First-order unification using variable-free relational algebra[☆]

Emilio Jesús Gallego Arias[a], James Lipton[b], Julio Mariño[a], Pablo Nogueira[a]

[a] *Universidad Politécnica de Madrid*
[b] *Wesleyan University*

Abstract

We present a framework for the representation and resolution of first-order unification problems and their abstract syntax in a variable-free relational formalism which is an executable variant of the Tarski-Givant relation algebra and of Freyd's allegories restricted to the fragment necessary to compile and run logic programs. We develop a decision procedure for validity of relational terms, which corresponds to solving the original unification problem. The decision procedure is carried out by a conditional relational-term rewriting system. There are advantages over classical unification approaches. First, cumbersome and underspecified meta-logical procedures (name clashes, substitution, etc.) and their properties (invariance under substitution of ground terms, equality's congruence with respect to term forming, etc.) are captured algebraically within the framework. Second, other unification problems can be accommodated, for example, existential quantification in the logic can be interpreted as a new operation whose effect is to formalize the costly and error prone handling of fresh names (renaming apart).

Key words: Unification, Relation Calculus, Rewriting, Abstract Syntax

1. Introduction

Tarski and Givant [41] observe that binary relations equipped with so-called quasi-projection operators faithfully capture all of classical first-order logic. Freyd and Scedrov [19] show that so-called tabular allegories satisfying certain conditions faithfully capture all of higher-order intuitionistic logic. Both of these formalisms effectively eliminate logical variables in different ways.

The elimination of first-order variables in logic via a mathematical translation is of considerable interest, and a significant number of mathematical formalisms have been developed to achieve this aim over the past century. In computer science, and especially in the case of declarative programming languages,

[☆] Work supported by CAM grants 3060/2006 (European Social Fund) and S-0505/TIC/0407 (PROMESAS) as well as MEC grant TIN2006-15660-C02-02 (DESAFIOS).

it provides a completely fresh approach to compilation, not unlike the translation of lambda calculus into combinators [42] or the program transformation performed by Warren's Abstract Machine [1].

From the standpoint of logic programming, we are not only interested in the algebraic semantics the mentioned formalisms provide, but also in the alternative approach to program transformation and execution they might suggest. A restricted variant of these formalisms was proposed in [6, 29], where logic programming problems (certain kinds of constraints in a fragment of first-order logic) are mapped (compilation) into variable-free relational representations (relational terms) of an appropriate relational theory, with resolution (computation) taking place via relational-term rewriting. This variant formalism restricts the foundational ones of Tarski, etc, in order to carve out an executable fragment.

The relational representation provides an abstract syntax (i.e., an axiomatic treatment of free and bound variables, quantifiers, abstraction, etc) for logic programs and a formal treatment of logic variables and unification, where meta-logical procedures (name clashes, substitution, etc) are now captured within an object-level algebraic theory.

However, unification is incorporated meta-logically as a black box via intersection of relational terms, with execution details left unspecified. It is only shown that unification is sound with respect to the chosen representation.

In this paper we adapt the framework and develop the missing piece, an algorithm for deciding equality constraint problems in the Herbrand domain, being this equivalent to the classical first-order unification problem. The algorithm proceeds by rewriting relational-term representations. We also present the semantics for our relational-term calculus and provide the proof of our version of the main theorem (equipollence).

Although first-order unification is a well-studied and solved problem, an algebraic approach has several advantages. First, cumbersome and underspecified meta-logical procedures (name clashes, substitution, etc.) and their properties (invariance under substitution of ground terms, equality's congruence with respect to term forming, etc.) are captured algebraically within the framework. Second, other constraint problems can be accommodated, for example, existential quantification in the logic can be interpreted as a new operation formalizing renaming apart.

In the first part of the paper, we overview and adapt the foundational material, in particular [6, 29]. We start in Section 2 with an overview of the relationship between logic, distributive relational algebras, and unification problems. The translation of first-order logic into a variable-free relational calculus is detailed in Section 3. Our semantics and proof are provided there. Section 4 overviews the current relational framework in which logic programs are executed. We focus on the encoding of terms within this framework, which serves as an essential data type for our algorithm.

In the second part of the paper, we present the unification algorithm. Section 5 defines solved forms for our relational representation. Section 6 presents the algorithm itself. A detailed example can be found in Section 7.

In the final part of the paper we examine related and future work, and wind up with our conclusions.

2. From logic to rewriting: algebraic logic programming

One of the main results in *Set Theory Without Variables* [41], due to Tarski and Givant (with an improved proof by Maddux), is the so-called *equipollence* theorem which states that every first-order sentence φ has a semantically equivalent equational counterpart $X_\varphi = \mathbf{1}$, with X_φ a relation expression and $\mathbf{1}$ the universal relation in the variable-free theory **QRA** of relation algebras with quasi-projections. Tarski and Givant also prove a stronger proof-theoretic version of this result, and present a bijective recursive transformation of sentences and their first-order proofs to their associated relation expressions and equational derivations.

We use a slightly modified version in order to translate a formula with free variables to the relation of all the instances of such a formula that make it true. Assume $[\![\_]\!]$ is some set-theoretic interpretation for relations between finite sequences $\langle a_1, \ldots, a_n \rangle$, with $a_i \in \mathcal{H}_\Sigma$ elements of the Herbrand domain of a first-order signature Σ. The equipollence theorem is:

Theorem 2.1 (Equipollence)**.** Let $(\varphi)^r$ be the relational translation for the formula φ. For an open formula φ with free variables among $x_1, \ldots, x_n$:

$$(\langle a_1, \ldots, a_n \rangle, \langle a_1, \ldots, a_n \rangle) \in [\![(\varphi)^r]\!] \iff \mathcal{H}_\Sigma \models \varphi[a_1/x_1, \ldots, a_n/x_n]$$

In words, a term-sequence $\langle a_1, \ldots, a_n \rangle$ belongs to $[\![(\varphi)^r]\!]$ iff substituting $a_1, \ldots, a_n$ for *all* its free variables $x_1, \ldots, x_n$ makes the formula valid. Intuitively, $\wedge$ will correspond to $\cap$, $\vee$ will correspond to $\cup$, $\exists$ is represented by a quasi-projection and $\neg$ is the complement.

This important equivalence allows us to formalize logic programs as relational expressions. We then convert the algebraic theory of distributive relation algebras into a rewriting system which performs SLD-resolution. (We anticipate that the equipollence theorem presented in Section 3 is not directly usable in practice for executing logic programs. The reader has to wait until Section 4 for more details.)

If we represent a unification problem as first order formula with equality φ, unification then is equivalent to the decision problem $(\varphi)^r = \mathbf{0}$, with $\mathbf{0}$ being the empty relation and $(\cdot)^r$ the translation to be defined in subsequent sections.

Concretely, assume terms t_1, t_2, with free variables $\vec{y}$, and fresh $x_1, z \notin \vec{y}$. We encode the unification problem $t_1 \approx t_2$ using atomic formulas of the form $x_i = t$:

$$t_1 \approx^? t_2 \iff \exists x_1 . \exists \vec{y} . (x_1 = t_1 \wedge x_1 = t_2)$$

Notice the generality and expressiveness of the encoding, thanks to the presence of the existential quantifier. For example, Prolog-like unification with renaming apart (assume $\nu x.t$ means rename apart x in t) is encoded as:

$$\begin{aligned} \nu x.t_1 \approx^? t_2 \quad &\iff \quad \exists x_1 . \exists \vec{y} . ((\exists x . x_1 = t_1) \wedge x_1 = t_2) \\ &\iff \quad \exists x_1 . \exists \vec{y} . \exists z . (x_1 = t_1[z/x] \wedge x_1 = t_2) \end{aligned}$$

The encoding can be made more powerful. Adding negation allows us to handle disunification problems, as well as universal quantification.

The key point is that the result of $(\cdot)^r$ is a ground relational expression, this is where variable elimination takes place.

Summarizing:

$$\begin{aligned} t_1 \approx^? t_2 \quad &\iff \quad \exists x_1.\exists \vec{y}.(x_1 = t_1 \wedge x_1 = t_2) \\ &\iff \quad (\exists x_1.\exists \vec{y}.(x_1 = t_1 \wedge x_1 = t_2))^r \neq \mathbf{0} \end{aligned}$$

As the resulting relational expression is ground, and the relational theory is equationally defined, rewriting seems appropriate to handle relational terms. However, two requisites are proven difficult to implement. First, occurs checks, which forces us to make our rewriting system conditional and to compute a side-condition. Second, identifying functions. The rewriting system must match multiple arrows as one.

3. Logic without variables

3.1. Signature, terms, and sequences

Assume a permutative convention on symbols, i.e., f, g are different and so are i, j, etc. A first-order signature $\Sigma = \langle \mathcal{C}_\Sigma, \mathcal{F}_\Sigma, \mathcal{P}_\Sigma \rangle$ is given by $\mathcal{C}_\Sigma$, the set of constant symbols, $\mathcal{F}_\Sigma$, the set of term formers or function symbols, and $\mathcal{P}_\Sigma$, the set of predicate symbols. Function $\alpha : \mathcal{F}_\Sigma \to \mathbb{N}$ returns the arity of its function symbol argument. The set of logic variables is $\mathcal{X}$ and $x_i \in \mathcal{X}$. We write $\mathcal{T}_\Sigma(\mathcal{X})$ for the set of open terms over Σ. The set of open sequences $\mathcal{T}_\Sigma^+(\mathcal{X})$ over $\mathcal{T}_\Sigma$ is defined with the addition to the signature of a right-associative list-cons-like operator.

We write $\langle t_1, \ldots, t_n \rangle \vec{x}$, with abbreviation $\vec{t\vec{x}}$, for an open sequence of $t_1, \ldots, t_n$ terms where $\vec{x}$ is a variable standing for an arbitrary open sequence. Thus, the open sequence $\langle t_1, t_2 \rangle \vec{x}$ denotes all term sequences beginning with t_1, followed by t_2, and followed by an arbitrary term sequence.

3.2. Relational language

The relational language R is built from a countable set of relation variables R, S, T, etc, all in R_{var} (not to be confused with first-order logic-variables), a set of relational constants R_Σ built from Σ, and a fixed set of relational constants and operators detailed below. Let us begin with R_Σ. Each constant $a \in \mathcal{C}_\Sigma$ defines a constant $(a, a) \in \mathsf{R}_\Sigma$, each function symbol $f \in \mathcal{F}_\Sigma$ defines a constant R_f in R_Σ, and each predicate $r \in \mathcal{P}_\Sigma$ defines a constant r in R_Σ. Formally:

$$\mathsf{R}_\Sigma \;=\; \{\mathsf{r} \mid r \in \mathcal{P}_\Sigma, \} \cup \{\mathsf{R}_f \mid f \in \mathcal{F}_\Sigma, \} \cup \{(a, a) \mid a \in \mathcal{C}_\Sigma\}$$

The full relational language R is given by the following grammar:

$$\begin{aligned} \mathsf{R}_{atom} \quad &::= \quad \mathsf{R}_{var} \mid \mathsf{R}_\Sigma \mid id \mid di \mid \mathbf{1} \mid \mathbf{0} \mid hd \mid tl \\ \mathsf{R} \quad &::= \quad \mathsf{R}_{atom} \mid \mathsf{R}^\circ \mid \mathsf{R} \cup \mathsf{R} \mid \mathsf{R} \cap \mathsf{R} \mid \mathsf{R}\mathsf{R} \mid \mathsf{R} \setminus \mathsf{R} \mid \mathbf{fp}\ \mathsf{R}_{var}\ .\ \mathsf{R} \end{aligned}$$

The constants $\mathbf{1}, \mathbf{0}, id, di$ respectively denote universal relation (whose standard semantics is the set of all ordered pairs on a certain set), empty relation, identity (diagonal) relation, and identity's complement. Operators $\cup$ and $\cap$ represent union and intersection whereas juxtaposition RR represents relation composition and "\" denotes difference or relative complement. For better readability we write sometimes $R; S$ for RS. R° is the converse of R, i.e., the relation obtained by swapping domain and codomain. Set-theoretically, R's domain is $\{x|(x, \_) \in R\}$ and its codomain is $\{x|(\_, x) \in R\}$.

The computer science applications in this paper do not require the full expressive power of R's grammar; in particular, complementation, relation variables R_{var}, and the fixed-point operator **fp**. Many logic programming languages (e.g. FOHC,HOHC, HOHH, [35]) are based on a fragment of intuitionistic logic for which it suffices, in a relational translation, to have available the complement di of the identity (or diagonal) constant id [6]. Full complementation is briefly discussed and used here for completeness; in particular, to state below the equipollence theorem in first-order classical logic, which inspired the relation applications in the cited papers. The other operators are included for compatibility with earlier definitions of R where they are used to translate arbitrary Horn Clause programs to closed-form relation expressions using the fixed-point binding operator **fp** and bound relation variables.

3.3. Projections

Tarski showed how to axiomatize the notion of products and projections of the first and second coordinate in a relational variable-free manner. The relations hd and tl are equationally axiomatized as follows:

$$hd(hd)^\circ \cap tl(tl)^\circ \subseteq id \qquad (hd)^\circ hd = (tl)^\circ tl = id \qquad (hd)^\circ tl = \mathbf{1}$$

The first condition formalizes the fact that pairs are uniquely determined by their first and second coordinates, the second, that hd and tl are functional relations (equivalently, their converses are injective), and the third, that any two elements can occur as either the head or the tail of a pair.

