in\ \not\in} = \binom{x\ y\ z\ w}{y\ x\ w\ z}$.

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 Legacy 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 ρ be a renaming and s, t be terms. If $s \notin t$, then $\rho(s) \notin \rho(t)$.*

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

Corollary 3.3 (renaming stability of "=", "∈", "⋈")**.** *Let ρ 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, $s \not\bowtie t$ iff $\rho(s) \not\bowtie \rho(t)$.*

Legacy 3.4 ((Lassez et al. 1988))**.** *A substitution ρ is a renaming iff $\rho(Core(\rho)) = Core(\rho)$.*

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 decomposition of a finite permutation.

[1] In Definition 2.1, we shall overload the dot operator with composing substitutions, in addition to its rôle as pair constructor.

[2] (Gallier 1986) uses the name *finite support*.

[3] Literature traditionally uses the name *domain*. However, in the usual mathematical sense it is always the whole **V** which is the domain of any substitution. It may be less confusing to have both *the domain*, which is uniformly **V**, and *the core* or *active domain*, making it clear that, while every variable can be mapped, only *active variables* are of interest.

[4] The name *active variable* appears in (Jacobs and Langen 1992).

Lemma 3.6 (cycles)**.** *Let σ be an all-vars substitution. It is injective iff for every $x \in \mathbf{V}$, there is $n \in \mathbf{N}$ such that $\sigma^n(x) = x$.*

Proof. Assume σ injective, and choose $x_0 \in \mathbf{V}$. 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, $\sigma^{i-1}(x_0) \in \mathit{Core}(\sigma)$ for every $i \geq 1$. Because of the finiteness of $\mathit{Core}(\sigma)$, 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 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 $\sigma^{m-n}(y) = \sigma^{m-n}(x) = \sigma^{m-n}(\sigma^n(x)) = \sigma^{m-n}(\sigma^n(y)) = \sigma^m(y) = y$, 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 of variables 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 $\mathit{Core}(\sigma) \subseteq \{x_1, ..., x_n\}$, where variables $x_1, ..., x_n$ are pairwise distinct, then $\{x_1, ..., x_n\}$ shall be called a *relaxed core* and $\begin{pmatrix} x_1 & \cdots & x_n \\ \sigma(x_1) & \cdots & \sigma(x_n) \end{pmatrix}$ shall be called a *relaxed core representation* for σ.

If we fix a relaxed core for σ, it shall be denoted $C(\sigma) := \{x_1, ..., x_n\}$. The associated range $\sigma(C(\sigma))$ we denote as $R(\sigma)$. The set of variables of σ is as expected, $V(\sigma) := \mathit{Vars}(C(\sigma)) \cup \mathit{Vars}(R(\sigma))$. To get back to the traditional core representation, we denote by $[\sigma]$ the core representation of σ.

For extending substitution, we shall employ disjoint union.

Definition 4.2 (sum of substitutions)**.** If $\sigma = \begin{pmatrix} x_1 & \cdots & x_m \\ s_1 & \cdots & s_m \end{pmatrix}$ and $\theta = \begin{pmatrix} y_1 & \cdots & y_n \\ t_1 & \cdots & t_n \end{pmatrix}$ are substitutions in relaxed representation such that $\{y_1, ..., y_n\} \not\bowtie \{x_1, ..., x_m\}$, then $\sigma \uplus \theta := \begin{pmatrix} x_1 & \cdots & x_m & y_1 & \cdots & y_n \\ s_1 & \cdots & s_m & t_1 & \cdots & t_n \end{pmatrix}$ is the *sum* of σ and θ.

In case $\{y_1, ..., y_n\} \bowtie \{x_1, ..., x_m\}$ 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 σ is an extension of θ, but simply write $\sigma \supseteq \theta$.

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 $(\alpha \uplus \beta)(t) = \alpha(t)$.*

Proof. For any $x \in \mathit{Vars}(t) \cap C(\alpha)$ by definition $(\alpha \uplus \beta)(x) = \alpha(x)$. Assume now $x \in \mathit{Vars}(t) \cap C(\beta)$. From the condition, $\beta(x) = x$, and by definition of extension, $x \notin C(\alpha)$, hence $(\alpha \uplus \beta)(x) = \beta(x) = x = \alpha(x)$. Clearly, if $x \in \mathit{Vars}(t) \setminus C(\alpha \uplus \beta)$ the claim also holds. ♢

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

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

Corollary 4.5 (backward compatibility)**.** *If σ is complete for t, then for any θ holds $(\sigma \uplus \theta)(t) = \sigma(t)$.*

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*[5]. Like any substitution, a prenaming α 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* α is an all-vars substitution injective on a finite set of variables $C(\alpha) \supseteq \mathit{Core}(\alpha)$.

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 $\alpha = \begin{pmatrix} x_1 & \cdots & x_n \\ y_1 & \cdots & y_n \end{pmatrix}$ and $\beta = \begin{pmatrix} u_1 & \cdots & u_k \\ v_1 & \cdots & v_k \end{pmatrix}$ be prenamings such that $\{u_1, ..., u_k\} \not\bowtie \{x_1, ..., x_n\}$ and $\{v_1, ..., v_k\} \not\bowtie \{y_1, ..., y_n\}$. Then $\alpha \uplus \beta = \begin{pmatrix} x_1 & \cdots & x_n & u_1 & \cdots & u_k \\ y_1 & \cdots & y_n & v_1 & \cdots & v_k \end{pmatrix}$ is also a prenaming.*

Clearly, $C(\alpha \uplus \beta) = C(\alpha) \uplus C(\beta)$ and $R(\alpha \uplus \beta) = R(\alpha) \uplus R(\beta)$.

5.1 The question of inverse

In practice, a prenaming 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 α 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 Scozzari 2009) with the emphasis on the existence[6] of such an extension. In (Eder 1985), the emphasis is on the actual reach[7] 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 α be a prenaming. Then there is a renaming ρ which coincides with α on $\mathbf{V} \setminus (R(\alpha) \setminus C(\alpha))$ such that $\mathit{Vars}(\rho) \subseteq V(\alpha)$.*

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

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

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

Algorithm 5.1: Closure, the natural relevant embedding

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

[6] (Apt 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."

[7] (Eder 1985, p. 35): "Let W be a co-finite set of variables (...) and let σ be a W-renaming. Then there is a permutation π which coincides with σ 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 $\alpha^k(x) \notin C$ and are stuck. For $y := \alpha^k(x)$ 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 α on $C(\alpha)$ we get $\alpha^{m-k}(\overline{\alpha}(\alpha^k(x))) = x$, and hence $\overline{\alpha}^{m+1}(x) = x$.

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

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

Remark 5.5 (relevant embedding is not unique). Let $\alpha = \binom{z\ u\ y\ w_1}{y\ z\ x\ w_2}$, and let us embed it in a relevant renaming. The Algorithm 5.1 gives $\overline{\alpha} = \binom{z\ u\ y\ w_1\ x\ w_2}{y\ z\ x\ w_2\ u\ w_1}$. But $\rho = \binom{z\ u\ y\ w_1\ \ x\ \ w_2}{y\ z\ x\ w_2\ w_1\ \ u}$ is also a relevant renaming which is embedding α. 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* $\alpha := \binom{x_1\ \cdots\ x_n}{y_1\ \cdots\ y_n}$ *and* $\beta := \binom{y_1\ \cdots\ y_n}{x_1\ \cdots\ x_n}$*. Then* $\overline{\beta} = \overline{\alpha}^{-1}$.

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

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

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 $\binom{y}{x}\,(p(x, f(y))) = p(x, f(x))$.

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

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

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

Proof. Let $x, y \in \mathbf{V} \setminus Pit(\alpha)$. 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 $x, y \notin C(\alpha)$, then $\alpha(x) = x \neq y = \alpha(y)$. 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 $y \in Pit(\alpha)$. Contradiction.

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

Definition 5.11 (injectivity domain)**.** For a prenaming α, let $InDom(\alpha) := \mathbf{V} \setminus Pit(\alpha)$. Since $InDom(\alpha)$ is the largest co-finite set containing $C(\alpha)$ on which α is injective, it shall be called the *injectivity domain* of α.

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

Definition 5.12 (safety of prenaming)**.** A prenaming *safe*[8] for a term t is a prenaming α with $Vars(t) \subseteq InDom(\alpha)$.

