Functions-as-constructors Higher-order Unification

Tomer Libal and Dale Miller

Inria Saclay & LIX/École Polytechnique, Palaiseau, France

Abstract

Unification is a central operation in the construction of a range of computational logic systems based on first-order and higher-order logics. First-order unification has a number of properties that dominates the way it is incorporated within such systems. In particular, first-order unification is decidable, unary, and can be performed on untyped term structures. None of these three properties hold for full higher-order unification: unification is undecidable, unifiers can be incomparable, and term-level typing can dominate the search for unifiers. The so-called *pattern* subset of higher-order unification was designed to be a small extension to first-order unification that respected the basic laws governing λ-binding (the equalities of α, β, and η-conversion) but which also satisfied those three properties. While the pattern fragment of higher-order unification has been popular in various implemented systems and in various theoretical consideration, it is too weak for a number of applications. In this paper, we define an extension of pattern unification that is motivated by some existing applications and which satisfies these three properties. The main idea behind this extension is that the arguments to a higher-order, free variables can be more than just distinct bound variables: they can also be terms constructed from (sufficient numbers of) such variables using term constructor and where no argument is a subterm of any other argument. We show that this extension to pattern unification satisfies the three properties mentioned above.

1998 ACM Subject Classification F.4.1 Mathematical Logic: Mechanical theorem proving

Keywords and phrases higher-order unification; pattern unification

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Unification is the process of solving equality constraints by the computation of substitutions. This process is used in computational logic systems ranging from automated theorem provers, proof assistants, type inference systems, and logic programming. The first-order unification—that is, unification restricted to first-order terms—enjoys at least three important computational properties, namely, (1) decidability, (2) determinacy, and (3) type-freeness. These properties of unification shaped the way it can be used within computational logic systems. The first two of these properties ensures that unification—as a process—will either fail to find a unifier for a given set of disagreement pairs or will succeed and return the *most general unifier* that solves all those disagreement pairs. The notion of type-freeness simply means that unification can be done independently of the possible typing discipline that might be employed with terms. Thus, first-order unification can be performed on untyped first-order terms (as terms are usually considered in, say, Prolog). This property is important since it means that unification can be used with any typing discipline that might be adopted. Since typing is usually an open-ended design issue in many languages (consider, for example, higher-order types, subtypes, dependent types, parametric types, linear types, etc.), the type-freeness of unification makes it possible for it to be apply to a range of typing disciplines.

Of course, many syntactic objects are not most naturally considered as purely first-order terms: this is the case when that syntax contains bindings. Instead, many systems have

First International Conference on Formal Structures for Computation and Deduction.
Editors: D. Kesner and B. Pientka; pp. 1–15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

adopted the approach used by Church in his Simple Theory of Types [11] where terms and term equality comes directly from the λ-calculus. All binding operations—quantification in first-order formulas, function arguments in functional programs, local variables, etc.—can be represented using the sole binder of the λ-calculus. Early papers showed that second-order pattern matching could be used to support interesting program analysis and program transformation [18] and that a higher-order version of Prolog could be used to do more general manipulations of programs and formulas [21, 22]. Today, there is a rich collection of computational logic systems that have moved beyond first-order term unification and rely on some form of higher-order unification. These include the theorem provers TPS [3], Leo [7] and Satallax [10]; the proof assistants Isabelle [26], Coq [35], Matita [4], Minlog [31], Agda [9], Abella [5], and Beluga [29]; the logic programming languages λProlog [23] and Twelf [28]; and various application domains such as natural language processing [12].

The integration of full higher-order unification into computational logic systems is not as simple as it is in first-order systems since the three properties mentioned above do not hold. The unification of simply typed λ-terms is undecidable [15, 16] and there can be incomparable unifiers, implying that no most general unifiers exist in the general situation. Also, types matter a great deal in determining the search space of unifiers. For example, let i and j be primitive types, let a be a constant of type i, and let F and X be variables of type $\alpha \to i$ and α, respectively, where α is a type variable. Consider the unification problem $(F\ X) = a$. If we set α to j, then there is an mgu for this problem, namely $[F \mapsto \lambda w.a]$. If we set α to i, then there are two incomparable solutions, namely $[F \mapsto \lambda w.a]$ and $[F \mapsto \lambda w.w,\ X \mapsto a]$. If we set α to $i \to i$, then there is an infinite number of incomparable solutions: $[F \mapsto \lambda f.a]$ and, for each natural number n, $[F \mapsto \lambda f.f^n a,\ X \mapsto \lambda w.w]$. If higher order values for α are considered, the possibility of unifiers becomes dizzying.

For these reasons, the integration of unification for simply typed λ-terms into computational logic systems is complex: most such integration efforts attempt to accommodate the (pre-)unification search procedure of Huet [17].