Definition 3.1. Define the countable sequence $\{P_i \mid 1 \leq i\}$ of projection relations as follows:

$$P_1 \;=\; hd \qquad P_2 \;=\; tl; hd \qquad \cdots \qquad P_n \;=\; tl^{n-1}; hd \qquad \ldots$$

In the standard semantics to be discussed below, P_i is a relation between a formal vector with at least $i + 1$ components and its i-th component. That is to say, P_i's set-theoretic interpretation consists of pairs (u, u_i), where u is a sequence.

It will also be convenient to introduce, for every function symbol f of arity n in the signature Σ the derived *labelled projections* f_i^n defined by $\mathsf{R}_f; P_i$. We will only need these projections when the intended interpretation of f is injective, as in the case of term models.

3.4. Equational formulas

We now give some definitions prior to sketching a simple proof of a semantic form of the equipollence theorem. The equational atomic statements of $\mathcal{L}_\Sigma$ are of the form $t_1 = t_2$ which can be rewritten (after introducing a new variable x) as an existentially quantified conjunction of two *basic* equations: $\exists x(x = t_1 \wedge x = t_2)$. If we continue introducing variables and equations, we can write this as an existential conjunction of *elementary* or *flat* equations of the form $x = a$, $x = y$ or $x = f(y_1, \ldots, y_n)$ where a is a constant and f a function symbol in Σ, and x, y, y_i are fresh variables. We assume all equational atomic formulas are of this form. Similarly, all atomic formulas $r(t_1, \ldots, t_n)$ where r is a relation symbol of arity n in the language $\mathcal{L}$, can be written in the form $r(x_{i_1}, \ldots, x_{i_n}) \wedge E$, where E is a conjunction of elementary equations. Fresh variables are being introduced, but only to eliminate them subsequently.

3.5. Semantics

We briefly sketch a standard semantics for our relational term calculus. We will give a more formal and also more specialized definition of model when we restrict attention to unification over the term model below. Let Σ be a signature and $\mathcal{L}_\Sigma$ a first-order language containing, in addition to the symbols of Σ, equality, and the predicate letters $r_1, \ldots, r_n, \ldots$ of different arities. Let $\mathfrak{A}$ be a first-order structure for this language, with underlying set A. Let the interpretations of the constant symbols c and function symbols f of Σ in this algebra be denoted by c^A, f^A

Let A^* be the set of all *finite* sequences in A, and $A^\dagger = A \cup A^* \cup A^{**} \cup \cdots \cup A^{n*} \cup \cdots$ all hereditarily finite sequences over A. Let R_A be the set of pairs of members of $A^\dagger$. We then make the power set of R_A into a model of the relation calculus by interpreting atomic relation terms in a certain canonical way, and the operators in their standard set-theoretic interpretation. Notice that we interpret *hd* and *tl* in this model as operations on sequences similar to the head and tail operations on lists. The intention is to identify expressions such as $\langle a, \langle b, c\rangle\rangle$ with $\langle a, b, c\rangle$. We use the notational convention of writing vector variables $\vec{x}$ to denote sequences $\langle x_1, \ldots, x_m\rangle$ for any $m > 0$.

Definition 3.2. Given a first-order model $\mathfrak{A}$, a relational $\mathfrak{A}$**-interpretation** is

a mapping $[\![\_]\!]_{\mathfrak{A}}$ of relational terms into R_A satisfying

$$\begin{array}{lcl}
[\![(c,c)]\!]_{\mathfrak{A}} & = & \{(c^A, c^A)\} \\
[\![id]\!]_{\mathfrak{A}} & = & \{(u,u) \mid u \in A^\dagger\} \\
[\![di]\!]_{\mathfrak{A}} & = & \{(u,v) \mid u \neq v\} \\
[\![hd]\!]_{\mathfrak{A}} & = & \{(\langle t_1, t_2, \ldots, t_m\rangle, t_1) \mid t_i \in A^\dagger, m \geq 1\} \\
[\![tl]\!]_{\mathfrak{A}} & = & \{(\langle t_1, t_2, \ldots, t_m\rangle, \langle t_2, \ldots, t_m\rangle) \mid t_i \in A^\dagger, m \geq 1\} \\
[\![R \cup S]\!]_{\mathfrak{A}} & = & [\![R]\!]_{\mathfrak{A}} \cup [\![S]\!]_{\mathfrak{A}} \\
[\![R \cap S]\!]_{\mathfrak{A}} & = & [\![R]\!]_{\mathfrak{A}} \cap [\![S]\!]_{\mathfrak{A}} \\
[\![R \setminus S]\!]_{\mathfrak{A}} & = & [\![R]\!]_{\mathfrak{A}} \setminus [\![S]\!]_{\mathfrak{A}} \\
[\![RS]\!]_{\mathfrak{A}} & = & \{(x,y) \mid \exists v((x,v) \in [\![R]\!]_{\mathfrak{A}} \wedge (v,y) \in [\![S]\!]_{\mathfrak{A}})\} \\
[\![\mathbf{1}]\!]_{\mathfrak{A}} & = & \mathsf{R}_A \\
[\![\mathbf{0}]\!]_{\mathfrak{A}} & = & \emptyset \\
[\![\mathsf{R}_f]\!]_{\mathfrak{A}} & = & \{(\vec{x}, \vec{x}) \mid \vec{x} = \langle t_0, t_1, \ldots, t_n\rangle \ \wedge \ t_0 = f^A(t_1, \ldots, t_n)\} \\
[\![\mathsf{r}]\!]_{\mathfrak{A}} & = & \{(\vec{x}, \vec{x}) \mid \vec{x} = \langle t_1, \ldots, t_n\rangle \wedge r^A(t_1, \ldots, t_n)\}
\end{array}$$

The semantics of the labelled projections is as follows:

$$[\![f_i^n]\!]_{\mathfrak{A}} = \{(x,y) \mid \exists v_1 \ldots v_{n-1}(x = f^A(v_1, \ldots, v_{i-1}, y, v_{i+1}, \ldots, v_{n-1}))\}$$

It is often assumed—for example, in logic programming—that the only relation in Σ is equality. This is natural given that we deal with term-models over a signature, where function symbols f are interpreted as "themselves", that is, as the injective function $\langle t_1, \ldots, t_n\rangle \mapsto f(t_1, \ldots, t_n)$. In this case relational primitives for predicates are not needed.

3.6. Logic into relation calculus

We define a translation $(\_)^r$ from formulas ψ in the language of $\mathcal{L}_\Sigma$ (consisting of the signature Σ, equality, and predicate symbols $r_1, r_2, \ldots$ of different arities), to relation expressions as follows. Let n be a natural number greater than the largest number of variables occurring in any sentence to be considered below.

Now we define

$$S_1 = P_1 \qquad S_2 = P_2 \qquad \ldots \qquad S_{n-1} = P_{n-1} \qquad S_n = tl^n.$$

where the P_i are given in Definition 3.1. Observe that, in the standard interpretation for $1 \leq i \leq n$, we have $(u,v) \in [\![S_k]\!]$ iff v is the k^{th} component of u. Now define

$$Q_i^n = \bigcap_{j \neq i \leq n} S_j(S_j)^\circ \qquad id_n = \bigcap_{j \leq n} S_j(S_j)^\circ$$

Observe that $(u,v) \in [\![id_n]\!]$ means u and v are vectors of length at least n and have the same components u_j for all $j \in \{1 \ldots n\}$.

$u[\![Q_i]\!]v$ means all but the i-th component of u and v agree. Let $x_1, \ldots, x_s$ be all the variables, free or bound, that may occur in ψ. Recall all equational

atomic formulas ψ may be taken *elementary*: either $x_i = a, x_i = x_j$ or $x_{i_0} = f(x_{i_1}, \ldots, x_{i_n})$. First we translate the atomic formulas as follows:

$$\begin{array}{rcl} (x_i = a)^r & = & S_i(a,a)S_i{}^\circ \cap id_n \\ (x_i = x_j)^r & = & S_iS_j{}^\circ \cap id_n \\ (x_{i_0} = f(x_{i_1}, \ldots, x_{i_n}))^r & = & \bigcap_j S_{i_j}S_j{}^\circ; \mathsf{R}_f; S_jS_{i_j}{}^\circ \cap id_n \\ (r(x_{i_1}, \ldots, x_{i_n}))^r & = & \bigcap_j S_{i_j}S_j{}^\circ; \mathsf{r}; S_jS_{i_j}{}^\circ \cap id_n \end{array}$$

The nonatomic formulas are translated as follows:

$$\begin{array}{rclrcl} (\varphi \wedge \psi)^r & = & (\varphi)^r \cap (\psi)^r & (\varphi \vee \psi)^r & = & (\varphi)^r \cup (\psi)^r \\ (\exists x_i \varphi)^r & = & Q_i(\varphi)^r Q_i \cap id_n & (\neg\varphi)^r & = & id_n \setminus (\varphi)^r \end{array}$$

Finally, in the case where the function symbol f will only be interpreted as an injective function, either because we are restricting attention to a class of models in which this is the case, as in the term model for a signature, or because it is implied by the axioms of the theory being used, we can simplify the translation of one of the atomic clauses above:

$$(x_{i_0} = f(x_{i_1}, \ldots, x_{i_n}))^r = \bigcap_{j=1}^n S_{i_o}; f_j^n; S_{i_j}^o$$

Then we have the following result, one of many established by Tarski and Givant [41] in a form more closely suited to our needs here. It was independently established by Freyd in a categorical setting [19]. The statement of the theorem and its proof are new, but draw on similar ideas independently proved by Freyd, Maddux and Tarski.

Theorem 3.1 (Freyd, Maddux, Tarski)**.** Let $\mathcal{L}$ be a first-order language, $\mathfrak{A}$ a model over $\mathcal{L}$. Let ψ be a first-order formula over $\mathcal{L}$, and let $x_1, \ldots, x_n$ contain all the variables, free or bound, that may occur in ψ. Then, if ψ is a closed formula $\mathfrak{A} \models \psi \iff [\![(\psi)^r]\!] = [\![id_n]\!]$. If ψ is open, with free variables $x_{i_1}, \ldots, x_{i_m}$ among $x_1, \ldots, x_n$

$$[\![(\psi)^r]\!] = \{(\langle a_1, \ldots, a_n\rangle, \langle a_1, \ldots, a_n\rangle) \mid \mathfrak{A} \models \psi[a_1/x_1, \ldots, a_m/x_m]\}$$

Note that $\mathfrak{A} \models \psi[a_1/x_1, \ldots, a_m/x_m]$ is equivalent to $\mathfrak{A} \models \psi[a_{i_1}/x_{i_1}, \ldots, a_{i_m}/x_{i_m}]$ because only the x_{i_j} occur freely in ψ, In particular, the first conclusion of the theorem is just a special case of the second, where none of the x_{i_j} are free in ψ.

The proof is a straightforward induction on the structure of formulas

Proof.

$\underline{\psi \equiv (x_i = a)}$:

Let $\vec{u}$ be an n-tuple of terms in $\mathfrak{A}$, and suppose $(\vec{u}, \vec{u}) \in [\![(x_i = a)^r]\!] = [\![S_i(a,a)S_i^\circ \cap id_n]\!]$. This implies that $u_i = a$, hence $\mathfrak{A} \models (x_i = a)[u_1/x_1, \ldots, u_n/x_n]$. Conversely $u_i = a$ forces $(\vec{u}, \vec{u}) \in [\![(x_i = a)^r]\!]$.

$\underline{\psi \equiv (x_i = x_j)}$:

Suppose $(\vec{u}, \vec{u}) \in [\![(x_i = x_j)^r]\!] = [\![S_iS_j^\circ \cap id_n]\!]$. Then we have $u_i = u_j$ and $\mathfrak{A} \models (x_i = x_j)[\vec{u}/\vec{x}]$. The converse is also immediate.

$\underline{\psi \equiv x_{i_0} = f(x_{i_1}, \ldots, x_{i_n})}$:
Suppose $(\vec{u}, \vec{u}) \in [\![(x_{i_0} = f(x_{i_1}, \ldots, x_{i_n}))^r]\!]$, that is to say, in

$$\bigcap\nolimits_j S_{i_j} S_j{}^{\circ}; \mathsf{R}_f; S_j S_{i_j}{}^{\circ} \cap id_n.$$

This is clearly equivalent to $u_{i_1} = f^A(u_{i_2}, \ldots, u_{i_m})$ and hence the conclusion of the theorem for this case.

$\underline{\psi \equiv r(x_{i_1}, \ldots, x_{i_n})}$: Suppose $(\vec{u}, \vec{u}) \in [\![(r(x_{i_1}, \ldots, x_{i_n}))^r]\!]$ i.e. $(\vec{u}, \vec{u})$ lies in

$$\bigcap\nolimits_j [\![S_{i_j} S_j{}^{\circ}; \mathsf{r}; S_j S_{i_j}{}^{\circ}]\!] \cap [\![id_n]\!].$$

Now suppose $\vec{v}$ is a sequence such that $\vec{u}[\![S_{i_j} S_j{}^{\circ}]\!]\vec{v}$. Then for each j between 1 and m $v_j = u_{i_j}$, so $\langle (u_{i_1}, \ldots, u_{i_m}, u_{i_1}, \ldots, u_{i_m}) \rangle \in [\![\mathsf{r}]\!]$, which implies that $\mathfrak{A} \models r(u_{i_1}, \ldots, u_{i_m})$. Conversely $\mathfrak{A} \models r(a_1, \ldots, a_m)$ implies that for any $\vec{u}$ with all $u_{i_j} = a_j$, $\vec{u}, \vec{u}) \in [\![(r(x_{i_1}, \ldots, x_{i_n}))^r]\!]$.