Clearly, $InDom(\alpha) = C(\alpha) \cup (\mathbf{V} \setminus R(\alpha))$, so α is safe for its relaxed core. Hence, if α is complete for a term, it is safe for that term. For a prenaming α with the quality $R(\alpha) = C(\alpha)$, i. e. a renaming, it is no surprise that $InDom(\alpha) = \mathbf{V}$ 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:

Corollary 5.13 (injectivity domain)**.** *Let* $x \in InDom(\alpha)$*. Then*

$$\alpha(x) = \overline{\alpha}(x).$$

Corollary 5.14 (prenaming stability)**.** *A generalization of Corollary 3.3 holds: Let* s, t *be terms and* α *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,* $s \not\bowtie t$ *iff* $\alpha(s) \not\bowtie \alpha(t)$*.*

Our definition of prenaming was inspired by the following more general notion from (Eder 1985).

Definition 5.15 (W-renaming)**.** Let $W \subseteq V$. A substitution σ is a *W-renaming* if σ is variable-pure on W, and σ is injective on W.

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

What about safety of extension? If α is safe for t, $\alpha \uplus \beta$ does not have to be, even if $\beta(t) = t$, as the following example shows: $\alpha := \binom{v}{w}$, $\beta := \binom{z\ u\ y}{y\ z\ x}$, $t := p(x)$. The following two claims try to redress that isssue.

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

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

Proof. Since $(\mathbf{V} \setminus A) \cup (\mathbf{V} \setminus B) = \mathbf{V} \setminus (A \cap B))$, and $Pit(\alpha) \not\bowtie Pit(\beta)$, we obtain $InDom(\alpha) \cup InDom(\beta) = \mathbf{V}$.

Further, $(\mathbf{V} \setminus A) \cap (\mathbf{V} \setminus B) = \mathbf{V} \setminus (A \cup B)$ and so $Pit(\alpha \uplus \beta) = (R(\alpha) \uplus R(\beta)) \setminus (C(\alpha) \uplus C(\beta)) \subseteq (R(\alpha) \setminus C(\alpha)) \cup (R(\beta) \setminus C(\beta)) = Pit(\alpha) \cup Pit(\beta)$. ♢

In Remark 5.8, $Pit(\alpha') = \{y\}$, $Pit(\binom{y}{x}) = \{x\}$, and $Pit(\alpha) = \{x\}$, hence $InDom(\alpha') = \mathbf{V} \setminus \{y\}$, $InDom(\binom{y}{x}) = \mathbf{V} \setminus \{x\}$ and $InDom(\alpha) = \mathbf{V} \setminus \{x\}$.

By the last claim, staying within $InDom(\alpha)$ and $InDom(\beta)$ ensures staying within $InDom(\alpha \uplus \beta)$. By assuming a bit more about α than just safety, we may ignore the nature of extension β, and still ensure safety and even backward compatibility of $\alpha \uplus \beta$. This shall be used in Section 7.

Theorem 5.17 (extensibility)**.** *Assume* $\alpha \uplus \beta$ *is defined.*

1. *If* α *is safe for* t *and* β *is safe for* t*, then* $\alpha \uplus \beta$ *is safe for* t*.*

[8] Our definition of safe prenaming is more general than the definition of *renaming for a term* in (Lloyd 1987, p. 22), since we do not require $Core(\alpha) \subseteq Vars(t)$.

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

The first part follows from Lemma 5.16 and the second from Corollary 4.5. Observe the importance of *relaxed core* for this to work: otherwise, passive bindings x/x would not be accounted for.

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 $\alpha(t)$ a *variant* of t, and write $\alpha(t) \cong t$. The particular variance and the direction of its application may be explicated: $s =_\alpha t$ iff $s = \alpha(t)$.

If $s \cong t$, then there is a unique α mapping s to t in a complete and relevant[9] 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 $(lfun_=(E), card_=(E))$ decreasing in lexicographic order with each rule application, where $lfun_=(E)$ is the number of function symbols in equations in E, and $card_=(E)$ 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 current set E is $Pren(s, t)$.

elimination $E \uplus \{x = y\} \rightsquigarrow E$, if $x/y \in E$

failure: alias $E \uplus \{x = y\} \rightsquigarrow$ failure, if ($x/z \in E$, $z \neq y$) or ($z/y \in E$, $z \neq x$)

binding $E \uplus \{x = y\} \rightsquigarrow E \cup \{x/y\}$, if ($x/\_ \notin E$) and ($\_/y \notin E$)

failure: instance $E \uplus \{x = t\} \rightsquigarrow$ failure, if $t \notin \mathbf{V}$; $E \uplus \{t = x\} \rightsquigarrow$ failure, if $t \notin \mathbf{V}$

decomposition $E \uplus \{f(s_1, ..., s_n) = f(t_1, ..., t_n)\} \rightsquigarrow E \cup \{s_1 = t_1, ..., s_n = t_n\}$

failure: clash $E \uplus \{f(s_1, ..., s_n) = g(t_1, ..., t_m)\} \rightsquigarrow$ failure, if $f \neq g$ or $m \neq n$

Algorithm 5.2: Computing the prenaming of s to t

Notation 5.19 (epsoid). 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 prenamings, $Pren(t, t)$ and ε are not the same. A prenaming α with relaxed core W mapping each variable on itself (in other words, $C(\alpha) = W$ and $\overline{\alpha} = \varepsilon$) shall be called the *$W$-epsoid* and denoted ε_W. For a term t, we abbreviate $\varepsilon_t := \varepsilon_{Vars(t)}$.

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

[9] for prenaming, we naturally use C for Dom and R for $Range$.

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 $\rho(\sigma) := \{\rho(x)/\rho(\sigma(x)) \mid x \in Core(\sigma)\}$. It is easy to see that $\rho(\sigma)$ is a substitution in core representation. For this we only need two properties of ρ: variable-pure on $Vars(\sigma)$ and injective on $Vars(\sigma)$. These requirements are clearly fulfilled by prenamings 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. $Vars(\sigma) \subseteq InDom(\alpha)$. Then a *variant of σ by α* is

$$\alpha(\sigma) := \{\alpha(x)/\alpha(\sigma(x)) \mid x \in Core(\sigma)\} \tag{1}$$

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 with $\alpha = \binom{y}{x}$, $\sigma = \left(\begin{smallmatrix} x & y \\ a & b \end{smallmatrix}\right)$.

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 $Core(\sigma) = \{x_1, ..., x_n\}$. Due to injectivity of α on $Vars(\sigma)$, if $\alpha(x_i) = \alpha(x_j)$, then $x_i = x_j$, so $i = j$. To finish the proof that (1) a core representation, observe $x \in Core(\sigma)$ iff $x \neq \sigma(x)$ iff $\alpha(x) \neq \alpha(\sigma(x))$, due to injectivity again.

Next, by Corollary 5.14, if $\alpha(\sigma(x_i)) \bowtie \alpha(\sigma(x_j))$, then $\sigma(x_i) \bowtie \sigma(x_j)$, 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 prenamings such that $\alpha(\sigma)$ and $(\alpha \cdot \beta)(\sigma)$ are 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 5.23 ((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, prenamings?

Theorem 5.24 (substitution variant)**.** *Let σ be a substitution and α be a prenaming safe for σ. Then*

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

Proof. First part: According to Definition 5.20, for every $x \in \mathbf{V}$ 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 $\overline{\alpha}(\sigma) \cdot \overline{\alpha} = \overline{\alpha} \cdot \sigma$, hence $\overline{\alpha}(\sigma) = \overline{\alpha} \cdot \sigma \cdot \overline{\alpha}^{-1}$. By Corollary 5.13 holds $\alpha(\sigma) = \overline{\alpha}(\sigma)$, which completes the proof. ♢

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

[10] An immediate consequence of which is $\rho(\sigma) \neq \rho \cdot \sigma$.

Theorem 5.25 (compositionality)**.** *Let σ, θ be substitutions and α be their safe prenaming. Then*

$$\alpha(\sigma \cdot \theta) = \alpha(\sigma) \cdot \alpha(\theta)$$

Proof. Since $Vars(\sigma \cdot \theta) \subseteq Vars(\sigma) \cup Vars(\theta)$, clearly $Vars(\sigma \cdot \theta) \subseteq InDom(\alpha)$. By Theorem 5.24 $\alpha(\sigma) \cdot \alpha(\theta) = \overline{\alpha} \cdot \sigma \cdot \overline{\alpha}^{-1} \cdot \overline{\alpha} \cdot \theta \cdot \overline{\alpha}^{-1} = \overline{\alpha} \cdot \sigma \cdot \theta \cdot \overline{\alpha}^{-1} = \alpha(\sigma \cdot \theta)$. ♢

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 $G \hookrightarrow_{\mathcal{K}_1 : \sigma_1} G_1 \hookrightarrow_{\mathcal{K}_2 : \sigma_2} \ldots \hookrightarrow_{\mathcal{K}_n : \sigma_n} G_n$.