Instead of moving from first-order unification to full higher-order unification, it is possible to move to an intermediate space of unification problems. Given that higher-order unification is undecidable, there is an infinite number of decidable classes that one could consider. The setting of *higher-order pattern unification* (proposed in [20] and called L_λ-unification there) could be seen as the weakest extension of first-order unification in which the equations of α, β, and η conversion hold. In this fragment, a free variable cannot be applied to general terms but only to bound variables that cannot appear free in eventual instantiations for the free variable. This restriction means that all forms of β-reduction encountered during unification are actually (α-equivalent) to the rule $(\lambda x.B)x = B$ (a conversion rule called β_0). Notice that in this setting, β-reduction reduces the size of terms: hence, unification takes on a much simpler nature. The unification problems that result retain all three properties we listed for first-order unification [20]. As a result, the integration of pattern unification into a prover is usually much simpler than incorporating all the search behavior implied by Huet's (pre-)unification procedure.

A somewhat surprising fact about pattern unification is that many computational logic systems actually need only this subset of higher-order unification in order to be "practically complete": that is, restricting unification to just this subset did not stop the bulk of specifications from being properly executed. For example, while both early implementations of λProlog and LF [24, 28] implemented full higher-order unification, the most recent versions of those languages implement only pattern unification [1, 30]. A design feature of both of those systems is to treat any unification problem that is not in the pattern fragment as a

suspended constraint: usually, subsequent substitutions will cause such delayed problems to convert into the pattern fragment. Also since pattern unification does not require typing information, this style of unification has been described for dependent and polymorphic types [27], product types [13, 14], and sum types [2].

Since pattern unification is a weak fragment of higher-order unification, it is natural to ask if it can be extended and still keep the same high-level properties. There have been extensions of pattern unification considered in the literature. The generalization (mentioned above) of pattern unification to patterns by Fettig and Löchner [14] and Duggan [13] allows for constructors denoting projections to be admitted in the scope of free functional variables. When attempting to encode the meta-theory of sequent calculus in which eigenvariables are seen as abstractions over sequents [19], a single bound variable was intended to be used as a list of bound (eigen)variables. Thus, in order to encode the sequent judgment $x_0, \ldots, x_n \vdash Cx_0 \ldots x_n$ (for $n \geq 0$ and all variables being of the same primitive type) one would instead use the simply typed term $\lambda l.C(\text{fst } l) \ldots (\text{fst}(\text{snd}^n l))$, where the environment abstraction l has type, say, evs, and fst and snd are constructors of type evs $\to i$ and evs $\to$ evs, respectively. Tiu showed how to lift pattern unification to this setting [33].

The Coq proof assistant allows for some limited forms of unification and many simple unification problems can appear that should be automatically solved. A typical such example is of the form $\lambda x.Y(gx) \doteq \lambda x.f(gx)$, where Y is a free variable of type $i \to i$ and f and g are constructors of the type $i \to i$. Clearly, this problem has the mgu $y \mapsto \lambda z.fz$ but it falls outside the pattern restriction. There are certain uses of Coq (for example, with the `bigop` library of SSReflect) which produces a number of non-pattern unification problems.[1]

Let us return to the definition of pattern unification problems. The restriction on occurrences of the free variable, say, M is that (1) it can be applied only to variables that cannot appear free in terms that are used to instantiate M and (2) that those arguments are distinct. Condition (1) essentially says that the arguments of M form a primitive pattern that allows one to form an abstraction to solve a unification problem. Thus, $M\ x\ y$ can equal, say, $(s\ x) + y$ simply by forming the abstraction $\lambda x \lambda y.(s\ x) + y$. Condition (2) implies that such abstracts are unique.

The examples of needing richer unification problems above illustrate that it is also natural to consider arguments built using variables *and* term constructors: that is, we should consider generalizing condition (1) above by allowing the application $\lambda l(M\ (\text{fst } l)\ (\text{fst } (\text{snd } l)))$. If this application is required to unify with a term of the form $\lambda l.t$ then all occurrence of l in t must occur in subterms of the form (fst l) or (fst (snd l)). In that case, forming an abstraction of t by replacing all occurrences of (fst l) and (fst (snd l)) with separate bound variables gives a solution to this unification problem. To guarantee uniqueness of such solutions, we shall also generalize condition (2) so that the arguments of M cannot be subterms of each other.

Many of the examples leading to this generalization of pattern unification arise in situations where operators (such as fst and snd) are really functions and *not* constructors: the intended meaning of those two operators are as functions that maps lists to either their first element or to their tail. When they arise in unification problems, however, we can only expect to treat them as constructors (ie, as injective functions). Thus, we shall name this extended pattern unification as *function-as-constructor* (pattern) unification, or just FCU for short.

The rest of this paper is structured as follows. We cover the basic concepts related to higher-order unification in Section 2. The class of unification problems addressed in this paper, the *functions-as-constructor* class, is defined in Section 3 as is a unification algorithm

[1] Personal communication with Enrico Tassi.

for that class. We prove the correctness of that algorithm in Section 4. We conclude in Section 5.

2 Preliminaries

2.1 λ-Calculus

In this section we will present the logical language that will be used throughout the paper. The language is a version of Church's simple theory of types [11] with an η-conversion rule as presented in [6] and [32] and with implicit α-conversions. Unless stated otherwise, all terms are implicitly converted into β-normal and η-expanded form. Most of the definitions in this section are adapted from [32].