$\underline{\psi \equiv \varphi_1 \wedge \varphi_2}$:
Suppose $(\vec{u}, \vec{u}) \in [\![(\varphi_1 \wedge \varphi_2)^r]\!]$. Then $(\vec{u}, \vec{u}) \in [\![(\varphi_1)^r]\!]$ and $(\vec{u}, \vec{u}) \in [\![(\varphi_2)^r]\!]$. By the induction hypothesis, $\mathfrak{A} \models \varphi_1[\vec{u}/\vec{x}]$ and $\mathfrak{A} \models \varphi_2[\vec{u}/\vec{x}]$ and hence $\mathfrak{A} \models (\varphi_1 \wedge \varphi_2)[\vec{u}/\vec{x}]$

$\underline{\psi \equiv \neg\varphi}$:
Suppose $(\vec{u}, \vec{u}) \in [\![(\neg\varphi)^r]\!]$. Then $(\vec{u}, \vec{u}) \in id_n \setminus [\![(\varphi)^r]\!]$ which is equivalent to saying that $(\vec{u}, \vec{u}) \in [\![(\varphi)^r]\!]$ is not the case. Using the induction hypothesis, this is equivalent to $\mathfrak{A} \not\models \varphi[\vec{u}/\vec{x}]$, and hence to $\mathfrak{A} \models \neg\varphi[\vec{u}/\vec{x}]$.

$\underline{\psi \equiv \exists x_i \varphi}$:
If $(\vec{u}, \vec{u}) \in [\![(\exists x_i \varphi)^r]\!]$ we must have $(\vec{u}, \vec{u}) \in [\![Q_i(\varphi)^r Q_i]\!]$. This means there is an n-tuple of terms $\vec{v}$ with $v_j = u_j$ for every j between 1 and n except i, and $(\vec{v}, \vec{v}) \in [\![(\varphi)^r]\!]$. By the induction hypothesis, this means $\mathfrak{A} \models \varphi[\vec{v}/\vec{x}]$. But this is equivalent to $\mathfrak{A} \models \exists x_i \varphi[\vec{u}/\vec{x}]$. The converse is left to the reader. □

The proof can be specialized to the case of a language consisting only of a signature Σ, with equality as the sole predicate symbol, with relation structure $\mathsf{R}_{\mathcal{H}_\Sigma}$ induced by the term model $\mathcal{H}_\Sigma$. The only case that changes is the one for the atomic formula $x_{i_0} = f(x_{i_1}, \ldots, x_{i_m})$, since we use the modified translation

$$(x_{i_0} = f(x_{i_1}, \ldots, x_{i_n}))^r = \bigcap\nolimits_{j=1}^{n} S_{i_o}; f_j^n; S_{i_j}^{\circ}$$

But if $(\vec{u}, \vec{u})$ is a member of

$$\bigcap\nolimits_j [\![S_{i_0}; f_j^n; S_{i_j}^{\circ}]\!] \cap [\![id_n]\!],$$

then $u_{i_1} = f(u_{i_2}, \ldots, u_{i_{n+1}})$ and $\mathcal{H}_\Sigma \models \varphi[\vec{u}/\vec{x}]$. The converse is immediate.

In the next sections we will define and turn our attention to the relation calculus $\mathsf{Rel}\Sigma$, which is a fragment of the one used in the equipollence theorem but has no complementation and is more suitable for capturing Horn Clause programming (Section 4 and [6, 29]) as well as unification with constraints, our central concern. It should be noted that the absence of negation in $\mathsf{Rel}\Sigma$ is no handicap, vis-a-vis first-order formulas with negation over the Herbrand Universe, because of the well-known results of Mal'cev [33, 32] that any such formula is equivalent to a two-quantifier formula in which negation occurs only immediately preceding equations between terms. This can be modelled in $\mathsf{Rel}\Sigma$ using *di* for disequality.

4. The relation calculus as a programming language

We now concentrate on evaluation of relation terms as a programming formalism. We provide computation or rewriting rules in order to have a notion of evaluation, but we have to carefully choose an axiomatic formulation of the relation calculus that will ensure good termination properties.

We prepare the reader by providing a brief survey of previous work [6, 29]. For the sake of brevity, full technical details are omitted and we simply illustrate the ideas using examples. However, we spell out in more detail the parts concerning unification.

4.1. Combinatory logic programming

The questions that gave rise to the results in this paper were motivated by work on variable-free logic programming in relation calculi [6, 29]. We briefly summarize the results most relevant to the work carried out in the following sections. Logic programming languages, at least in principle, are based on the notion of computing directly with a logical description of a problem, that is to say, with specifications. Computation is reduced to proof-search in such a way as to ideally separate the programmer from some of the procedural aspects of computing, to eliminate concerns about *how* results are obtained and replace them with definitions that specify *what* is to be computed. One quickly finds, however, that many procedural concerns must be taken into account, not only in the choice of definitions, but in the metalogical management of proof search, including, in particular, treatment of logical variables, avoiding name clashes, etc. This meta-logical layer lives outside the specification semantics, although recent research in abstract syntax has worked towards providing an axiomatic foundation to this component (e.g., [7, 21]).

The Tarski-Givant theorems on formalizing mathematics without variables sketched above suggested a way to translate logic programs to variable-free terms in the relation calculus, essentially supplying a built-in abstract syntax. The resulting formalism, tailored to handle the fragment of logic of interest (e.g. Horn clauses) in logic programming, includes an evaluation mechanism, namely rewriting, in lieu of proof search. A simple example illustrates the main idea.

Consider the Prolog program for the reflexive transitive closure of a simple graph, and two queries.

```
conn(X,X).                          edge(a,b).
conn(X,Y):-edge(X,Z),conn(Z,Y).     edge(b,c).
                                    edge(a,l).
                                    edge(l,c).

| ?- conn(a,c).
| ?- conn(X,c).
```

Translating the program. We introduce the binary relation symbols *conn* and *edge* (and let the font difference suffice to distinguish between program predicates and the new symbols), and translate the program into a pair of relation equations. The *conn* identifier stands for a relation defined via an equation and *edge* stands for an expression which explicitly describes a finite binary relation on the Herbrand universe $\mathcal{H}$ of the program. For readability, we denote composition of relations by semicolon.

$$\begin{aligned} edge &= \{(a,b),(b,c),(a,l),(l,c)\} \\ conn &= id \cup edge; conn \end{aligned}$$

where *id* denotes the identity relation.

The first query above is represented by $\{(a,c)\} \cap conn$ and the second by $(\mathbf{1};(c,c)) \cap conn$ where $\mathbf{1}$ is the universal relation defined above. Hence $(\mathbf{1};(c,c))$ represents the set of all pairs whose second component is c. A solution to the first query would have be to be of the form $\{(a,c)\}$, confirming that $(a,c) \in conn$,, or otherwise $\mathbf{0}$, the empty relation. A solution to the second query should be a more explicit description, such as $\{(a,c),(b,c),(c,c),(l,c)\}$.

We illustrate how the computation of the second query can be carried out using simple rewriting rules that capture basic relation identities.

A natural first strategy is to unfold the recursive definition of *conn*:

$$\mathbf{1};(c,c) \cap conn = (\mathbf{1};(c,c) \cap id) \cup (\mathbf{1};(c,c) \cap edge; conn)$$

The first term can be reduced to (c,c), and the second can be further unfolded.

$$\begin{aligned} \mathbf{1};(c,c) \cap edge; conn &= \mathbf{1};(c,c) \cap edge; (id \cup edge; conn) \\ &= \mathbf{1};(c,c) \cap (edge; id \cup edge; edge; conn) \\ &= \mathbf{1};(c,c) \cap (edge \cup edge; edge; conn) \\ &= (\mathbf{1};(c,c) \cap edge) \cup (\mathbf{1};(c,c) \cap edge; edge; conn) \\ &= (b,c) \cup (l,c) \cup (\mathbf{1};(c,c) \cap edge; edge; conn) \end{aligned}$$

Continuing this way, we will eventually obtain the term $\mathbf{1};(c,c) \cap (edge; edge)$ which is equal to $\{(a,c)\}$. The next unfolding gives us $\mathbf{1};(c,c) \cap (edge)^{(3)}; conn$. Notice *edge* is finite, for any $n > 2$, $(edge)^{(n)} = \mathbf{0}$, thus we are done. The query evaluates to $(c,c) \cup \{(b,c),(l,c)\} \cup \{(a,c)\}$ which is simply $\{(c,c),(b,c),(l,c),(a,c)\}$.

$$R \cap R = R \qquad R \cap S = S \cap R \qquad R \cap (S \cap T) = (R \cap S) \cap T$$
$$R \cup R = R \qquad R \cup S = S \cup R \qquad R \cup (S \cup T) = (R \cup S) \cup T$$
$$R\,id = R \qquad R\mathbf{0} = \mathbf{0} \qquad \mathbf{0} \subseteq R \subseteq \mathbf{1}$$
$$R \cup (S \cap R) = R = (R \cup S) \cap R$$
$$R(S \cup T) = RS \cup RT \qquad (S \cup T)R = SR \cup TR$$
$$R \cap (S \cup T) = (R \cap S) \cup (R \cap T)$$
$$(R \cup S)^\circ = R^\circ \cup S^\circ \qquad (R \cap S)^\circ = S^\circ \cap R^\circ$$
$$R^{\circ\circ} = R \qquad (RS)^\circ = S^\circ R^\circ$$
$$R(S \cap T) \subseteq RS \cap RT \qquad RS \cap T \subseteq (R \cap TS^\circ)S$$
$$id \cup di = \mathbf{1} \qquad id \cap di = \mathbf{0} \qquad \mathbf{fp}x.\mathcal{E}(x) = \mathcal{E}(\mathbf{fp}x.\mathcal{E}(x))$$

Figure 1: The equational theory **DRA**.

$$\mathbf{1}(a,a)\mathbf{1} = \mathbf{1} \qquad (a,a)R(a,a) = (a,a) \cap R \qquad (a,a) \subseteq id$$
$$hd(hd)^\circ \cap tl(tl)^\circ \subseteq id \qquad (hd)^\circ hd = (tl)^\circ tl = id \qquad (hd)^\circ tl = \mathbf{1}$$
$$id_f \stackrel{\text{def}}{=} \bigcap_{1 \le i \le n} f_i^n (f_i^n)^\circ \subseteq id \qquad (f_j^n)^\circ f_i^n = \mathbf{1} \quad (i \neq j)$$
$$(f_i^n)^\circ f_i^n = id \qquad (f_i^n)^\circ g_j^m = \mathbf{0}$$
$$(f_1)_{i_1}^{n_1} (f_2)_{i_2}^{n_2} \ldots (f_k)_{i_k}^{n_k} \;\cap\; id = \mathbf{0}$$
$$hd^\circ f_i^n = 0 = tl^\circ f_i^n \qquad f_i^n hd = 0 = f_i^n tl \qquad hd \cap id = 0 = tl \cap id$$
$$id = \bigcup\{(a,a) : a \in \mathcal{C}_\Sigma\} \;\cup\; \bigcup\{id_f : f \in \mathcal{F}_\Sigma\}$$

Figure 2: The equational theory $\mathsf{Rel}\Sigma$.

4.2. Relational theories

It is well known that the theory of binary relations is not finitely axiomatizable [31, 19], so we need to use a specific axiomatization. Tarki's equipollence result uses the theory of distributive relation algebras with quasi-projections, **QRA**.

However, we use the sightly different setting of [6], where logic programs are represented as relations in two relational theories, namely, Distributive Relational Algebras (**DRA**, Fig. 1) and an specialized $\mathsf{Rel}\Sigma$ (Fig. 2) theory, which depends on the signature Σ of the logic program.

The latter captures algebraically Clark's Equality Theory [30], the domain closure axiom (shown in the last line: every term is either a constant or an application of some term former to terms), the occurs check (fifth line: no proper subterm of a term may be equal to the whole term), and open sequences of terms (second line: formalizes projection operations, sixth line: rules out the use of open sequences in term forming operations).

The modular law $RS \cap T \subseteq (R \cap TS^\circ)S$ (recall $X \subseteq Y$ means $X \cap Y = X$) has left and right equational versions, $T \cap RS = R(R^\circ T \cap S)$ and $T \cap RS = R(R^\circ T \cap S) \cap T$, both derivable from **DRA**. The fixed point equation (**fp**) allows us to represent recursive predicates.

The fundamental role of $\mathbf{DRA} + \mathsf{Rel}\Sigma$ will be clearer in subsequent sections, where we will derive the unification algorithm just by orienting some of the equations into a convergent rewrite system. This allows us to claim correctness almost for free, provided we show the rewriting system is terminating.

4.3. Rewriting and the modular law

SLD resolution can be implemented by orienting the equations of the relational theories. It turns out that a critical equation for logic-programming execution is the *modular law* [19].

Let us consider the binary relations over the set of closed terms induced by the signature a, b, s^1, that is to to say, two constants and a unary successor function. Suppose we wish to compute membership of the ordered pair $(s(s(a)), a)$ in the transitive closure R^* of the binary relation R representing the successor relation, i.e., informally, whose semantics is $\{(u, v) \mid u = s(v)\}$ where u, v are metavariables ranging over ground terms. We assume that membership in R is computed in one step by a black box, in the following way:

$$(s(a), a) \cap R \mapsto (s(a), a) \qquad\qquad (a, b) \cap R \mapsto \mathbf{0}$$

We also assume some other elementary operations with R and R°, e.g. composition with singleton relations, can be carried out in one step. Note that any pair (u, v) can be represented in our syntax as the term $(u, u)\mathbf{1}(v, v)$, or we can just introduce them as defined terms if we wish.