- $\mathcal{K}_i$ is here the *actually used* variant of a program clause (i. e., the current *input clause*) and not the program clause itself.
- The substitution $\sigma_n \cdot \ldots \cdot \sigma_1$ 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 $(\sigma_n \cdot \ldots \cdot \sigma_1)\restriction_G$, whenever $G_n = \square$. 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, $\sigma_n \cdot \ldots \cdot \sigma_1$. A c.a.s. is then a complete answer made relevant by restricting it to query variables.

An input clause $\mathcal{K}_i$ obtained from a program clause $\bar{\mathcal{K}}$ by replacing the variables in order of appearance with $A_1, \ldots, A_n$ may be denoted as $\mathcal{K}_i = \bar{\mathcal{K}}[A_1, \ldots, A_n]$. We also say that $\bar{\mathcal{K}}$ is *applicable* on G_i with *effector clause* $\mathcal{K}_i$, and that $\mathcal{K}_i$ is *effective* on G_i with σ_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 $G \hookrightarrow_{\mathcal{K}_1 : \sigma_1} G_1 \hookrightarrow_{\mathcal{K}_2 : \sigma_2} \ldots \hookrightarrow_{\mathcal{K}_n : \sigma_n} G_n$. We shall define the set of variables of **D** as would be natural for a term, i. e. we regard the annotations $\mathcal{K}_i : \sigma_i$ as part of the derivation. Hence, $Vars(\mathbf{D}) := (Vars(G) \cup \ldots \cup Vars(G_n)) \cup (Vars(\sigma_1) \cup \ldots \cup Vars(\sigma_n)) \cup (Vars(\mathcal{K}_1) \cup \ldots \cup Vars(\mathcal{K}_n))$.

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

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 σ such that $\sigma(g) = s$, then we say *g matches s*, and also that s is an *instance of g* (as already defined in Definition 2.1). The substitution σ is then a *matcher* of g on s.

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

Example: $f(x)$ matches $f(g(x))$, but does not subsume it, while $f(x, y)$ subsumes $f(x, x)$. For relation to Prolog see (Neumerkel 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 $t = \boxed{s}$.

Note that there may be several occurences 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 is t itself, if t 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 *pendant*s, 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 $t = f(t_1, \ldots, t_n)$ and $s \in t_k$, then $AP(s \in t) := f/k \cdot AP(s \in t_k)$.

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 *pendant*s 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 ocurrence $s \in t$. According to list definition, $[f(y), z] = \cdot(f(y), \cdot(z, nil))$. Hence, $AP(s \in t) = (\cdot)/2 \cdot (\cdot)/1 \cdot z$, so the position of $s \in t$ is $2 \cdot 1$ and its ancestry is $(\cdot) \cdot (\cdot)$. An example of pendants: $f(y) \in [f(y), z]$ 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 placeholding facility of relaxed core representation, the following algorithm is linear and rather succinct. In fact, *without* the placeholding facility it would be difficult to capture the error in matching $f(x, x)$ on $f(x, y)$ in just one pass along the terms and without auxiliary registers.

---

variable Let L be a variable. If $L/S \in \delta$ and $S \neq R$, then stop with $FAILURE(\text{"}divergence\text{"})$. Otherwise, $Match(L, R, \delta) := \delta \cup \left(\begin{smallmatrix} L \\ R \end{smallmatrix}\right)$.

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

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

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 ... 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 $Match(L, R, \varepsilon)$ stops with failure, then L does not match R; otherwise, it stops with a substitution δ 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 μ 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 δ. Additionally, δ' is complete for L and $Vars(\delta') \subseteq Vars((L, R, \delta))$.*

Proof. This algorithm clearly always terminates. For the *Proof of 1*, we need two observations, readily verified by structural induction:

- If μ 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 δ.

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

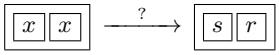

Figure 1. Divergence: one variable, two terms

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 δ_i is complete for s_i as well as relevant. Due to completeness and Corollary 4.5, from $\delta_n \supseteq ... \supseteq \delta_2 \supseteq \delta_1$ follows $\delta_n(s_1) = ... = \delta_2(s_1) = t_1$ and so forth. Hence, $\delta_n(L) = R$. Clearly, $\delta_n \supseteq \delta$ and δ_n is complete for L and relevant. As a final detail, recall that δ_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 σ is *more general* (or *less instantiated*)[11] than a substitution θ, written as $\sigma \leq \theta$,[12] if σ is a right-divisor of θ, i. e. if there exists a substitution δ with the property $\theta = \delta \cdot \sigma$.

How 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 δ 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 σ, θ, denoted $w := VarList((\sigma, \theta))$, 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)**.** *$\sigma \leq \theta$ iff for some w with $Vars(w) = Vars((\sigma, \theta))$ holds that $\sigma(w)$ matches $\theta(w)$.*

[11] It has also been said that σ *schematises* θ (Huet 1976).

[12] Some authors like (Jacobs and Langen 1992) and (Amato and Scozzari 2009) turn the symbol $\leq$ around. Indeed the choice may appear to be abitrary. But we shall stick to the notion that a more general object is "smaller", because it correlates with the "smallness" of the substitution stack.

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

For the other direction, assume there is μ with $\mu(\sigma(w)) = \theta(w)$. By Theorem 6.6, we can choose the matcher μ to be relevant, so $Vars(\mu) \subseteq Vars((\sigma, \theta))$. If for some $x \in \mathbf{V}$ holds $\mu(\sigma(x)) \neq \theta(x)$, then clearly $x \notin Vars((\sigma, \theta))$, hence the inequality becomes $\mu(x) \neq x$, meaning $x \in Core(\mu)$, 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 $Vars(w) \subseteq Vars((\sigma, \theta))$ 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. *$\sigma(w)$ exibits two occurrences of variable x, but at the corresponding positions in $\theta(w)$ there are two mutually distinct terms ("divergence").*

The search for an impasse can be performed by Algorithm 6.1 via $Match(\sigma(w_0), \theta(w_0), \varepsilon)$, where $w_0 := VarList((\sigma, \theta))$.

If no impasse is found, the algorithm produces δ 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: $\sigma := \binom{x}{y}$ is more general than $\binom{x\ y}{a\ a}$, but not more general than $\theta := \binom{x}{a}$. The former claim is justified by $\binom{x\ y}{a\ a} = \binom{y}{a} \cdot \binom{x}{y}$. 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 Hai-Ping Ko (reported in (Shepherdson 1994)), which was pivotal in showing that the strong completeness theorem for SLD-derivation from (Lloyd 1987) needed a revision. The Ko example purports that $\sigma := \binom{x}{f(y,z)}$ is not more general than $\theta := \binom{x}{f(a,a)}$, where y, z are distinct variables. For proof, it was observed: if $\delta \cdot \binom{x}{f(y,z)} = \binom{x}{f(a,a)}$, 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 $\delta \cdot \binom{x}{f(y,z)}$. To aleviate the need for such "ad-hoc", custom-made proofs, Algorithm 6.1 could be used, giving divergence (Figure 3).

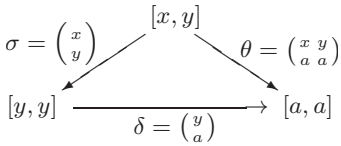

Figure 2. Successfull check on $\leq$

6.3 Equivalence

The set of substitutions is not partially ordered by $\leq$, namely it is possible that $\sigma \leq \theta$ and $\theta \leq \sigma$ for $\sigma \neq \theta$. Such cases form an equivalence relation, called simply *equivalence*[13] and denoted by $\sigma \sim \theta$.

The following property, in similar form, has been proven in (Eder 1985); the formulation is from (Apt 1997). The property follows from Theorem 6.8 as well, since the case where one of a pendant pair is a variable and the other a non-variable is clearly not possible (shrinkage failure).

[13] perhaps a new name like *equigeneral* would be less confusing, in view of counter-intuitive equivalences?