Thus we want to compute the value of $(s(s(a)), a) \cap (R \cup R^2 \cup \cdots \cup R^n \cup \cdots)$, to obtain either the result $\mathbf{0}$ or $(s(s(a)), a)$. If we proceed in the obvious way, we obtain.

$$(s(s(a)), a) \cap R \ \cup \ (s(s(a)), a) \cap R^2 \ldots$$

Proceeding from left to right, the computation will terminate if $(s(s(a)), a) \cap R^k = (s(s(a)), a)$ for some k. In fact, this will occur when $k = 2$. But what if we pick a term, such as (a, b) which does not lie in R^*? If we proceed in the same way we require a non-terminating computation to yield the output $\mathbf{0}$. For example, after n steps we have the term:

$$\mathbf{0} \cup \cdots \cup \mathbf{0} \cup (a, b) \cap R^n \ldots .$$

To solve this problem we introduce a new rewrite rule corresponding to the *equational left-modular law*:

$$T \cap RS \mapsto R(R^o T \cap S) \cap T.$$

We will also add the rule $R^* \mapsto R \cup RR^*$, in order to work with a definition of transitive closure, and avoid writing infinite expressions. We then unfold the computation

$$\begin{array}{rcl}
(a, b) \cap R^* & \mapsto & (a, b) \cap (R \cup RR^*) \\
& \mapsto & (a, b) \cap R \ \cup \ (a, b) \cap (RR^*) \\
& \mapsto & \mathbf{0} \cup \ R(R^o(a, b) \cap R^*) \cap (a, b) \\
& \mapsto & \mathbf{0} \cup \ R(\mathbf{0} \cap R^*) \cap (a, b) \\
& \mapsto & \mathbf{0} \cup \ \mathbf{0} \\
& \mapsto & \mathbf{0}
\end{array}$$

On the third line, $R^o(a, b)$ reduces in one step to $\mathbf{0}$ since every member of R^o is a pair of ground terms of the form $(v, s(v))$ and no term of the form $s(v)$ is identical to the constant a.

This example illustrates how the use of the modular law can improve termination properties when we consider relation terms as programs that can be evaluated.

4.4. The unification problem

In [6, 29], logic program execution is achieved by means of a simple set of rewrite rules based on the relational theories above. However, unification is included in the rewriting rules as a black box. Terms t in the Herbrand universe are represented via certain variable-free relation expressions $\dot{K}(t)$ (discussed in detail below), and unification was shown to be soundly represented by intersections of such terms using, e.g. the rules in Table 1.

$$
\begin{array}{lcll}
\dot{K}(u) \cap \dot{K}(v) & \overset{P}{\longmapsto} & \dot{K}(\theta v') & (\theta = \mathsf{mgu}(u, v'), \parallel u \parallel \leq \parallel v \parallel) \\
\dot{K}(v) \cap \dot{K}(u) & \overset{P}{\longmapsto} & \dot{K}(\theta v') & (\theta = \mathsf{mgu}(u, v'), \parallel u \parallel \leq \parallel v \parallel) \\
\dot{K}(v) \cap \dot{K}(u) & \overset{P}{\longmapsto} & \mathbf{0} & (u, v \text{ not unifiable})
\end{array}
$$

$v' = v$ renamed apart.

Table 1: Meta-reductions

The details of how the mgu was to be computed were left to the implementation.

In some regards, this black-box treatment is a natural choice, since this is precisely where constraint systems are added to logic programming engines in conventional constraint logic programming languages, such as CLP(X) [26]. However, the benefits of variable-elimination may be lost here if unification is carried out using typical algorithms that operate directly on the variable names in terms. Then variables must be restored just for the sake of this step.

The translation K from terms $t \in \mathcal{T}_\Sigma(\mathcal{X})$ to terms in R is defined in a way that every ground instance of t is in $[\![K(t)]\!]$. We will also define the set of relational terms in K's image inductively and call them $\mathbf{U}$-terms. The result of applying the most general unifier of two terms $t_1, t_2 \in \mathcal{T}_\Sigma(\mathcal{X})$ to any of them is then represented by $K(t_1) \cap K(t_2)$. It is therefore of interest to define and compute a normal form for such an expression which conveys all the desired information.

In a relational setting we have no variables and no syntactic notion of substitution. Therefore, the natural output of a unification procedure is some sort of normalized constraint with the right semantics. The next paragraphs will develop this idea.

Definition 4.1. $K : \mathcal{T}_\Sigma(\mathcal{X}) \to \mathsf{R}$ is defined by induction on $\mathcal{T}_\Sigma(\mathcal{X})$ terms:

$$\begin{aligned} K(a) &= (a,a)\mathbf{1} \\ K(x_i) &= (P_i)^\circ \\ K(f(t_1,\ldots,t_n)) &= \bigcap_{i\leq n} f_i^n K(t_i) \end{aligned}$$

For example, $K(f(x_1, g(a, x_2)))$ yields $f_1^2 P_1^\circ \cap f_2^2(g_1^2(a,a)\mathbf{1} \cap g_2^2 P_2^\circ)$.

It should be noted that the symbols $t_1, \ldots, t_n$ in the expression $K(f(t_1, \ldots, t_n)$ in the preceding definition, are *metavariables* denoting arbitrary terms in $\mathcal{T}_\Sigma(\mathcal{X})$. In particular, several of them may refer to the same variable x_j or the same compound term.

In general, the relations semantically denoted by the terms $K(t)$ are non-coreflexive, that is, domain and co-domain are not equal. Indeed, consider the base case of the definition, $K(a) = (a, a)\mathbf{1}$. The semantics of the resulting relation term is the set $\{(a^A, \vec{u})|\vec{u} \text{ any sequence of terms}\}$. Indeed, we could informally state that the domain any relation generated by $K(t)$ is the set of t's instantiations whereas the co-domain is the set of instantiations of the variables of t. So, for a ground term t, it follows that $[\![K(t)]\!] = \{(t^A, \vec{u})|\vec{u} \text{ any sequence of terms}\}$. The fact that in the ground case the co-domain is the set of all sequences of terms reflects the invariability of t under substitution.

In order to simplify some computations, and especially in the context of the logic programming simulations in [6, 29] , we need to make use of a variant of this translation. So we define the $\dot{K}$ translation of a term as the coreflexive version of K extended over sequences:

Definition 4.2 ($\dot{K}$).

$$\dot{K}(\langle t_1, \ldots, t_n\rangle) = (P_1 t_1 \cap \cdots \cap P_n t_n)\mathbf{1} \cap id$$

If we instantiate the (modified) equipollence theorem to this particular case we get:

Lemma 4.1. For all terms t_1, t_2 in $\mathcal{T}_\Sigma(\mathcal{X})$ with variables $x_1, \ldots, x_n$ and an arbitrary open sequence $\vec{u}$:

$$(t_1\sigma, \langle a_1, \ldots, a_n\rangle\vec{u}) \in [\![K(t_1) \cap K(t_2)]\!] \quad \text{iff} \quad \mathcal{H} \models t_1\sigma = t_2\sigma$$

for all grounding substitutions $\sigma = a_1, \ldots, a_n/x_1, \ldots, x_n$.

This result is proved in detail in [29]. In words, given an open term t all pairs relating ground instances of t with grounding elements $a_1, \ldots, a_n$ belong to $[\![K(t)]\!]$.

Definition 4.3 (U-terms). The set of **U**-terms (relational terms in K's image) is defined inductively:

- $\mathbf{0} \in \mathbf{U}$, $(a, a)\mathbf{1} \in \mathbf{U}$ for every $(a, a) \in \mathsf{R}_\Sigma$, and $P_i^\circ \in \mathbf{U}$ for every $i \in \mathbb{N}$.

- If $R \in \mathbf{U}$ then $f_i^n R \in \mathbf{U}$ for every $f_i^n \in \mathsf{R}_\Sigma$.
- If $R_1, \ldots, R_n \in \mathbf{U}$ then $R_1 \cap \cdots \cap R_n \in \mathbf{U}$.

Definition 4.4 (Unification problem). The unification problem in this relational setting is to reduce, given terms $t_1, t_2 \in \mathcal{T}_\Sigma(\mathcal{X})$, $K(t_1) \cap K(t_2)$ into an equivalent $\mathbf{U}$-term.

The unification procedure above effectively decides unifiability, since the following holds:

$$[\![K(t_1) \cap K(t_2)]\!] = \begin{cases} [\![K(t_1\sigma)]\!] & \text{where } \sigma = \mathbf{mgu}(t_1, t_2) \text{ if exists} \\ \emptyset & \text{otherwise} \end{cases} \tag{1}$$

5. Solved forms for U-terms

It is standard practice in decision problems to use a *solved form* which has a trivial decision procedure. The decision problem is then reduced to developing a method for producing solved forms [10]. At the core of our decision procedure is the theory $\mathsf{RelI}\Sigma$, for it provides a strong enough axiomatization of the original underlying algebra of finite trees.

Given the term $T = R \cap S$, we informally say that R is constrained by S in T and vice versa. We also say that R is obtained from $R \cap S$ by dropping the constraint S.

Definition 5.1 (P-constraint completeness). A term R is *P-constraint complete* or $\Xi(R)$ iff for all subterms t_1 and t_2 of R of the form:

$$t_1 = P_i^\circ \cap R_1 \cap \cdots \cap R_m$$
$$t_2 = P_j^\circ \cap S_1 \cap \cdots \cap S_n$$

if $i = j$ then the equality $t_1 = t_2$ holds modulo $\cap$-commutativity.

This formally captures the notion that every P_i° appearing in a term must have the same set of constraints. Some examples of P-constraint-complete terms are: $P_1^\circ \cap P_2^\circ$, $f_1^2(P_1^\circ \cap R) \cap g_1^2(R \cap P_1^\circ)$ and $f_1^2(P_1^\circ \cap R) \cap g_1^2(S \cap P_2^\circ)$. Some non-P-constraint-complete terms are: $P_1^\circ \cap f_1^1 P_1^\circ$ and $f_1^2(P_1^\circ \cap R) \cap g_1^2(S \cap P_1^\circ)$.

Definition 5.2 (Indexed P-constraint completeness). A term R is *P-constraint complete on i* or $\Xi_i(R)$ iff for all subterms t_1, t_2 of R of the form:

$$t_1 = P_j^\circ \cap R_1 \cap \cdots \cap R_m$$
$$t_2 = P_k^\circ \cap S_1 \cap \cdots \cap S_n$$

if $i = j = k$ then the equality $t_1 = t_2$ holds modulo $\cap$-commutativity.

Definition 5.3 (Solved form). **U**-terms in solved form are inductively defined for all a, i, j, f, g as:

R			for $R \in \{(a,a)\mathbf{1}, P_i^\circ, \mathbf{0}\}$
R	$\cap$	S	for $R, S \in \{(a,a)\mathbf{1}, P_i^\circ\}$
$(a,a)\mathbf{1}$	$\cap$	$f_i^j R$	
$f_i^n R$			if R in solved form.
P_i°	$\cap$	$f_j^n R$	if R in solved form.
$f_i^m R$	$\cap$	$g_j^n S$	if $f \neq g$.
$f_i^n R$	$\cap$	$f_j^n S$	if R, S in solved form and $\Xi(R \cap S)$.
R_1	$\cap \cdots \cap$	R_n	if every pair $R_i \cap R_j, i \neq j$ in solved form.

If a term t is in solved form we say $\mathit{solved}(t)$. Let $\mathbf{U_S} = \{t \in \mathbf{U} \mid \mathit{solved}(t)\}$ and $\mathbf{U_N} = \{t \in \mathbf{U} \mid \neg \mathit{solved}(t)\}$. By definition, $\mathbf{U_S}$ and $\mathbf{U_N}$ partition $\mathbf{U}$. The inductive definition of unsolved forms is thus obtained from the logical negation of the solved form definition:

Definition 5.4 (Unsolved form). **U**-terms in unsolved form are those terms not in solved form. Inductively, for all i, j, f:

$f_i^n R$			if R not in solved form.
P_i°	$\cap$	$f_j^n R$	if R not in solved form.
$f_i^n R$	$\cap$	$f_i^n S$	
$f_i^n R$	$\cap$	$f_j^n S$	if $\neg\Xi(R \cap S)$ or any of R, S not in solved form.
R_1	$\cap \cdots \cap$	R_n	if $R_i \cap R_j$ not in solved form for some $i, j, i \neq j$.

The decision procedure for solved forms is an exhaustive check for incompatible intersections.

Lemma 5.1 (Validity of solved forms)**.** For any term $t \in \mathbf{U_S}$, we can always decide whether $t = \mathbf{0}$ or, what is the same, whether $[\![t]\!] = \emptyset$.

Proof. By cases on t:

- R and $R \cap S$ for $R, S \in \{(a,a)\mathbf{1}, P_i^\circ\}$: Follows directly by term semantics.
- $(a,a)\mathbf{1} \cap f_i^j R$: Always $\mathbf{0}$, because the relation domains are disjoint.
- $f_i^n R$: We check R for validity.
- $P_i^\circ \cap f_j^n R$: We check R for validity.
- $f_i^m R \cap g_j^n S$: Always $\mathbf{0}$, because relation domains are disjoint ($f \neq g$).
- $f_i^n R \cap f_j^n S$: We check R and S for validity. This check is enough because no term $f_i^n R \cap f_i^n S$ is in solved form. Consequently, the domain of the resulting relations is known. Given that $\Xi(R \cap S)$ holds, the codomain is also known, as every projection P_i° into the codomain shares the same set of constraints, so validity of the codomain is effectively reduced to validity of constraints.