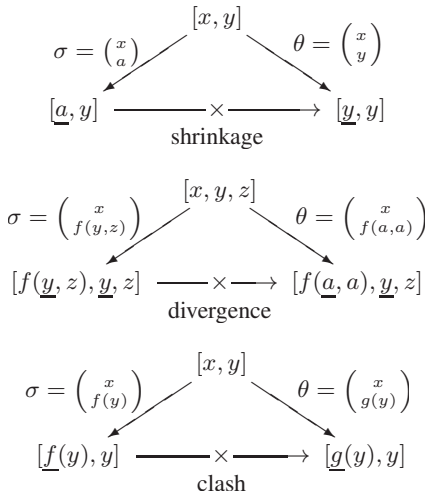

Figure 3. Failed check on $\leq$

Legacy 6.11 (equivalence)**.** *θ is more general than θ' and θ' is more general than θ iff for some renaming ρ such that $Vars(\rho) \subseteq Vars(\theta) \cup Vars(\theta')$ holds $\rho \cdot \theta = \theta'$.*

With some practice, such a renaming ρ can be guessed, or simply constructed by Algorithm 6.1.

Example 6.12. Since $\left(\begin{smallmatrix} y & x \\ x & y \end{smallmatrix}\right)$ is a renaming and $\left(\begin{smallmatrix} y & x \\ x & y \end{smallmatrix}\right) \cdot \left(\begin{smallmatrix} y & x \\ x & y \end{smallmatrix}\right) = \varepsilon$, we have $\left(\begin{smallmatrix} y & x \\ x & y \end{smallmatrix}\right) \sim \varepsilon$. 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 $\sigma := \left(\begin{smallmatrix} y \\ x \end{smallmatrix}\right)$, $\sigma' := \left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right)$ and $\theta := \left(\begin{smallmatrix} x \\ z \end{smallmatrix}\right)$. Then $\sigma \sim \sigma'$, but $\theta \cdot \sigma = \left(\begin{smallmatrix} y & x \\ z & z \end{smallmatrix}\right) \not\sim \theta \cdot \sigma' = \left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right)$. 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 θ such that $\theta(s) = \theta(t)$, then s and t are said to be *unifiable*, and θ is their *unifier*, the set of all such being $Unif(s,t)$. It is a *relevant*, if $Vars(\theta) \subseteq Vars(s) \cup Vars(t)$. A unifier θ 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 $Mgus(s,t) := \{\theta \in Unif(s,t) \mid \text{for every } \alpha \in Unif(s,t) \text{ holds } \theta \leq \alpha\}$.

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 $s{=}t$ to a set of equations E by defining $Unif(E) := \{\theta \mid \text{ for every } (s{=}t) \in E \text{ holds } \theta(s) = \theta(t)\}$. Similarly for $Mgus(E)$. 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 $\sigma \in Mgus(s,t)$, then for any renaming ρ by Corollary 3.3 $\rho \cdot \sigma \in Mgus(s,t)$. In fact, any element from $Mgus(s,t)$ has this form, as a consequence of Legacy 6.11:

Legacy 6.15 (equivalence of mgus)**.** *Let $\mu \in Mgus(E)$. Then $\mu' \in Mgus(E)$ iff there is a renaming ρ such that $Vars(\rho) \subseteq Vars(\mu) \cup Vars(\mu')$ and $\mu' = \rho \cdot \mu$.*

Thus, the set $Mgus(s,t)$ is either empty or infinite.[14] As a meta-function, $Mgus$ has two pleasing properties: it is compatible with renaming and it is compatible with LD-resolution.

Lemma 6.16 (renaming compatibility of $Mgus$)**.** *For every ρ and E holds $Mgus(\rho(E)) = \rho(Mgus(E))$.*

Proof. This follows from Theorem 5.24 and Corollary 3.3. Assume $\sigma \in Mgus(s,t)$, then $\rho(\sigma)(\rho(s)) = \rho(\sigma(s)) = \rho(\sigma(t)) = \rho(\sigma)(\rho(t))$. Further, if θ is a unifier of $\rho(s), \rho(t)$, then $\theta \cdot \rho$ is a unifier of s, t, hence there is a renaming δ with $\theta \cdot \rho = \delta \cdot \sigma$, giving $\theta = \delta \cdot \sigma \cdot \rho^{-1} = \delta \cdot \rho^{-1} \cdot \rho \cdot \sigma \cdot \rho^{-1} = (\delta \cdot \rho^{-1}) \cdot \rho(\sigma)$, meaning $\rho(\sigma) \in Mgus(\rho(E))$. For the other direction, observe $\theta = \rho \cdot \rho^{-1} \cdot \delta \cdot \sigma \cdot \rho^{-1} = \rho(\rho^{-1} \cdot \delta \cdot \sigma)$. ♢

Compatibility of $Mgus$ with LD-resolution, also called *iteration property*, is proved in (Apt 1997).

Legacy 6.17 (iteration for $Mgus$)**.** *1. Let E_1, E_2 be sets of equations. If σ is a mgu of E_1 and θ is a mgu of $\sigma(E_2)$, then $\theta \cdot \sigma$ is a mgu of $E_1 \cup E_2$.*

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

6.4.1 Unification by algorithm

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

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 $\mathfrak{A}_{\mathtt{m2}}$, but it can be made deterministic using sequences instead of sets and picking the leftmost equation eligible for a rule application, as observed in (Apt 1997, p. 36). The resulting algorithm shall be denoted $\mathfrak{A}_{\overline{\mathtt{m2}}}$.

6.4.2 ... and iteration property

As opposed to $Mgus$, any particular unification algorithm like $\mathfrak{A}_{\mathtt{m2}}$ does not have to satisfy the iteration property. But the deterministic version $\mathfrak{A}_{\overline{\mathtt{m2}}}$ does.

Example 6.18 (no iteration for $\mathfrak{A}_{\mathtt{m2}}$)**.** Let $E_1 := (x{=}y, y{=}x)$ and $E_2 := (z{=}f(x))$. If we pick the equation $y{=}x$ for binding (denoted by underlining), we obtain $E_1 = \{x{=}y, \underline{y{=}x}\} \rightsquigarrow \{x{=}x, y{=}x\} \rightsquigarrow \{y{=}x\} \rightsquigarrow \left(\begin{smallmatrix} y \\ x \end{smallmatrix}\right) =: \sigma$ and $\sigma(E_2) \rightsquigarrow \left(\begin{smallmatrix} z \\ f(x) \end{smallmatrix}\right) =: \theta$. However, for unifying $E_1 \cup E_2$ we may as well pick $x{=}y$ for binding, and get $E_1 \cup E_2 = \{\underline{x{=}y}, y{=}x, z{=}f(x)\} \rightsquigarrow \{x{=}y, y{=}y, z{=}f(y)\} \rightsquigarrow \left(\begin{smallmatrix} x & z \\ y & f(y) \end{smallmatrix}\right) \neq \theta \cdot \sigma$.

Lemma 6.19 (iteration for $\mathfrak{A}_{\overline{\mathtt{m2}}}$)**.** *Assume $\sigma = \mathfrak{A}_{\overline{\mathtt{m2}}}(E')$ and $\theta = \mathfrak{A}_{\overline{\mathtt{m2}}}(\sigma(E''))$. Then $\mathfrak{A}_{\overline{\mathtt{m2}}}((E', E'')) = \theta \cdot \sigma$.*

Proof. By Legacy 6.17, we know that E', E'' is unifiable.

The deterministic version of $\mathfrak{A}_{\mathtt{m2}}$ transforms an equation sequence from left to right. This has the nice consequence that $\mathfrak{A}_{\overline{\mathtt{m2}}}$ chooses the same equations to transform in E' as in E', E'' for so

[14] Because of the equivalence, the sloppy formulation "*the* most general unifier of s and t" is often used.

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.

$\mathfrak{A}_{\overline{\mathtt{mz}}}((E', E'')) = \mathfrak{A}_{\overline{\mathtt{mz}}}((\underline{x_1{=}t_1}, E'_1, E''))$
$= \mathfrak{A}_{\overline{\mathtt{mz}}}((\,x_1{=}t_1\,, \sigma_1(E'_1), \sigma_1(E''))$
$= \mathfrak{A}_{\overline{\mathtt{mz}}}((\,x_1{=}t_1\,, \underline{x_2{=}t_2}, E'_2, \sigma_1(E'')))$
$= \mathfrak{A}_{\overline{\mathtt{mz}}}((\,\sigma_2(x_1{=}t_1)\,, \,x_2{=}t_2\,, \sigma_2(E'_2), \sigma_2(\sigma_1(E''))))$
$= ... = \mathfrak{A}_{\overline{\mathtt{mz}}}((\,(\sigma_k \cdot ... \cdot \sigma_2)(x_1{=}t_1)\,, \,...,\, \sigma_k(x_{k-1}{=}t_{k-1})\,, \underline{x_k{=}t_k},$
$(\sigma_k \cdot ... \cdot \sigma_1)(E'')))$ $= \mathfrak{A}_{\overline{\mathtt{mz}}}((\,x_1{=}(\sigma_k \cdot ... \cdot \sigma_2)(t_1)\,, \,...,$
$x_{k-1}{=}\sigma_k(t_{k-1})\,, \underline{x_k{=}t_k}, (\sigma_k \cdot ... \cdot \sigma_1)(E'')))$

where $\sigma_1 = \binom{x_1}{t_1}, ..., \sigma_k = \binom{x_k}{t_k}$. Here we made use of $x_1 \notin \sigma_1(E'_1)$ meaning $x_1 \not\bowtie \sigma_2$, and $x_1, x_2 \notin \sigma_2(E'_2)$ meaning $x_1, x_2 \not\bowtie \sigma_3$, etc., which are due to definition of binding steps. Putting $E'' = \square$, we obtain solved form, hence

$$\sigma = \mathfrak{A}_{\overline{\mathtt{mz}}}(E') = \begin{pmatrix} x_1 & ... & x_{k-1} & x_k \\ (\sigma_k \cdot ... \cdot \sigma_2)(t_1) & ... & \sigma_k(t_{k-1}) & t_k \end{pmatrix}$$
$$= \binom{x_k}{t_k} \cdot \binom{x_{k-1}}{t_{k-1}} \cdot ... \cdot \binom{x_1}{t_1} = \sigma_k \cdot ... \cdot \sigma_1$$

Consider now an arbitrary E''. From the definition of binding steps follows that $x_1, ..., x_k \not\bowtie (\sigma_k \cdot ... \cdot \sigma_1)(E'') = \sigma(E'')$, which shows that the pairs from the solved form for E' remain passive during the handling of $\sigma(E'')$. Thus, $\mathfrak{A}_{\overline{\mathtt{mz}}}((E', E'')) = \theta_m \cdot ... \cdot \theta_1 \cdot \sigma_k \cdot ... \cdot \sigma_1$, where $\theta_1, ..., \theta_m$ stem from binding steps for $\sigma(E'')$, and $\theta = \mathfrak{A}_{\overline{\mathtt{mz}}}(\sigma(E'')) = \theta_m \cdot ... \cdot \theta_1$. Overall, $\mathfrak{A}_{\overline{\mathtt{mz}}}((E', E'')) = \theta \cdot \sigma$, as hoped for. ♢

6.4.3 ... and renaming-compatibility

For freely choosable mgus, renaming-compatibility holds, as seen in Lemma 6.16. But what if mgus are chosen by an algorithm? For example, the simplest unification problem $p(x) = p(y)$ has among others two equally attractive candidate mgus, $\{x/y\}$ and $\{y/x\}$. Assume our unification algorithm decided upon $\{x/y\}$. Assume further that we rename the protagonists and obtain the unification problem $p(x) = p(z)$. What mgu shall be chosen this time? To ensure some dependability in this issue, we shall place on any unification algorithm the following simple requirement.

Axiom 6.20 (renaming compatibility of $\mathfrak{A}$)**.** Let $\mathfrak{A}$ be a unification algorithm. For any renaming ρ and any equation E, it has to hold $\mathfrak{A}(\rho(E)) = \rho(\mathfrak{A}(E))$.

Since classical unification algorithms like Robinson's and Martelli-Montanari's do not depend upon the actual names of variables[15], this requirement is in praxis always satisfied.

Deterministic character of unification algorithms allows for some predictability when applying a program clause on a query in different ways, i. e. using different variants. This is expressed in the first part of our next claim. The second part makes a connection between *any* resolution for a query, and the one made with algorithm $\mathfrak{A}$.

Lemma 6.21 (two input clauses)**.** *Assume a unification algorithm $\mathfrak{A}$ satisfying Axiom 6.20. Let G be a query and $\mathcal{K}$ and $\mathcal{L}$ be two variants of the same program clause for G, each variable-disjunct with G. Let $\lambda := Pren(\mathcal{K}, \mathcal{L})$. Then*

1. *there is $\mathfrak{A}(G{=}Head(\mathcal{L})) = \theta$ iff there is $\mathfrak{A}(G{=}Head(\mathcal{K})) = \overline{\lambda}^{-1}(\theta)$*
2. *$\mathcal{K}$ is effective on G with some mgu σ iff there is renaming ρ with $\theta = \rho \cdot \overline{\lambda}(\sigma)$*

Additionally, if σ is relevant, then $\overline{\lambda}(\sigma) = \lambda(\sigma)$. If furthermore $\mathfrak{A}$ produces relevant mgus, then $Vars(\rho) \subseteq Vars(G) \cup Vars(\mathcal{L})$.

Proof. For the first part, $\overline{\lambda}(\mathfrak{A}(G{=}Head(\mathcal{K}))) = \mathfrak{A}(\overline{\lambda}(G{=}Head(\mathcal{K})))$ $= \mathfrak{A}(\overline{\lambda}(G){=}\overline{\lambda}(Head(\mathcal{K})))) = \mathfrak{A}(G{=}Head(\mathcal{L}))$, which is due to Axiom 6.20, $\overline{\lambda}(G) = G$ and $\overline{\lambda}(\mathcal{K}) = \mathcal{L}$. Hence, if $\theta := \mathfrak{A}(G{=}Head(\mathcal{L}))$, then $\mathfrak{A}(G{=}Head(\mathcal{K})) = \overline{\lambda}^{-1}(\theta)$.

For the second part, note that σ and $\mathfrak{A}(G{=}Head(\mathcal{K}))$ are two mgus for the same unification task. Hence, by Legacy 6.15, there is renaming δ with

$$Vars(\delta) \subseteq Vars(\sigma) \cup Vars(\mathfrak{A}(G{=}Head(\mathcal{K}))) \qquad (2)$$

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

It remains to consider relevance. If σ is relevant, $Vars(\sigma) \subseteq Vars(\mathcal{K}) \cup Vars(G)$. On the other hand, $Pit(\lambda) = Vars(\mathcal{L}) \setminus Vars(\mathcal{K})$ and $G \not\bowtie \mathcal{L}$, hence $Pit(\lambda) \not\bowtie \sigma$, meaning that $\lambda(\sigma)$ is defined and $\overline{\lambda}(\sigma) = \lambda(\sigma)$. Lastly, if both σ and $\mathfrak{A}(G{=}Head(\mathcal{K}))$ are relevant, from (2) follows $Vars(\rho) \subseteq Vars(G) \cup Vars(\mathcal{L})$. ♢

Notation 6.22 (equivalent modulo prenaming). If for some prenaming λ holds $\theta \sim \lambda(\sigma)$, then we also write $\theta \sim_\lambda \sigma$.

With this notation, in the previous claim we would have had $\theta \sim_{\overline{\lambda}} \sigma$. Assuming relevant mgus, this can be further simplified to $\theta \sim_\lambda \sigma$.

Remark 6.23. The relationship from Lemma 6.21 is symmetrical. Let $\mu := Pren(\mathcal{L}, \mathcal{K})$, then $\overline{\mu} = \overline{\lambda}^{-1}$ (Lemma 5.6). From $\theta = \rho \cdot \overline{\lambda}(\sigma)$ follows $\overline{\lambda}^{-1}(\rho^{-1} \cdot \theta) = \sigma$. By assigning $\delta := \overline{\lambda}^{-1}(\rho^{-1})$ we obtain $\sigma = \delta \cdot \overline{\mu}(\theta)$, or $\sigma \sim_{\overline{\mu}} \theta$.

If θ is relevant, $Vars(\theta) \in Vars(G, \mathcal{L})$, so due to $G \not\bowtie \mathcal{K}$ and $Pit(\mu) = Vars(\mathcal{K}) \setminus Vars(\mathcal{L})$ we have $\theta \not\bowtie Pit(\mu)$, thus $\overline{\mu}(\theta) = \mu(\theta)$, and $\sigma \sim_\mu \theta$.

6.5 Idempotence

Recall that a substitution θ 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 (Eder 1985), composition of two idempotent substitutions does not have to be idempotent – not even equivalent to an idempotent substitution. Example: $\sigma := \binom{x}{f(y)}$ and $\theta := \binom{y}{f(z)}$ give $\sigma \cdot \theta = \begin{pmatrix} x & y \\ f(y) & f(z) \end{pmatrix}$.