- $R_1 \cap \cdots \cap R_n$: Every pair $R_i \cap R_j$ with $i \neq j$ is in solved form, thus the term's validity depends on the pairs' validity.

□

6. The algorithm

This section defines an algorithm for computing normal forms of **U**-terms. The core of the algorithm consists of two term rewriting systems whose effect can be explained by analogy with the classic non-deterministic algorithm (ND) [34]. The first rewriting system (system $\to_{\mathcal{L}}$ defined in Sec. 6.2) performs what we call left-factoring, analogous to generation of new equations from a common root term in ND, which is now algebraically understood as the distribution of composition over intersection. The following diagram illustrates ($\mathcal{E}$ stands for a multiset of equations):

$$\begin{array}{rcl} \{f(t_1,\ldots,t_n) = f(u_1,\ldots,u_n), \mathcal{E}\} & \Rightarrow & \{t_1 = u_1, \ldots, t_n = u_n, \mathcal{E}\} \\ f_i^n R \cap f_i^n S & \to_{\mathcal{L}} & f_i^n (R \cap S) \end{array}$$

The other step in ND, equation elimination, is now algebraically understood as constraint propagation (system $\to_{\mathcal{R}}$ defined Sec. 6.4):

$$\begin{array}{rcll} \{x = t, \mathcal{E}\} & \Rightarrow & \{\mathcal{E}[t/x]\} & \text{when } x \notin t \\ F(P_i \cap R) \cap G(P_i \cap S) & \to_{\mathcal{R}} & F(P_i \cap R \cap S) \cap G(P_i \cap S \cap R) & \end{array}$$

System $\to_{\mathcal{R}}$ is conditional on what we call *functorial compatibility*, a novel way of performing occurs checks which was motivated by $\mathsf{RelS}\Sigma$'s axiom $f_i^n \cap id = \mathbf{0}$. The two rewriting systems are carefully composed with the help of a constraint-propagation one (system $\to_{\mathcal{S}}$ defined in Section 6.3) to guarantee termination.

6.1. Rewriting preliminaries

The representation of **U**-terms in our rewriting systems is given by the following term-forming operations: $\mathsf{c} : \mathcal{C}_\Sigma \to \mathbf{U}$, $\mathsf{t} : (\mathcal{F}_\Sigma \times \mathbb{N} \times \mathbb{N}) \to \mathbf{U}$, $\mathsf{P} : \mathbb{N} \to \mathbf{U}$, $\odot : (\mathbf{U}_1 \times \cdots \times \mathbf{U}_n) \to \mathbf{U}$, and $\cap : (\mathbf{U}_1 \times \cdots \times \mathbf{U}_n) \to \mathbf{U}$, with $n \geq 2$ and n-ary $\odot$ and $\cap$. In addition to the above ground representation, we define patterns of **U**-terms in a standard way using a set of variables. Let i, j, k etc, range over $\mathbb{N}$. Let a, b, c etc, range over $\mathcal{C}_\Sigma$. Let f, g, h etc, range over $\mathcal{F}_\Sigma$. Let R, S, T etc, range over **U**-terms. We write $(a, a)\mathbf{1}$ for $\mathsf{c}(a)$, write f_i^j for $\mathsf{t}(f, i, j)$, write P_i° for $\mathsf{P}(i)$, write $R_1 \ldots R_n$ for $\odot(R_1, \ldots, R_n)$, and write $R_1 \cap \cdots \cap R_n$ for $\cap(R_1, \ldots, R_n)$.

Rewrite rules are of the form $\rho : l \to r$ with ρ the rule's name, l and r patterns, and l not a variable. Conditional rewrite rules are of the form $\rho : l \to r \Leftarrow C$ belonging to type III CTRS [27] (see Def. 6.3). We write $\to^!$ for the normalization relation derived from a terminating and confluent (convergent) relation $\to$. We write $\circ$ for composition of rewriting relations. We use $\equiv$ for syntactic identity (modulo AC).

We use A and AC rewriting for $\odot$ and $\cap$. More precisely, we define the equational theories for relational terms $A_\odot = \{R(ST) = (RS)T\}$ and $AC_\cap = \{R \cap (S \cap T) = (T \cap S) \cap T,\ R \cap S = S \cap R\}$. The AC rewriting used is described in [13, p577–581]: associative term formers are flattened and rewrite rules are extended with dummy variables to take into account the arity of term-forming operations. Matching efficiency can be improved by using ordered rewriting.

A rewrite rule $\rho : l \to r$ matches a term t iff $l\sigma = t$ modulo $A_\odot$ and $AC_\cap$. A sequence of integers p is called a position. $f(t_1, \ldots, t_n)_{|\langle i,l \rangle} \equiv t_{i|l}$, given $1 \leq i \leq n$ and $t_{|\langle\rangle} = t$. We allow subterm matching: if there exists a position p such that $l\sigma = t_{|p}$ (modulo previous AC), then t reduces to $t\{t_{|p} \mapsto r\sigma\}$. Importantly, we also allow matching over *functorial variables* which represent the largest composition of f_i^n terms.[1] We use F, G, H etc, for functorial variables. The expression $\mathit{length}(F)$ delivers the length of functorial variable F. For example, the term $r_1^2 s_2^2((a,a)\mathbf{1} \cap (b,b)\mathbf{1} \cap t_1^1 P_1^\circ)$ matches $F(R \cap GS)$ with $\sigma = \{F \mapsto r_1^2 s_2^2,\ R \mapsto (a,a)\mathbf{1} \cap (b,b)\mathbf{1},\ G \mapsto t_1^1,\ S \mapsto P_1^\circ\}$. Matching over functorial variables is a sort of specialized list-matching. Note that such a variable always matches the largest sequence possible.

6.2. Left-factoring rewriting system

Definition 6.1. The rewriting system $\mathcal{L}$ consists of the set of rewrite rules (where i ranges over all indices and f over all function symbols):

$$f_i^n R \cap f_i^n S \quad \to_{\mathcal{L}} \quad f_i^n (R \cap S)$$

Lemma 6.1. $\mathcal{L}$ is sound.

Proof. Soundness is a consequence of the following equation:

$$RS \cap RT = R(S \cap T) \tag{2}$$

which holds in **DRA** when R is functional (which is the case for every f_i^n):

$$R(S \cap T) \supseteq_{\textbf{[by } id \supseteq R^\circ R\textbf{]}} R(S \cap R^\circ RT) \supseteq_{\textbf{[by modular law]}} RS \cap RT$$

Conversely, $RS \cap RT \supseteq R(S \cap T)$ by **DRA**. □

Lemma 6.2. $\mathcal{L}$ is terminating and confluent.

Proof. To prove termination it suffices to give a lexicographic path ordering on terms [12]. The ordering is $\cap \succ \odot$. The system has no critical pairs, so it is locally confluent, local confluence plus termination implies confluence [28]. □

[1] We use "functorial" in connection with function symbols (term formers), not with functors in category theory.

6.3. Split-rewriting system

Definition 6.2. The rewriting system $\mathcal{S}$ consists of the rewrite rules:

$$F(R \cap G(P_i^\circ \cap S)) \to_{\mathcal{S}} FR \cap FG(P_i^\circ \cap S)$$

Lemma 6.3. $\mathcal{S}$ is sound, terminating and confluent.

Proof. Soundness and closure properties are immediate from (2). Termination is proven giving the lexicographic path ordering $\odot \succ \cap$ and confluence follows from the fact that $\to_{\mathcal{S}_i}$ is terminating and locally confluent. □

We will also use a parametrized version of $\mathcal{S}$, written $\mathcal{S}_i$ where i is fixed.

6.4. Constraint-propagation rewriting system

The purpose of this system is to propagate constraints over P_i° terms. Constraint propagation has two main technical difficulties. First, addressing occurs check to avoid infinite rewriting. Second, propagating constraints before checking for term clashes. Both difficulties can be addressed by introducing a decidable notion of functorial compatibility:

Definition 6.3. The convergent rewriting relation $\to_\Delta$ is defined as:

$$(f_i^n)^\circ f_i^n \to_\Delta id \qquad (f_i^n)^\circ g_j^m \to_\Delta \mathbf{0} \qquad (f_i^n)^\circ f_j^n \to_\Delta \mathbf{1} \qquad id f_i^n \to_\Delta f_i^n$$
$$(f_i^n)^\circ \mathbf{1} \to_\Delta \mathbf{1} \qquad \mathbf{1} f_i^n \to_\Delta \mathbf{1} \qquad (f_i^n)^\circ \mathbf{0} \to_\Delta \mathbf{0} \qquad \mathbf{0} f_i^n \to_\Delta \mathbf{0}$$

The above convergent rewriting relation gives rise to their associated function:

Definition 6.4 (Functorial delta). Given functorials F and G, we define $\Delta(F, G)$ as follows:

$$\Delta(F,G) = S, \qquad F^\circ G \to_\Delta^! S \qquad \text{if } length(G) \geq length(F)$$
$$\Delta(F,G) = S, \qquad G^\circ F \to_\Delta^! S \qquad \text{if } length(G) < length(F)$$

Lemma 6.4.

$$\Delta(F,G) = \begin{cases} \mathbf{0} & \text{if } [\![F \cap G]\!] = \emptyset \\ id & \text{if } F \equiv G. \\ S & \text{if } G \equiv FS \\ S & \text{if } F \equiv GS \\ \mathbf{1} & \text{otherwise.} \end{cases}$$

Proof. By induction on functorial terms. □

Remark 6.1. We may understand functorials as pointers to a position in a term. From this standpoint, Δ handles "pointer interference". That is, whether the functorials "point" to the same, to different but compatible, or to incompatible positions in a term. The delicate case is when one functorial points to a sub-position of another.

Definition 6.5 (Syntactic difference). The syntactic difference $\Theta(R_1 \cap \cdots \cap R_m, S_1 \cap \cdots \cap S_n)$ between two arbitrary-length intersection of terms is defined as the term $\bigcap_{i \in D} S_i$ such that $(\forall j \in \{1..m\}.R_j \not\equiv S_i) \iff i \in D$. Abusing notation, we use $\{\}$ for the empty intersection.

Lemma 6.5. $R \cap \Theta(R, S) \equiv S \cap \Theta(S, R)$

Proof. By induction on the length of R. □

Recall that $\equiv$ is syntactic identity modulo AC (Section 6.1). Should we use the idempotency axiom $A \cap A \equiv A$ to check equality modulo ACI, we would get an extended version of Lemma 6.5:

$$R \cap S \equiv R \cap \Theta(R, S) \equiv S \cap \Theta(S, R) \equiv S \cap R$$

Corollary 6.1. For any term R, $\Theta(R, R) = \{\}$.

Definition 6.6. The rewriting system $\mathcal{R}$ consists of the rewrite rules:

$$\begin{array}{ll} R0: & P_i^\circ \cap F(P_i^\circ \cap S) \to_{\mathcal{R}} \mathbf{0} \\ R1: & F(P_i^\circ \cap R) \cap G(P_i^\circ \cap S) \to_{\mathcal{R}} \mathbf{0} \quad \Leftarrow \Delta(F,G) = \mathbf{0} \vee \Delta(F,G) = f_i^n \ldots g_j^m \\ R2: & F(P_i^\circ \cap R) \cap G(P_i^\circ \cap S) \to_{\mathcal{R}} F(P_i^\circ \cap R \cap \Theta(R,S)) \cap G(P_i^\circ \cap S \cap \Theta(S,R)) \\ & \quad \Leftarrow (\Theta(R,S) \neq \{\} \vee \Theta(S,R) \neq \{\}) \wedge (\Delta(F,G) = \mathbf{1} \vee \Delta(F,G) = id) \end{array}$$

Notice Θ helps us deal conveniently with the equivalence of terms $R \cap S$ and $S \cap R$. We will also use a parametrized version of $\mathcal{R}$, written $\mathcal{R}_i$, where i is fixed.

Lemma 6.6. $\mathcal{R}$ is sound.

Proof. $R0$ is sound beacuse every term matching the left hand side can be factored into a term of the form $id \cap f_i^n \ldots g_j^m$ using $RP_i^\circ \cap SP_i^\circ = (R \cap S)P_i^\circ$ which is a version of Eq. (3) below. $R1$ is sound for two reasons:

1. If $\Delta(F, G) = \mathbf{0}$ then F and G are incompatible and the left hand side rewrites to $\mathbf{0}$ by left-factoring.
2. If $\Delta(F, G) = f_i^n \ldots g_j^m$ then there is a common prefix F' of F and G such that $F \cap G = F'(id \cap f_i^n \ldots g_j^m)$ and the right-hand-side rewrites to $\mathbf{0}$ by RelΣ's occurs-check axiom.