However, $\theta \cdot \sigma = \begin{pmatrix} x & y \\ f(f(z)) & f(z) \end{pmatrix}$ is idempotent. This is an instance of a useful property from (Apt 1997).

Legacy 6.25 (two idempotence criteria)**.** *Let σ, θ be substitutions.*

1. *θ is idempotent iff $Core(\theta) \cap VarsRan(\theta) = \emptyset$.*
2. *Let σ and θ be idempotent. If $VarsRan(\theta) \not\bowtie Core(\sigma)$, then $\theta \cdot \sigma$ is also idempotent.*

The first criterion is quite intuitive: if the active domain and the active range of a substitution σ have no variables in common, then all the variables from the active domain shall be *released* from the term t after the application of σ upon t. Therefore, a repeated application of σ cannot change anything.

[15] as remarked in (Amato and Scozzari 2009)

By the unruly charm of substitutions, $\binom{x\ y}{y\ x}$ 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 σ is idempotent and $\sigma \in Mgus(t,t)$ for some t, then $\sigma = \varepsilon$.*

Proof. Since we know that $\varepsilon \in Mgus(t,t)$, by Legacy 6.15 there has to hold $\varepsilon \sim \sigma$, so there is a renaming ρ such that $\rho = \rho \cdot \varepsilon = \sigma$. If σ is idempotent, by Legacy 6.25 we know that $Core(\sigma) \cap VarsRan(\sigma) = \emptyset$. The only renaming with this property is ε, since for a renaming always holds $Core = Ran$ (Legacy 3.4). ♢

It turned out that relevance is a mandatory property of an idempotent mgu (Apt 1997). 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 $\sigma \in Mgus(E)$ is idempotent, but not relevant, i. e. there is $z \in Vars(\sigma)$ such that $z \notin Vars(E)$. Let us show that σ cannot be an mgu, by finding a unifier θ of E such that $\sigma \not\leq \theta$. Technically, we construct θ and a witness term w such that for no δ can hold $\delta(\sigma(w)) = \theta(w)$, as outlined in Corollary 6.9.

Case $z \in Core(\sigma)$: Here we choose $\theta := \sigma\restriction_{-z}$. If σ is an idempotent unifier of E, then so is θ (Legacy 6.25).

Subcase 1: $\sigma(z) = g$ is ground. Since $\theta(z) = z$, the witness can be $w := z$ (shrinkage). Subcase 2: $\sigma(z)$ contains a variable, say x, pictured as $\sigma(z) = \boxed{x}$. Due to idempotency of σ, holds $x \notin Core(\sigma)$, so $x \neq z$. We get $\sigma([x,z]) = [x, \boxed{x}]$, whereas $\theta([x,z]) = [x,z] = [x, \boxed{\neq}]$. So with $w := [x,z]$ we have divergence (if $\sigma(z) = x$) or shrinkage (otherwise).

Case $z \notin Core(\sigma)$, but $z \in Ran(\sigma)$: There is $x \in Core(\sigma)$ (and therefore $x \neq z$) such that $\sigma(x) = \boxed{z}$. Here we take θ to be an idempotent and relevant mgu of E (e. g. the outcome of a classical unification algorithm). Due to relevance, $z \notin \theta(x)$. We get $\sigma([z,x]) = [z, \boxed{z}]$, whereas $\theta([z,x]) = [z, \boxed{\neq}]$ (divergence). ♢

6.6 Restriction

Remark 6.28 (restriction is not compositional). In general, $(\sigma \cdot \theta)\restriction_W = \sigma\restriction_W \cdot \theta\restriction_W$ does not hold. Take $\theta := \binom{x}{y}$, $\sigma := \binom{y}{a}$, $W := \{x\}$. Then $\sigma \cdot \theta = \binom{y}{a} \cdot \binom{x}{y} = \binom{x\ y}{a\ a}$, $(\sigma \cdot \theta)\restriction_W = \binom{x}{a}$, but on the other hand, $\sigma\restriction_W = \varepsilon$, $\theta\restriction_W = \theta$, $\sigma\restriction_W \cdot \theta\restriction_W = \binom{x}{y}$.

On the plus side, restriction is renaming-compatible: $\rho(\sigma\restriction_W) = \rho(\sigma)\restriction_{\rho(W)}$. 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 LD-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 mgus, 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 $\mathbf{D}$ assigned the input clause $\mathcal{K} :=$ "$p(U, V) \leftarrow q(U, W)$.". With renaming $\rho = \binom{U\ W}{W\ U}$, and assuming the algorithm is renaming-compatible, the query $\rho(G) = G$ at the tip of the derivation $\rho(\mathbf{D}) = \mathbf{D}$ would need to be assigned the input clause $\rho(\mathcal{K}) =$ "$p(W, V) \leftarrow q(W, U)$.". But already something else has been assigned to it.

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

7.1.2 Resolution-compatibility ("iteration property")

As seen in subsubsection 6.4.1, $\mathfrak{A}_{\mathfrak{m}2}$ does not but $\mathfrak{A}_{\overline{\mathfrak{m}2}}$ 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 mgus, that is to say for all of them,[16] a propagation result can be proved, which leads to a constructive and incremental version of the variant lemma.

Assume the program "$son(S) \leftarrow male(S)$, $child(S, P)$.", and let us enquire about son in two derivations (Table 1). If we know that one query, say $son(X)$, is a variant of the other, $son(A)$, 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 variable*s, as opposed to *query 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 (Table 1), at least the prenaming can be extended to accomodate those local variables. The claim is proved in a constructive manner.

query	input clause	resolvent
$son(X)$	$son(B) \leftarrow male(B), child(B, A).$	$male(X), child(X, A)$
$son(A)$	$son(X) \leftarrow male(X), child(X, B).$	$male(A), child(A, B)$

Table 1. Resolution may produce local variables

[16] Classical unification algorithms not only satisfy Axiom 6.20 but also yield idempotent mgus. Idempotent mgus are always relevant (Legacy 6.27).

Lemma 7.1 (propagation of variance)**.** *Assume a unification algorithm $\mathfrak{A}$ satisfying Axiom 6.20. Assume an SLD-derivation $\boldsymbol{D}$ ending with G and an SLD-derivation $\boldsymbol{D}'$ ending with G' such that $\alpha(G) = G'$ for some prenaming α which is complete for G and relevant for $\boldsymbol{D}$ to $\boldsymbol{D}'$.*

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

1. *$\alpha \uplus \lambda$ is complete for H*
2. *$\alpha \uplus \lambda$ is relevant for $\boldsymbol{D} \hookrightarrow_{\mathcal{K}\,:\,\sigma} H$ to $\boldsymbol{D}' \hookrightarrow_{\mathcal{K}'\,:\,\sigma'} H'$*
3. *$\sigma' = (\alpha \uplus \lambda)(\sigma)$ and $H' = (\alpha \uplus \lambda)(H)$*

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

$$\begin{array}{ccc} G & \xrightarrow[\hookrightarrow]{} & H \\ {\scriptstyle\alpha}\big\downarrow{\scriptstyle\alpha \uplus \lambda} & & \big\downarrow{\scriptstyle\alpha \uplus \lambda} \\ G' & \xrightarrow[\hookrightarrow]{} & H' \end{array}$$

Figure 4. Propagation of variance

$$\begin{array}{ccc} son(X) & \xrightarrow[\hookrightarrow]{} & male(X),\ child(X, A) \\ \big\downarrow{\scriptstyle\alpha=\left(\begin{smallmatrix} X & A \\ A & C \end{smallmatrix}\right)} & & \big\downarrow{\scriptstyle\alpha \uplus ?} \\ son(A) & \xrightarrow[\hookrightarrow]{} & male(A),\ child(A, B) \end{array}$$

Figure 5. ...is not always possible

Proof. First let us establish that $\alpha \uplus \lambda$ is defined. Due to *relevance of α for $\boldsymbol{D}, \boldsymbol{D}'$*,

$$C(\alpha) \subseteq Vars(\mathbf{D}) \text{ and } R(\alpha) \subseteq Vars(\mathbf{D}') \tag{3}$$

Due to *standardization-apart*, $\mathcal{K} \not\bowtie \mathbf{D}$ and $\mathcal{K}' \not\bowtie \mathbf{D}'$, hence

$$C(\lambda) \not\bowtie \mathbf{D} \text{ and } R(\lambda) \not\bowtie \mathbf{D}' \tag{4}$$

Thus $C(\alpha) \not\bowtie C(\lambda)$ and $R(\alpha) \not\bowtie R(\lambda)$, so $\alpha \uplus \lambda$ is defined. Also, (4) proves that λ is passive on old variables, i. e. $\lambda(\mathbf{D}) = \mathbf{D}$.