That $R2$ is sound follows from the equation:

$$F(P_i^\circ \cap R) \cap GP_i^\circ = F(P_i^\circ \cap R) \cap G(P_i^\circ \cap R) \tag{3}$$

Recall that a relation is injective when $RR^\circ \subseteq id$. Recall Lemma 6.1 which states that $RT \cap ST = (R \cap S)T$ for T injective. Using these facts we prove the $\subseteq$ direction:

$$\begin{array}{l} F(P_i^\circ \cap R) \cap GP_i^\circ =_{\textbf{[left+right factoring]}} (F \cap G)P_i^\circ \cap FR \subseteq_{\textbf{[modular law]}} \\ (F \cap G)((F^\circ \cap G^\circ)FR \cap P_i^\circ) \subseteq_{\textbf{[by } (F^\circ \cap G^\circ)F \subseteq F^\circ F = id\textbf{]}} (F \cap G)(R \cap P_i^\circ) \\ =_{\textbf{[injectivity of } (R \cap P_i^\circ)\textbf{]}} F(R \cap P_i^\circ) \cap G(R \cap P_i^\circ) \end{array}$$

The $\supseteq$ direction is easier:

$$F(P_i^\circ \cap R) \cap GP_i^\circ \;\; \supseteq_{\textbf{[monotonicity of } \cap\textbf{]}} F(P_i^\circ \cap R) \cap G(P_i^\circ \cap R)$$

□

The following example illustrates why the compatibility check is needed to ensure termination. Take the term $P_1^\circ \cap f_1^1(P_1^\circ \cap R)$. If we propagate restrictions by orienting (3) we get an infinite rewrite: $P_1^\circ \cap f_1^1(P_1^\circ \cap R) \to P_1^\circ \cap f_1^1(R \cap P_1^\circ \cap f_1^1(P_1^\circ \cap R)) \to P_1^\circ \cap f_1^1(R \cap P_1^\circ \cap f_1^1(P_1^\circ \cap R \cap f_1^1(P_1^\circ \cap R))) \ldots$

Definition 6.7. Let $\succ_C$ be the order relation:

$$F(P_i^\circ \cap R) \cap G(P_i^\circ \cap S) \;\; \succ_C \;\; F(P_i^\circ \cap R \cap \Theta(R,S)) \cap G(P_i^\circ \cap S \cap \Theta(S,R))$$

where $\Theta(R,S) \neq \{\}$ and $\Theta(S,R) \neq \{\}$.

Informally, think of $\succ_C$ being defined over the measure "number of different elements of R and S".

We prove $\succ_C$ is well-founded using Lemma 6.5.

Lemma 6.7. $\succ_C$ is well-founded.

Proof. $\succ_C$ has bounded depth, with no chain of length longer than two. Suppose $\succ_C$ had a chain of length three:

$$\begin{array}{ll} F(P_i^\circ \cap R) \cap G(P_i^\circ \cap S) & \succ_C \\ F(P_i^\circ \cap R \cap \Theta(R,S)) \cap G(P_i^\circ \cap S \cap \Theta(S,R)) & \succ_C \\ F(P_i^\circ \cap R \cap \Theta(R,S) \cap \Theta(R \cap \Theta(R,S), S \cap \Theta(S,R))) \cap & \\ G(P_i^\circ \cap S \cap \Theta(S,R) \cap \Theta(S \cap \Theta(S,R), R \cap \Theta(R,S))) & \end{array}$$

and note that in order for the chain to exist $\Theta(R \cap \Theta(R,S), S \cap \Theta(S,R)) \neq \{\}$. But using Lemma 6.5, $R \cap \Theta(R,S) \equiv S \cap \Theta(S,R)$, so by Corollary 6.1, $\Theta(R \cap \Theta(R,S), S \cap \Theta(S,R)) = \{\}$, contradicting the existence of the chain. □

Lemma 6.8. $\mathcal{R}$ terminates.

Proof. Rules $R0$ and $R1$ rewrite to $\mathbf{0}$. For rule $R2$ we define a well-founded order that contains the relation induced by $R2$. Notice that the finiteness of the terms involved in the rewriting ensures that the transformation for handling the n-ary term former $\cap$ doesn't affect termination. □

Lemma 6.9. $\mathcal{R}$ is confluent (modulo commutativity).

Proof. We give the same lexicographic path ordering as Lemma 6.2 and $\mathcal{R}$ has no overlapping rules. □

6.5. The algorithm

Unfortunately, a naïve application of the previous rewriting rules does not necessarily reach a solved form. Take for example the term $r_1^2(P_1^\circ \cap s_1^1(P_2^\circ \cap (a,a)\mathbf{1})) \cap r_2^2(P_2^\circ \cap s_1^1(P_1^\circ \cap (b,b)\mathbf{1}))$. If we apply the rewriting strategy $\to_{\mathcal{S}}^! \circ \to_{\mathcal{R}}^! \circ \to_{\mathcal{L}}^!$ we do not reach a solved form because $\to_{\mathcal{S}}^!$ destroys constraints over P_1°, impeding the constraint propagation step to work properly. One solution is to complete the constraints for each P_i° one at a time.

Lemma 6.10. Given a rewriting $t \to_{\mathcal{S}_i}^! t'$, every P_i° in t' occurs at *top level relative to functorials*, i.e., t' has the form:

$$P_i^\circ \cap F(P_i^\circ \cap R) \cap \cdots \cap G(P_i^\circ \cap S) \cap \ldots$$

Proof. If the P_i° term occurs deeper in the term, t' has the form:

$$F(R \cap G(S \cap \cdots H(P_i^\circ \cap T))) \cap \ldots$$

which is not a normal form for $\to_{\mathcal{S}_i}^!$. □

Definition 6.8 (Individual constraint propagation). The rewrite relation $\to_{\mathcal{U}_i}$ parametrized on i is defined as:

$$\to_{\mathcal{L}}^! \circ \to_{\mathcal{S}_i}^! \circ \to_{\mathcal{R}_i}^! \circ \to_{\mathcal{L}}^!$$

Lemma 6.11. $\Xi_i(t')$ holds for every term t and reduction $t \to_{\mathcal{U}_i} t'$.

Proof. Assume a term t exists such that $t \to_{\mathcal{U}_i} t'$ and $\neg\Xi_i(t')$. There are subterms of t' of the form $P_i^\circ \cap R$ and $P_i^\circ \cap S$, with $\Theta(R,S) \neq \{\}$ or $\Theta(S,R) \neq \{\}$. Let $t \to_{\mathcal{L}}^! u$, then no subterm of the form $FR \cap FS$ exists in u. Let $u \to_{\mathcal{S}_i}^! v$, then every P_i° in v is of the form described in Lemma 6.10. Let $v \to_{\mathcal{R}_i}^! w$, then $\Xi_i(w)$ because w is a normal form of $\to_{\mathcal{R}_i}$ and $P_i^\circ \cap R$ and $P_i^\circ \cap S$ with $\Theta(R,S) \neq \{\}$ and $\Theta(S,R) \neq \{\}$ cannot occur at the top-level, and every P_i° is at the top level. Let finally $w \to_{\mathcal{L}}^! t'$. For $\to_{\mathcal{L}}^!$ to break P-constraint completeness, w must have a term of the form $FP_i^\circ \cap FR$ such that after $\to_{\mathcal{L}}^!$, R becomes a new constraint on P_i°. However this cannot happen, for u had no such terms and v had only terms of the form $F(P_i^\circ \cap R) \cap FG(P_i^\circ \cap R)$, which are rewritten to $\mathbf{0}$ by $R1$. □

Lemma 6.12. Given t such that $\Xi_i(t)$ holds then $t \to_{\mathcal{S}_i}^! t'$ and $\Xi_i(t')$.

Proof. Follows from $\to_{\mathcal{S}_i}$ rules because if $\Xi(t)$ then no term of the form $F(P_i^\circ \cap GR)$ with P_i° in R can occur in t. Such occurrence is the necessary condition for $\to_{\mathcal{S}_i}^!$ to destroy P-constraint completeness. □

Lemma 6.13. Given t such that $\Xi_i(t)$ holds then $t \to_{\mathcal{R}_i}^! t$.

Proof. Follows from the definition of $\to_{\mathcal{R}_i}$. Note that if $R0$ or $R1$ are applicable for a term t, this easily implies $\neg\Xi(t)$. □

Lemma 6.14. $\to_{\mathcal{U}_i}$ reaches a fixed point, in other words, given a term t then $t \to_{\mathcal{U}_i} t' \to_{\mathcal{U}_i} t''$ and $t' = t''$.

$dep(P_j^\circ)$	$=$	$\emptyset$	$dep'(P_j^\circ)$	$=$	$\{j\}$
$dep(R \cap S)$	$=$	$dep(R) \cup dep(S)$	$dep'(R \cap S)$	$=$	$dep'(R) \cup dep'(S)$
$dep(f_i^n R)$	$=$	$dep'(R)$	$dep'(f_i^n R)$	$=$	$dep'(R)$

Figure 3: Definition of $dep : \mathbf{U} \to \mathcal{P}(\mathbb{N})$

Proof. $\Xi_i(t')$ holds by Lemma 6.11. In the second $\to_{\mathcal{U}_i}$ step, $t' \to^!_{\mathcal{L}} t'$ holds. By Lemma 6.12, $t' \to^!_{\mathcal{S}_i} u$ and $\Xi_i(u)$. By Lemma 6.13, we have $u \to^!_{\mathcal{R}_i} u$. Finally, we need to prove $t' \to^!_{\mathcal{S}_i} u \to^!_{\mathcal{L}} t'$. This is proven by the fact that $\to^!_{\mathcal{L}}$ *undoes* everything $\to^!_{\mathcal{S}_i}$ does on already factorized terms, which is the case of t'. Given rewriting rules $F(R \cap G(P_i^\circ \cap S)) \to_{\mathcal{S}_i} FR \cap FG(P_i^\circ \cap S)$ and $FT \cap FU \to_{\mathcal{L}} F(T \cap U)$, their composition happens with substitution $\{T \mapsto R,\ U \mapsto G(P_i^\circ \cap S)\}$, resulting in the rewriting rule $F(R \cap G(P_i^\circ \cap S)) \to_{\mathcal{L} \circ \mathcal{S}_i} F(R \cap G(P_i^\circ \cap S))$ which is the identity. □

Definition 6.9 (P-dependency). Given a term $P_i^\circ \cap R$, we say i P-depends on j if $j \in dep(R)$ where $dep : \mathbf{U} \to \mathcal{P}(\mathbb{N})$ is defined in Fig. 3. We can build the P-dependency for a term t taking all its subterms in the form $P_i^\circ \cap R$.

Lemma 6.15 (Constraint destruction)**.** Given $\Xi_j(t)$, $t \to_{\mathcal{U}_i} t'$ can make $\neg\Xi_j(t')$ iff j P-depends on i and $i \neq j$.

Proof. The only way $\to_{\mathcal{U}_i}$ can add a new constraint to a P_j° by constraint propagation is if the term is in the form $P_j^\circ \cap FR$ and P_i° occurs in R which is precisely the definition of P-dependency. □

Lemma 6.16 (Occurs check)**.** If i P-depends on i in a term t then $t \to_{\mathcal{U}_i} \mathbf{0}$.

Proof. Such P-dependency means $P_i^\circ \cap FR$ and P_i° occurs in R which gets rewritten to $\mathbf{0}$ by $\to^!_{\mathcal{R}_i}$. □

Definition 6.10 (Solved form algorithm). Given a term $t \in \mathbf{U}$, containing P_i° terms with $i \in \{1 \dots n\}$ and $n \geq 1$, we define the rewriting relation $\to_{\mathcal{U}_{\downarrow n}}$ as $\to_{\mathcal{U}_1} \circ \cdots \circ \to_{\mathcal{U}_n}$, that is:

$$\to^!_{\mathcal{L}} \circ \to^!_{\mathcal{S}_1} \circ \to^!_{\mathcal{R}_1} \circ \to^!_{\mathcal{L}} \circ \to^!_{\mathcal{S}_2} \circ \to^!_{\mathcal{R}_2} \circ \to^!_{\mathcal{L}} \circ \cdots \circ \to^!_{\mathcal{S}_n} \circ \to^!_{\mathcal{R}_n} \circ \to^!_{\mathcal{L}}$$

For $n = 0$, there are no P_i° in t and we define $\to_{\mathcal{U}_{\downarrow 0}}$ as $\to^!_{\mathcal{L}}$.

Lemma 6.17. There is a finite k such that $t \to^k_{\mathcal{U}_{\downarrow n}} t'$ and $\Xi(t')$.

Proof. The case $n = 0$ follows from Lemma 6.2 with $k = 1$. The case $n = 1$ follows from Lemma 6.14 with $k = 1$. In the case $n > 1$, for a step $u \to_{\mathcal{U}_i} u'$ then $\Xi_i(u')$ by Lemma 6.11. However, by Lemma 6.15 such a step can make $\neg\Xi_j(u')$ hold iff j P-depends on i. Suppose the P-dependency graph of t is acyclic. Then there is a set of terminal edges E such that forall $l \in E$, Ξ_l holds after $\to_{\mathcal{U}_l}$ and no other step $\to_{\mathcal{U}_i}$ can make Ξ_l false, for they depend on no j. Once the term is Ξ_l for every $l \in E$, E can be removed from the graph. The

process can be repeated with the new set E' of terminal edges a finite number of times, for the graph is acyclic and the number of edges is finite. Finally if the P-dependency graph of the original term is cyclic, then by Lemma 6.16 the term would get rewritten to $\mathbf{0}$ in the $\to_{\mathcal{U}_i}$ iteration corresponding to any i on the cycle. Thus when the process ends, $\Xi_l(t')$ for all $l \in \{1 \dots n\}$ which is equivalent to $\Xi(t')$. □

Lemma 6.18. $\to_{\mathcal{U}_{\downarrow n}}$ reaches a fixed point.

Proof. The proof is similar to the one in Lemma 6.14, but using Lemma 6.17, for once $\Xi(t)$ holds, $\to_{\mathcal{U}_{\downarrow n}}$ application does not modify t. □

Lemma 6.19. The fixed point for $\to_{\mathcal{U}_{\downarrow n}}$ is in solved form.

Proof. We check by induction that no unsolved term can be a fixed point:

- $f_i^n R$ is unsolved when R is unsolved, this means one of the cases apply for R.
- $P_i^\circ \cap f_j^n R$, if $P_i^\circ \notin R$ and R not in solved form: If R is in unsolved form then one of the cases below applies. Otherwise, if P_i° is a subterm of R when $\to^!_{\mathcal{R}_i}$ fires, it rewrites to $\mathbf{0}$.
- $f_i^n R \cap f_i^n S$: Terms of this form match the left side of $\to_{\mathcal{L}}$.
- $f_i^n R \cap f_j^n S$, we have two cases:
  - $\Xi(R \cap S)$ doesn't hold: Lemma 6.18 implies that $\to_{\mathcal{U}_n}$ reaches a fixed point for t precisely when $\Xi(t)$ holds, so $\neg\Xi(R \cap S)$ is a contradiction.
  - Any of R, S are in unsolved form: for the term in unsolved form one of the cases of the definition of unsolved applies.
- $R_1 \cap \dots \cap R_n$, if some pair $R_i \cap R_j, i \neq j$ is in unsolved form: If the pair R_i, R_j is in unsolved form one of the cases above apply.

□

Definition 6.11 (Relational unification). We define the relation between **U**-terms with maximum index n, denoted $t \to_{\{UNIF,n\}} t'$, as follows: If $t \to^!_{\mathcal{U}_{\downarrow n}} s$ and s is not valid then $t' = \mathbf{0}$; otherwise, $t' = s$.

Theorem 6.1. Terms t_1, t_2 with n different variables are not unifiable iff $K(t_1) \cap K(t_2) \to_{\{UNIF,n\}} \mathbf{0}$.

Proof. By Lemma 6.19 we know that $\to^!_{\mathcal{U}_{\downarrow n}}$ brings any **U**-term to solved form. As we have a complete decision procedure for solved terms by Lemma 5.1, we can decide if $K(t_1) \cap K(t_2)$ is $\mathbf{0}$, which means by Eq. 1 that t_1, t_2 are not unifiable. □

Definition 6.12 (Relational unifiers). We say a term t in solved form has a unifier R for i if $P_i^\circ \cap R$ is a subterm of t.

For example, suppose we have the term $P_1^\circ \cap (a, a)\mathbf{1} \cap P_2^\circ$. This means in $\mathcal{T}_\Sigma(\mathcal{X})$ that $x_1 = x_2 = a$.

7. Example

We will use the example given in [34]:

$$t_1 = f(g(h(a, x_5), x_2), x_1, h(a, x_4), x_4)$$
$$t_2 = f(x_1, g(x_2, x_3), x_2, b)$$

First, we use K-translation to translate the term into a suitable relational form:

$$K(t_1) = f_1^4(g_1^2(h_1^2(a,a) \cap h_2^2 P_5^\circ) \cap g_2^2 P_2^\circ) \cap f_2^4 P_1^\circ \cap f_3^4(h_1^2(a,a) \cap h_2^2 P_4^\circ) \cap f_4^4 P_4^\circ$$
$$K(t_2) = f_1^4 P_1^\circ \cap f_2^4(g_1^2 P_2^\circ \cap g_2^2 P_3^\circ) \cap f_3^4 P_2^\circ \cap f_4^4(b,b)\mathbf{1}$$

So $K(t_1) \cap K(t_2)$ is:

$$\begin{array}{lll} f_1^4(g_1^2(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_5^\circ) \cap g_2^2 P_2^\circ) & \cap\ f_1^4 P_1^\circ & \cap \\ f_2^4 P_1^\circ & \cap\ f_2^4(g_1^2 P_2^\circ \cap g_2^2 P_3^\circ) & \cap \\ f_3^4(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_4^\circ) & \cap\ f_3^4 P_2^\circ & \cap \\ f_4^4 P_4^\circ & \cap\ f_4^4(b,b)\mathbf{1} & \end{array}$$

Now we apply our rewriting strategy. For the sake of brevity, we will rewrite variables in the optimal order, which is easily calculable after factoring. We first factor the term:

$$\begin{array}{ll} f_1^4(g_1^2(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_5^\circ) \cap g_2^2 P_2^\circ \cap P_1^\circ) & \cap \\ f_2^4(P_1^\circ \cap g_1^2 P_2^\circ \cap g_2^2 P_3^\circ) & \cap \\ f_3^4(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_4^\circ \cap P_2^\circ) & \cap \\ f_4^4(P_4^\circ \cap (b,b)\mathbf{1}) & \end{array}$$

One of the orders induced by Γ is $1, 3, 2, 4, 5$. We will start with variable P_5°. The first step is to split on 5. We apply $\rightarrow_{\mathcal{S}_5}$:

$$\begin{array}{ll} f_1^4(g_1^2(h_1^2(a,a)\mathbf{1}) \cap g_2^2 P_2^\circ \cap P_1^\circ) \cap f_1^4 g_1^2 h_2^2 P_5^\circ & \cap \\ f_2^4(P_1^\circ \cap g_1^2 P_2^\circ \cap g_2^2 P_3^\circ) & \cap \\ f_3^4(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_4^\circ \cap P_2^\circ) & \cap \\ f_4^4(P_4^\circ \cap (b,b)\mathbf{1}) & \end{array}$$

No constraint can be propagated, so we come back and split on 4:

$$\begin{array}{ll} f_1^4(g_1^2(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_5^\circ) \cap g_2^2 P_2^\circ \cap P_1^\circ) & \cap \\ f_2^4(P_1^\circ \cap g_1^2 P_2^\circ \cap g_2^2 P_3^\circ) & \cap \\ f_3^4(h_1^2(a,a)\mathbf{1} \cap P_2^\circ) \cap f_3^4 h_2^2 P_4^\circ & \cap \\ f_4^4(P_4^\circ \cap (b,b)\mathbf{1}) & \end{array}$$

At this point some propagation takes place:

$$\begin{array}{ll} f_3^4 h_2^2 P_4^\circ \cap f_4^4(P_4^\circ \cap (b,b)\mathbf{1}) & \rightarrow_{\mathcal{R}} \\ f_3^4 h_2^2(P_4^\circ \cap (b,b)\mathbf{1}) \cap f_4^4(P_4^\circ \cap (b,b)\mathbf{1}) & \end{array}$$

The full term is now (after factoring again):

$$\begin{array}{ll} f_1^4(g_1^2(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_5^\circ) \cap g_2^2 P_2^\circ \cap P_1^\circ) & \cap \\ f_2^4(P_1^\circ \cap g_1^2 P_2^\circ \cap g_2^2 P_3^\circ) & \cap \\ f_3^4(h_1^2(a,a)\mathbf{1} \cap P_2^\circ \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) & \cap \\ f_4^4(P_4^\circ \cap (b,b)\mathbf{1}) & \end{array}$$

We factor for 2:

$$\begin{array}{ll} f_1^4 g_2^2 P_2^\circ & \cap \\ f_2^4 g_1^2 P_2^\circ & \cap \\ f_3^4(P_2^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) & \cdots \end{array}$$

and we get propagated the constraint $h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})$ to the other terms yielding a result:

$$\begin{array}{ll} f_1^4(g_1^2(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_5^\circ) \cap g_2^2(P_2^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) \cap P_1^\circ) & \cap \\ f_2^4(P_1^\circ \cap g_1^2(P_2^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) \cap g_2^2 P_3^\circ) & \cap \\ f_3^4(h_1^2(a,a)\mathbf{1} \cap P_2^\circ \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) & \cap \\ f_4^4(P_4^\circ \cap (b,b)\mathbf{1}) & \end{array}$$

For P_3° there is only one occurrence, so we go to propagate on P_1°.

$$\begin{array}{ll} f_1^4(P_1^\circ \cap g_1^2(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_5^\circ) \cap g_2^2(P_2^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1}))) & \cap \\ f_2^4(P_1^\circ \cap g_1^2(P_2^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) \cap g_2^2 P_3^\circ) & \cap \end{array}$$

which results in

$$\begin{array}{lll} f_1^4(P_1^\circ \cap & g_1^2(h_1^2(a,a)\mathbf{1} \cap h_2^2 P_5^\circ) \cap g_1^2(P_2^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) & \cap \\ & g_2^2(P_2^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1}) \cap g_2^2 P_3^\circ) & \end{array}$$

after factoring becomes:

$$\begin{array}{lll} f_1^4(P_1^\circ \cap & g_1^2(h_1^2(a,a)\mathbf{1} \cap P_2^\circ h_2^2(P_4^\circ \cap P_5^\circ \cap (b,b)\mathbf{1})) & \cap \\ & g_2^2(P_2^\circ \cap P_3^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) & \end{array}$$

The final term:

$$\begin{array}{ll} f_1^4(P_1^\circ \cap g_1^2(h_1^2(a,a)\mathbf{1} \cap P_2^\circ h_2^2(P_4^\circ \cap P_5^\circ \cap (b,b)\mathbf{1})) & \cap \\ \quad g_2^2(P_2^\circ \cap P_3^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) & \cap \\ f_2^4(P_1^\circ \cap g_1^2(h_1^2(a,a)\mathbf{1} \cap P_2^\circ h_2^2(P_4^\circ \cap P_5^\circ \cap (b,b)\mathbf{1})) & \cap \\ \quad g_2^2(P_2^\circ \cap P_3^\circ \cap h_1^2(a,a)\mathbf{1} \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) & \cap \\ f_3^4(h_1^2(a,a)\mathbf{1} \cap P_2^\circ \cap h_2^2(P_4^\circ \cap (b,b)\mathbf{1})) & \cap \\ f_4^4(P_4^\circ \cap (b,b)\mathbf{1}) & \end{array}$$

We checked the term is valid: we couldn't find an invalidating constraint. Now, we erase the P_i° terms to help see better what actually happened:

$$\begin{array}{ll} f_1^4(g_1^2(h_1^2(a,a)\mathbf{1} \cap h_2^2(b,b)\mathbf{1}) \cap g_2^2(h_1^2(a,a)\mathbf{1} \cap h_2^2(b,b)\mathbf{1})) & \cap \\ f_2^4(g_1^2(h_1^2(a,a)\mathbf{1} \cap h_2^2(b,b)\mathbf{1}) \cap g_2^2(h_1^2(a,a)\mathbf{1} \cap h_2^2(b,b)\mathbf{1})) & \cap \\ f_3^4(h_1^2(a,a)\mathbf{1} \cap h_2^2(b,b)\mathbf{1}) & \cap \\ f_4^4(b,b)\mathbf{1} & \end{array}$$

Enhanced visualization:

$$\begin{array}{ll} f_1(g_1(h_1\mathbf{a} \cap h_2\mathbf{b}) \cap g_2(h_1\mathbf{a} \cap h_2\mathbf{b})) & \cap \\ f_2(g_1(h_1\mathbf{a} \cap h_2\mathbf{b}) \cap g_2(h_1\mathbf{a} \cap h_2\mathbf{b})) & \cap \\ f_3(h_1\mathbf{a} \cap h_2\mathbf{b}) & \cap \\ f_4\mathbf{b} & \end{array}$$

If we worry about space we can borrow categorical notation to stretch the text:

$$f\langle g\langle h\langle \mathbf{a}, \mathbf{b}\rangle, h\langle \mathbf{a}, \mathbf{b}\rangle\rangle, g\langle h\langle \mathbf{a}, \mathbf{b}\rangle, h\langle \mathbf{a}, \mathbf{b}\rangle\rangle, h\langle \mathbf{a}, \mathbf{b}\rangle, \mathbf{b}\rangle$$

or indeed with $\delta : 1 \to 2, \delta(f) = \langle f, f\rangle$:

$$f\langle gh\langle \mathbf{a}, \mathbf{b}\rangle\delta^2, h\langle \mathbf{a}, \mathbf{b}\rangle, \mathbf{b}\rangle$$

which can be translated back to

$$f(g(h(a, b), h(a, b)), g(h(a, b), h(a, b)), h(a, b), b)$$

8. Related work

Unification. Unification was first proposed in [38] and axiomatized in [32, 33]. Classical algorithms for first-order unification are surveyed in [3], such as [34, 37] which define fast and well-understood algorithms targeted to implementors using imperative languages. Other interesting approaches to unification include categorical views [22, 40], unification for lambda terms [25, 15] and a remarkable one [14], which studies unification in the typed combinatorial paradigm [11]. Nominal unification [43] is an alternative approach to meta-theory formalization that uses nominal logic. C-expressions [5, 4] also provide a combinatorial unification algorithm, based on applicative terms instead of relations. Unification using explicit substitutions is handled in the higher-order case in [16].

In practical terms, our work is not different from these approaches because the result of the algorithm is a unifier.

However, our work is very different in its theoretical foundations and implications. In a sense, all the combinatorial approaches are based on applicative structures, where a term $f(x)$ is represented by the function f such that $f(x) = \mathsf{f}(\mathsf{x})$, so variables are implicit. The relational approach allows us to go a step further by *explicity* handling variables at the object level by means of projections. In fact, the relational term P_i can be effectively identified with the i^{th} variable. Because of this, some primitives that are difficult to define in other frameworks, such as the term-destructor function $f^i : \mathcal{T}_\Sigma \to (\mathcal{T}_{\Sigma 1} \times \cdots \times \mathcal{T}_{\Sigma i})$, are easy to define in ours.

The formalism chosen in this paper is based on binary relation algebras [41] and allegories [19], both of which capture predicates via relations that extend any given base of functions. Due to the presence of *quasiprojections* (Tarski) or *tabulations* (Freyd) we are able to represent predicates and terms of arbitrary arity. The addition of rewrite rules in our paper is completely new, and makes this axiomatization a full-blown programming language, with a variable-free notion of reduction.