Let $G := (\mathbf{M}, A, \mathbf{N})$ and $H := \sigma(\mathbf{M}, B, \mathbf{N})$, where $\mathbf{M}, \mathbf{N}$ are conjunctions. Then $G' = (\mathbf{M}', A', \mathbf{N}') = (\alpha(\mathbf{M}), \alpha(A), \alpha(\mathbf{N}))$.

Let $\mathcal{K}\colon\ A_1 \leftarrow B_1$ and $\mathcal{K}'\colon\ A_2 \leftarrow B_2$. Then $\sigma = \mathfrak{A}(A, A_1)$, $B = \sigma(B_1)$ and $\sigma' = \mathfrak{A}(A', A_2)$, $B' = \sigma'(B_2)$. Also,

$$Vars(\mathbf{M}, A, \mathbf{N}) \subseteq C(\alpha),\ \text{by } \textit{completeness} \text{ of } \alpha \text{ for } G \tag{5}$$

$$Vars(\mathcal{K}) = Vars(A_1, B_1),\ \text{by definition of } \lambda \tag{6}$$

$$Vars(\sigma) \subseteq Vars(A) \cup Vars(A_1),\ \text{by } \textit{relevance} \text{ of } \sigma \tag{7}$$

Having thus fielded all the assumptions, we obtain

$$Vars(\mathbf{M}, A, \mathbf{N}) \subseteq InDom(\alpha \uplus \lambda),\ \text{by (5) and Theorem 5.17} \tag{8}$$

$$Vars(A_1, B_1) \subseteq InDom(\alpha \uplus \lambda),\ \text{by (6) and Theorem 5.17} \tag{9}$$

$$Vars(\sigma) \subseteq InDom(\alpha \uplus \lambda),\ \text{by (7), (8) and (9)} \tag{10}$$

$$\overline{(\alpha \uplus \lambda)}(\sigma) = (\alpha \uplus \lambda)(\sigma),\ \text{by (10)} \tag{11}$$

Proof of 3:

$$\begin{aligned}
\sigma' &= \mathfrak{A}(A', A_2) = \mathfrak{A}(\alpha(A), \lambda(A_1)) \\
&= \mathfrak{A}((\alpha \uplus \lambda)(A), (\alpha \uplus \lambda)(A_1)),\ \text{by (5), (6) and Theorem 5.17} \\
&= \mathfrak{A}((\overline{\alpha \uplus \lambda})(A), (\overline{\alpha \uplus \lambda})(A_1)),\ \text{by (8) and (9)} \\
&= \overline{(\alpha \uplus \lambda)}(\mathfrak{A}(A, A_1)),\ \text{by Axiom 6.20} \\
&= \overline{(\alpha \uplus \lambda)}(\sigma) = (\alpha \uplus \lambda)(\sigma),\ \text{by (11)}
\end{aligned}$$

$$\begin{aligned}
B' &= \sigma'(B_2) = (\alpha \uplus \lambda)(\sigma)(\lambda(B_1)) \\
&= (\alpha \uplus \lambda)(\sigma)((\alpha \uplus \lambda)(B_1)),\ \text{by (6) and Theorem 5.17} \\
&= (\alpha \uplus \lambda)(\sigma(B_1)),\ \text{by Theorem 5.24} \\
&= (\alpha \uplus \lambda)(B)
\end{aligned}$$

$$\begin{aligned}
H' &= \sigma'(\alpha(\mathbf{M}), B', \alpha(\mathbf{N})) \\
&= (\alpha \uplus \lambda)(\sigma)(\alpha(\mathbf{M}), (\alpha \uplus \lambda)(B), \alpha(\mathbf{N})) \\
&= (\alpha \uplus \lambda)(\sigma)((\alpha \uplus \lambda)(\mathbf{M}), (\alpha \uplus \lambda)(B), (\alpha \uplus \lambda)(\mathbf{N})),\ \text{by (5)} \\
&= (\alpha \uplus \lambda) \cdot \sigma(\mathbf{M}, B, \mathbf{N}),\ \text{by Theorem 5.24} \\
&= (\alpha \uplus \lambda)(H)
\end{aligned}$$

Proof of 2: By definition, $C(\lambda) = Vars(\mathcal{K})$ and $R(\lambda) = Vars(\mathcal{K}')$. Hence, and due to relevance of α, $C(\alpha \uplus \lambda) = C(\alpha) \uplus C(\lambda) \subseteq Vars(\mathbf{D}) \cup Vars(\mathcal{K}) \subseteq Vars(\mathbf{D} \hookrightarrow_{\mathcal{K}\,:\,\sigma} H)$. Similarly, $R(\alpha \uplus \lambda) \subseteq Vars(\mathbf{D}' \hookrightarrow_{\mathcal{K}'\,:\,\sigma'} H')$, therefore $\alpha \uplus \lambda$ is relevant for $\mathbf{D} \hookrightarrow_{\mathcal{K}\,:\,\sigma} H$ to $\mathbf{D}' \hookrightarrow_{\mathcal{K}'\,:\,\sigma'} H'$.

Proof of 1: By (5) and (6), $Vars(H) \subseteq Vars(G) \cup Vars(\mathcal{K}) \subseteq C(\alpha) \cup C(\lambda) = C(\alpha \uplus \lambda)$, meaning $\alpha \uplus \lambda$ is complete for H. $\diamondsuit$

Definition 7.2 (similarity)**.** SLD-derivations of the same length

$$\begin{array}{l} G \hookrightarrow_{\mathcal{K}_1\,:\,\sigma_1} G_1 \hookrightarrow_{\mathcal{K}_2\,:\,\sigma_2} \ldots \hookrightarrow_{\mathcal{K}_n\,:\,\sigma_n} G_n \\ G' \hookrightarrow_{\mathcal{K}'_1\,:\,\sigma'_1} G'_1 \hookrightarrow_{\mathcal{K}'_2\,:\,\sigma'_2} \ldots \hookrightarrow_{\mathcal{K}'_n\,:\,\sigma'_n} G'_n \end{array} \tag{12}$$

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 $\mathcal{K}_i$ and $\mathcal{K}'_i$ 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 Shepherdson 1991), (Apt 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 mgus are not independent. Second, the precise variance is now known.

Theorem 7.4 (variant claim for logic programming systems)**.** *Assume a unification algorithm $\mathfrak{A}$ satisfying Axiom 6.20 and yielding relevant mgus. 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 $i = 1, \ldots, n$ holds $G'_i = \beta_i(G_i)$, $\sigma'_i = \beta_i(\sigma_i)$ and $\sigma'_i \cdot \ldots \cdot \sigma'_1 = \beta_i(\sigma_i \cdot \ldots \cdot \sigma_1)$, where $\beta_i := \alpha \uplus \lambda_1 \uplus \ldots \uplus \lambda_i$, $\alpha := Pren(G, G')$ and $\lambda_i := Pren(\mathcal{K}_i, \mathcal{K}'_i)$.*

Proof. By assumption, G and G' are variants, so

$$\alpha := Pren(G, G') \tag{13}$$

exists. Clearly, α is complete for G, since $Vars(G) = C(\alpha)$. By construction, α is also relevant for $\mathbf{D}_0 := G$ to $\mathbf{D}'_0 := G'$.

We may iterate Lemma 7.1, obtaining for every $i = 1, ..., n$

$$\sigma'_i = (\alpha \uplus \lambda_1 \uplus ... \uplus \lambda_i)(\sigma_i) \tag{14}$$

$$G'_i = (\alpha \uplus \lambda_1 \uplus ... \uplus \lambda_i)(G_i) \tag{15}$$

where $\lambda_i := Pren(\mathcal{K}_i, \mathcal{K}'_i)$. Therefore, $\sigma'_i \cdot \sigma'_{i-1} \cdot ... \cdot \sigma'_1 = (\alpha \uplus \lambda_1 \uplus ... \uplus \lambda_i)(\sigma_i) \cdot (\alpha \uplus \lambda_1 \uplus ... \uplus \lambda_{i-1})(\sigma_{i-1}) \cdot ... \cdot (\alpha \uplus \lambda_1)(\sigma_1)$.