Comparison with cylindric algebras. Tarski and Givant [41] discuss in depth the benefits of variable elimination via binary relation algebras, including their relation to cylindric algebras. (It is interesting to note that Nemeti deems both approaches complementary in his review of Tarski and Givant's book [36]). In our context, binary relation algebras provide a mathematically rigorous notion of compilation in which target code retains the declarative content of the source code and is extensible to constraints and many other logic programming formalisms. The use of binary relations, as compared to other relational formalisms in general and cylindric algebras in particular, greatly simplifies the rewrite rules. First, there is no need for indexed joins, i.e., relations obtained by "composing" two n-ary relations with respect to a particular component. Second, functions can be interpreted "as themselves", i.e., as single-valued relations.

Variable elimination and abstract syntax. Other efforts to eliminate variables in logic programming include the n-ary relation based combinatory work of [24] and [4] where a new polynomial algebra is developed for the purpose of eliminating variables. These results can be viewed as a special case of the research in abstract syntax aimed at axiomatizing the treatment of renaming, freshness, instantiation, and quantification as a mathematical theory in its own right. We refer the reader to [20, 21, 8] and to [9], the latter an excellent and detailed description of the field, its aims and its history. Finally, category theory itself can be viewed as a device to eliminate variables from mathematics. Categorical tools have been increasingly employed to formalize abstract syntax (see [18, 21, 19, 17, 2]).

Tabular allegories. Freyd shows in [19] that there is a natural canonical extension of a category to a relations-enriched category (an allegory) in which the original arrows are recoverable as functional relations. Our work shares a special relationship with Freyd's tabular allegories. An allegory is a category whose $HomSets$ are enriched with semi-lattice structure[19]. An allegory is tabular iff every relation $R : A \to B$ has a tabulation $C, f : C \to A, g : C \to B$ such that $R = f^{\circ}g$ and $(x; f = y; f) \wedge (x; g = y; g) \Rightarrow x = y$ (; denotes diagrammatic composition).

This informally means that we have an object C such that it uniquely determines all the pairs of elements belonging to the relation. As an example, fix the set $A = \{1, 2, 3\}$ and $B = \{t, f\}$, then the relation $R : B \to B = \{(t, u), (t, v), (v, t)\}$ is tabulated by $f(1) = t, f(2) = t, f(3) = v, g(1) = u, g(2) = v, g(3) = t$.

In our encoding, the left tabulation corresponds to the term structure and the right tabulation captures *equations among variables.* Instantiation of a variable is a change of the domain of the tabulations. As an example imagine the relational term $K(m(x_1, x_2, x_3)) \cap K(m(x_1, x_1, a))$. Then the domain C of the tabulations is isomorphic to the Herbrand domain and the tabulations f, g

would be defined (assume C is $\mathcal{H}_\Sigma$):

$$\begin{aligned} f(x) &= m(x,x,a) \\ g(x) &= \langle x,x,a\rangle \end{aligned}$$

In this setting, we can interpret our algorithm as a check for disjoint images of the tabulations.

9. Conclusions and future work

We have presented an algorithm for first-order unification using rewriting in variable-free relational algebra. The simple systems $\to_{\mathcal{L}}$ and $\to_{\mathcal{R}}$ suffice to decide unification for occurs-check-free pairs of terms. Function Δ and the split-rewriting system are introduced to deal with occurs check at the expense of losing simplicity. We can regain simplicity by using a more powerful notion of rewriting and matching, obtaining as a result an efficient algorithm largely in the spirit of [34]. Furthermore, a dual right-factoring version of the algorithm exists where constraints are accumulated over f_i^n terms and factorization happens for P_i° terms, using $FP_i^\circ \cap GP_i^\circ = (F \cap G)P_i^\circ$. We plan to relate this kind of duality with other duality/symmetry notions such as deep inference [23].

Substitution. In the relational world we have no variables and no substitution notion. This raises an interesting paradox: the heart of unification is the computation of a substitution acting on variables. How can we unify in such a setting? The answer is that substitution gets replaced by constraint propagation and occurs check translates to functorial compatibility in such a way that *pure* (conditional) rewriting is enough!

Abstract syntax and algebraic approach. Another advantage of performing unification in the relational setting is that variable elimination is one way of formalizing abstract syntax, effectively giving syntax, syntactic operations (renaming, substitution, etc.), and syntactic machinery (contexts, multi-equations, etc.) a rigorous axiomatization by coding them out of existence into a particular formalism and making them as declarative as the object language. Even properties are captured algebraically, as mentioned in the introduction. For example, invariance under substitution of ground terms is reflected in our framework by $K(t)^\circ K(t) = \mathbf{1}$ (the equation is true iff t is ground) and equality's congruence with respect to term forming ($f(t_1,\ldots,t_n) = f(u_1,\ldots,u_n)$ iff $t_1 = u_1 \wedge \cdots \wedge t_n = u_n$) is captured in **DRA** as $FS \cap FR = F(S \cap R)$, provided F is a mapping.

Extending the framework. Our framework can be seamlessly extended in order to formalize and decide other notions of unification. For instance, a common unification pattern found in logic programming is unification among *renamed apart* terms. Consider a restriction operator ν such that $\nu x_1.t = x_1$ is equivalent

to $t\sigma = x_1$, where σ is a renaming apart of x_1, and $\nu x_1.t = x_1$ is equivalent to $t = x_1$ iff x_1 does not occur in t.

This variable-restriction concept can be faithfully represented in our framework using the partial projection relation Q_i, which relates an open sequence with the ones where the ith element may be any term. (In [29] Q_i is understood as an existential quantifier.) Our framework is modified as follows. First, extend K with case $K(\nu x_i.t) = K(t)Q_i$. In words, the ith position of $K(\nu x_i.t)$'s codomain is free, whereas for $K(t)$ it contains the set of possible groundings for x_i. The new definition of K extends $\mathbf{U}$, so a new decision procedure is needed. It is defined in modular fashion by adding a new rewriting subsystem for Q_i elimination. Some rules of this system are $P_jQ_i \to P_j$ and $(f_i^n P_j \cap R)Q_i \to f_i^n P_j Q_i \cap RQ_i$.

Performing unification modulo additional theories is possible in our framework and we think it should not be difficult to derive the corresponding algorithm. As supporting evidence, see [29] on how to represent disunification problems with relations.

Categorical interpretation. Defining the full set of rules for the system in the preceding paragraph will result in a too complex rewriting system. We believe such a complexity is not inherent to our approach, but to the formalism used, in this case rewriting.

We could modify our rewriting procedure — adding more *structure* awareness to it — in order to avoid most of the cumbersome rewritings, like the splitting/factoring, where terms are split to be joined again. However we believe the algorithm's true nature will be revealed in a categorical interpretation.

A deeper algebraic study of the algorithm reveals some interesting properties. For instance, a "functorial" variable directly maps to an arrow in a category. In fact, normal forms are just a representation of the tabulation for the intersection of two relations.

Another example of a categorical construction is the "functorial compatibility" test, $F^\circ G$, which is just a tabulation for the compatibility relation.

Establishing a categorical semantics where the current algorithm can be compared with existing pullback construction algorithms is one of our goals. We hope categorical methods will help to generalize the algorithm.

Acknowledgements. We thank the anonymous referees for their insightful comments.

References

[1] Hassan Ait-Kaci. *Warren's Abstract Machine. A Tutorial Reconstruction.* The MIT Press, Cambridge, Massachusetts. London, England, 1991.

[2] Gianluca Amato, James Lipton, and Robert McGrail. On the algebraic structure of declarative programming languages. *Theoretical Computer Science*, 2009. Accepted for publication.

[3] Franz Baader and Wayne Snyder. Unification theory. In Robinson and Voronkov [39], pages 445–532.

[4] Marco Bellia and M. Eugenia Occhiuto. C-expressions: A variable-free calculus for equational logic programming. *Theor. Comput. Sci.*, 107(2):209–252, 1993.

[5] Marco Bellia and M. Eugenia Occhiuto. Lazy linear combinatorial unification. *Journal of Symbolic Computation*, 27(2):185–206, 1999.

[6] Paul Broome and James Lipton. Combinatory logic programming: computing in relation calculi. In *ILPS '94: Proceedings of the 1994 International Symposium on Logic programming*, pages 269–285, Cambridge, MA, USA, 1994. MIT Press.

[7] James Cheney. Toward a general theory of names: binding and scope. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Mechanized reasoning about languages with variable binding*, pages 33–40, New York, NY, USA, 2005. ACM.

[8] James Cheney and Christian Urban. alpha-prolog: A logic programming language with names, binding and a-equivalence. In Bart Demoen and Vladimir Lifschitz, editors, *ICLP*, volume 3132 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2004.

[9] James Robert Cheney. *Nominal Logic Programming.* PhD thesis, Cornell University, Ithaca, NY, USA, 2004. Advisor – Morrisett, Greg.

[10] Hubert Comon. Disunification: A survey. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 322–359, 1991.

[11] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1. North Holland, 1958. Second edition, 1968.

[12] Nachum Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982.

[13] Nachum Dershowitz and David A. Plaisted. Rewriting. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 535–610. Elsevier and MIT Press, 2001.

[14] Daniel J. Dougherty. Higher-order unification via combinators. *Theoretical Computer Science*, 114(2):273–298, 1993.

[15] Gilles Dowek. Higher-order unification and matching. In Robinson and Voronkov [39], pages 1009–1062.

[16] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher order unification via explicit substitutions. *Inf. Comput.*, 157(1-2):183–235, 2000.

[17] Stacy E. Finkelstein, Peter J. Freyd, and James Lipton. A new framework for declarative programming. *Theor. Comput. Sci.*, 300(1-3):91–160, 2003.

[18] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. *Logic in Computer Science, Symposium on*, 0:193, 1999.

[19] P. J. Freyd and A. Scedrov. *Categories, Allegories*. North Holland Publishing Company, 1991.

[20] Murdoch J. Gabbay, Samuel Rota Buló, and Andrea Marin. Denotations for functions in which variables are first-class denotational citizens — or, variables are data. Submitted to TLCA'2007, 2007.

[21] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax involving binders. In *14th Annual Symposium on Logic in Computer Science*, pages 214–224, Washington, DC, USA, 1999. IEEE Computer Society Press.

[22] Joseph Goguen. What is unification? A categorical view of substitution, equation and solution. In Maurice Nivat and Hassan Aït-Kaci, editors, *Resolution of Equations in Algebraic Structures, Volume 1: Algebraic Techniques*, pages 217–261. Academic, 1989.

[23] Alessio Guglielmi. A system of interaction and structure. *ACM Transactions on Computational Logic*, 8(1):1–64, 2007. `http://cs.bath.ac.uk/ag/p/SystIntStr.pdf`.

[24] Andreas Hamfelt and Jørgen Fischer Nilsson. Inductive synthesis of logic programs by composition of combinatory program schemes. In P. Flener, editor, *LOPSTR'98, 8th. International Workshop on Logic-Based Program Synthesis and Transformation*, volume 1559 of *Lecture Notes in Computer Science*, pages 143–158. Springer, 1998.

[25] Gérard P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975.

[26] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[27] J. W. Klop. Term rewriting systems. In Abramsky, Gabbay, and Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2. Clarendon Press, 1992.

[28] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 342–376. Springer, Berlin, Heidelberg, 1983.

[29] Jim Lipton and Emily Chapman. Some notes on logic programming with a relational machine. In Ali Jaoua, Peter Kempf, and Gunther Schmidt, editors, *Using Relational Methods in Computer Science*, Technical Report Nr. 1998-03, pages 1–34. Fakultät für Informatik, Universität der Bundeswehr München, July 1998.

[30] J. W. Lloyd. *Foundations of logic programming.* Springer-Verlag New York, Inc., New York, NY, USA, 1984.

[31] R.C. Lyndon. The representation of relational algebras. *Ann. of Math.*, Ser 2(51):707–729, 1950.

[32] Michael J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceedings, Third Annual Symposium on Logic in Computer Science, 5-8 July 1988, Edinburgh, Scotland, UK*, pages 348–357. IEEE Computer Society, 1988.

[33] A. I. Mal'cev. On the elementary theories of locally free universal algebras. *Soviet Math*, pages 768–771, 1961.

[34] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[35] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51(1-2):125–157, 1991.

[36] István Németi. Review of [41] (untitled). The Journal of Symbolic Logic, Vol 55. No. 1, pp 350–352, March 1990.

[37] Mike Paterson and Mark N. Wegman. Linear unification. *J. Comput. Syst. Sci.*, 16(2):158–167, 1978.

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

[39] John Alan Robinson and Andrei Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes).* Elsevier and MIT Press, 2001.

[40] D. E. Rydeheard and R. M. Burstall. A categorical unification algorithm. In *Proceedings of a tutorial and workshop on Category theory and computer programming*, pages 493–505, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[41] Alfred Tarski and Steven Givant. *A Formalization of Set Theory Without Variables*, volume 41 of *Colloquium Publications.* American Mathematical Society, Providence, Rhode Island, 1987.

[42] D. A. Turner. A new implementation technique for applicative languages. *Software – Practice and Experience*, 9:31 – 49, 1979.

[43] Christian Urban, Andrew M. Pitts, and Murdoch Gabbay. Nominal unification. In Matthias Baaz and Johann A. Makowsky, editors, *CSL*, volume 2803 of *Lecture Notes in Computer Science*, pages 513–527. Springer, 2003.