Let $k < i$. Since $Vars(\sigma_k) \subseteq Vars(G) \cup Vars(\mathcal{K}_1) \cup ... \cup Vars(\mathcal{K}_k) \subseteq C(\alpha) \cup C(\lambda_1) \cup ... \cup C(\lambda_k) = C(\alpha \uplus \lambda_1 \uplus ... \uplus \lambda_k)$, by Theorem 5.17 $\alpha \uplus \lambda_1 \uplus ... \uplus \lambda_k \uplus ... \uplus \lambda_i$ is safe for σ_k and $(\alpha \uplus \lambda_1 \uplus ... \uplus \lambda_k \uplus ... \uplus \lambda_i)(\sigma_k) = (\alpha \uplus \lambda_1 \uplus ... \uplus \lambda_k)(\sigma_k)$. Hence,

$$\begin{aligned} (\alpha \uplus \lambda_1 \uplus ... \uplus \lambda_{i-1})(\sigma_{i-1}) &= (\alpha \uplus \lambda_1 \uplus ... \uplus \lambda_i)(\sigma_{i-1}) \\ &... \\ (\alpha \uplus \lambda_1)(\sigma_1) &= (\alpha \uplus \lambda_1 \uplus ... \uplus \lambda_i)(\sigma_1) \end{aligned} \tag{16}$$

$$\alpha(G) = (\alpha \uplus \lambda_1 \uplus ... \uplus \lambda_i)(G) \tag{17}$$

Let us abbreviate $\beta_i := \alpha \uplus \lambda_1 \uplus ... \uplus \lambda_i$. Then from (14) and (16) by Theorem 5.25

$$\sigma'_i \cdot \sigma'_{i-1} \cdot ... \cdot \sigma'_1 = \beta_i(\sigma_i) \cdot \beta_i(\sigma_{i-1}) \cdot ... \cdot \beta_i(\sigma_1) = \beta_i(\sigma_i \cdot \sigma_{i-1} \cdot ... \cdot \sigma_1) \tag{18}$$

which is the promised connection between partial answers.

Clearly, variance of partial answers means variance of complete answers, and c.a.s. and resultants as well: For the cases when $G_n = \square$, we obtain, by (18), the expected relationship between the respective complete answers: $\sigma'_n \cdot ... \cdot \sigma'_1 = \beta_n(\sigma_n \cdot ... \cdot \sigma_1)$. 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 $\beta_n(G) = \alpha(G) = G'$, we obtain $\sigma'_n \cdot ... \cdot \sigma'_1 \restriction_{G'} = \beta_n(\sigma_n \cdot ... \cdot \sigma_1 \restriction_G)$, i. e. the same relationship. Finally, knowing that the *resultant* of step i is $R_i := (\sigma_i \cdot ... \cdot \sigma_1(G) \leftarrow G_i)$, we obtain

$$\begin{aligned} R'_i &= (\sigma'_i \cdot ... \cdot \sigma'_1(G') \leftarrow G'_i) \\ &= \beta_i(\sigma_i \cdot ... \cdot \sigma_1)(\alpha(G)) \leftarrow \beta_i(G_i), \text{ by (18), (13) and (15)} \\ &= \beta_i(\sigma_i \cdot ... \cdot \sigma_1)(\beta_i(G)) \leftarrow \beta_i(G_i), \text{ by (17)} \\ &= \beta_i((\sigma_i \cdot ... \cdot \sigma_1)(G)) \leftarrow \beta_i(G_i) = \beta_i(R_i), \text{ by Theorem 5.24} \end{aligned}$$

♢

Example 7.5 (similarity)**.** Assume the program

$$\begin{aligned} &son(S) \leftarrow male(S),\ child(S, P). && \%\, \bar{\mathcal{K}}_1 \\ &male(c).\ male(d).\ child(a, d). && \%\, \bar{\mathcal{K}}_2,\ \bar{\mathcal{K}}_3,\ \bar{\mathcal{K}}_4 \end{aligned}$$

An interpreter for LD-resolution may produce derivations

$$son(A) \hookrightarrow_{\mathcal{K}_1 : \sigma_1} male(A),\ child(C, A) \hookrightarrow_{\mathcal{K}_2 : \sigma_2} child(C, d)$$
$$son(B) \hookrightarrow_{\mathcal{K}'_1 : \sigma'_1} male(B),\ child(D, B) \hookrightarrow_{\mathcal{K}'_2 : \sigma'_2} child(D, d)$$

They are obviously similar, with $\mathcal{K}_1 = \bar{\mathcal{K}}_1[X, C]$, $\mathcal{K}'_1 = \bar{\mathcal{K}}_1[Y, D]$, $\mathcal{K}_2 = \mathcal{K}'_2 = \bar{\mathcal{K}}_3$. The variables X, Y stand for actually used variables, which cannot be deduced from the form of derivations. From the queries, input clauses and resolvents we can further deduce relevant mgus $\sigma_1 = \binom{X}{A}$, $\sigma'_1 = \binom{Y}{B}$, $\sigma_2 = \binom{A}{d}$ and $\sigma'_2 = \binom{B}{d}$. The mappings are $\alpha = \binom{A}{B}$, $\lambda_1 = \binom{X\ C}{Y\ D}$ and $\lambda_2 = \varepsilon$. Clearly, they fulfill $(\alpha \uplus \lambda_1)(male(A),\ child(C, A)) = male(B),\ child(D, B)$ and $(\alpha \uplus \lambda_1)(\binom{B}{A}) = \binom{C}{B}$, also $(\alpha \uplus \lambda_1 \uplus \lambda_2)(child(C, d)) = child(D, d)$, and so on.

Observe that in step $1'$ there is a relevant mgu $\binom{B}{Y}$ as well, but even without knowing the resolvent, we know that $\binom{B}{Y}$ 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 *prenaming* 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, prenamings 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

G. Amato and F. Scozzari. Optimality in goal-dependent analysis of sharing. *Theory and Practice of Logic Programming*, 9:617–689, 2009.

K. R. Apt. *From logic programming to Prolog*. Prentice Hall, 1997.

K. Doets. Levationis Laus. *J. Logic Computation*, 3(5):487–516, 1993.

C. Dwork, P. Kanellakis, and J. C. Mitchell. On the sequential nature of unification. *J. Logic Programming*, 1:35–50, 1984.

E. Eder. Properties of substitutions and unifications. *J. Symbolic Computation*, 1(1):31–46, 1985.

J. Gallier. *Logic for computer science: Foundations of automatic theorem proving*. Longman Higher Education, 1986.

J. Herbrand. *Recherches sur la Théorie de la Démonstration*. PhD thesis, Université de Paris, 1930. Dated Apr 14, 1929.

G. Huet. *Résolution d'équations dans des langages d'ordre 1,2,...,ω*. PhD thesis, Paris VII, 1976.

D. Jacobs and A. Langen. Static analysis of logic programs for independent AND parallelism. *J. Logic Programming*, 13(2-3):291 – 314, 1992.

M. Kulaš. Toward the concept of backtracking computation. In L. Aceto, W. J. Fokkink, and I. Ulidowski, editors, *Proc. SOS'04*, volume 128, issue 1 of *ENTCS*, pages 39–59. Elsevier, 2005.

J. L. Lassez, M. J. Maher, and K. Marriott. Unification revisited. In M. Boscarol et al., editors, *Foundations of Logic and Functional Programming*, volume 306 of *LNCS*, pages 67–113. 1988.

J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2. edition, 1987.

J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11(3-4):217–242, 1991.

A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. on Prog. Languages and Systems*, 4(2):258–282, Apr. 1982.

J. McCarthy. Recursive functions of symbolic expressions and their computation by machine (part I). *J. of ACM*, 1960.

U. Neumerkel. ISO / IEC JTC1 SC22 WG17 Post-N225: Built-in predicates, current practice. Version 1.36, 2010. `www.complang.tuwien.ac.at/ulrich/iso-prolog/built-in_predicates`.

C. Palamidessi. Algebraic properties of idempotent substitutions. In *Proc. 17th ICALP*, 1990.

C. Pusch. Verification of compiler correctness for the WAM. In *Proc. TPHOLs*, pages 347–361, 1996.

J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. of ACM*, 12(1):23–41, 1965.

J. C. Shepherdson. The role of standardising apart in logic programming. *Theoretical Computer Science*, 129(1):143–166, 1994.

C. Urban, A. Pitts, and M. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1-3):473–497, 2004. Proof at `http://www.inf.kcl.ac.uk/staff/urbanc/Unification/`.

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

[369] Güting, R.H., Valdés, F., Damiani, M.L.:
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