Let $\mathfrak{T}_o$ be a set of basic types, then the set of types $\mathfrak{T}$ is generated by $\mathfrak{T} := \mathfrak{T}_o | \mathfrak{T} \to \mathfrak{T}$. Let $\mathfrak{C}$ be a signature of function symbols and let $\mathfrak{V}$ be a countably infinite set of variable symbols. *Variables* are normally denoted by the letters x, y, z and *function symbols* by the letters f, g, h, a. We sometimes use subscripts and superscript as well. We sometimes add a superscript to symbols in order to specify their type. The set $\mathtt{Term}^\alpha$ of terms of type α is generated by $\mathtt{Term}^\alpha := f^\alpha | x^\alpha | (\lambda x^\beta.\mathtt{Term}^\gamma) | (\mathtt{Term}^{\beta\to\alpha}\mathtt{Term}^\beta)$ where $f \in \mathfrak{C}, x \in \mathfrak{V}$ and $\alpha \in \mathfrak{T}$ (in the abstraction, $\alpha = \beta \to \gamma$). Applications throughout the paper will be associated to the left. We will sometimes omit brackets when the meaning is clear. We will also normally omit typing information when it is not crucial for the correctness of the results. $\tau(t^\alpha) = \alpha$ refers to the type of a term. The set $\mathtt{Term}$ denotes the set of all terms. *Subterms* and *positions* are defined as usual. We denote the fact that t is a (strict) subterm of s using the infix binary symbol $(\sqsubset)$ $\sqsubseteq$. *Sizes of positions* denote the length of the path to the position. We denote the subterm of t at position p by $t|_p$. *Bound* and *free variables* are defined as usual. All variable occurrences of different type and scope are assumed to be distinct. Given a term t, we denote by $\mathtt{hd}(t)$ its *head symbol* and distinguish between *flex* terms, whose head is a free variable and *rigid* terms, whose head is a function symbol or a bound variable.

Substitutions and their *composition* $(\circ)$ are defined as usual. $\mathtt{id}$ denotes the trivial substitution mapping each variable to itself. We denote by $\sigma|_W$ the substitution obtained from substitution σ by restricting its domain to variables in W. We denote by $\sigma[x \mapsto t]$ the substitution obtained from σ by mapping x to t, where x might already exist in the domain of σ. We extend the application of substitutions to terms in the usual way and denote it by postfix notation. Variable capture is avoided by implicitly renaming variables to fresh names upon binding. A substitution σ is *more general* than a substitution θ, denoted $\sigma \le \theta$, if there is a substitution δ such that $\sigma \circ \delta = \theta$. The *domain* of a substitution σ is denoted by $\mathtt{dom}(\sigma)$.

We introduce also a vector notation $\overline{t_n}$ for the sequence of terms $t_1, \ldots, t_n$. This notation also hold for nesting of sequences. For example, the term $f(x_1z_1z_2)(x_2z_1z_2)(x_3z_1z_2)$ will be denoted by $f\overline{x_3\overline{z_2}}$. The meaning of the notation $\lambda\overline{z_n}$ is $\lambda z_1, \ldots, \lambda z_n$. When the order of the sequence is not important, we will use this notation also for multisets.

2.2 Higher-order Pre-unification

In this section we present Huet's pre-unification procedure [17] as defined in [32]. The procedure will be proven, in Section 3.2, to be deterministic for the class of FCU problems. This result, together with the completeness of the procedure, implies the existence of most-general unifiers for unifiable problems of this class.

The presentation in this paper of both the pattern and FCU unification algorithms is much simplified if the following non-standard normal form is being used. All terms, including

functional existential variables but excluding the arguments of these variables, are considered to be in η-expanded form. The arguments of these variables are expected to be in η-normal forms. In a similar manner to the one in [32], one can prove that all substitutions used in this paper preserve this normal form.

▶ **Definition 1** (Unification Problem). An equation is a formula $t \doteq s$ where t and s are $\beta\eta$-normalized (see remark above) terms. A unification problem is a formula of the form $\exists\overline{q}.e_1 \wedge \ldots \wedge e_n$ where e_i for $0 < i \leq n$ is an equation.

▶ **Definition 2** (Unification System). A unification system over a signature $\mathfrak{C}$ is the following quadruple $\langle Q_\exists, Q_\forall, S, \sigma\rangle$ where $Q_\exists$ and $Q_\forall$ are disjoint sets of variables, S is a set of equations and σ a substitution. Given a unification problem $\exists\overline{q}.e_1 \wedge \ldots \wedge e_n$ we consider the unification system over signature $\mathfrak{C}$ by setting $Q_\exists = q$, $Q_\forall = \{\}$, $S = \{e_1, \ldots, e_n\}$ and $\sigma = \mathtt{id}$. Let $\mathtt{bvars}(e_i) = \overline{z_n}$ for $e_i = \lambda\overline{z_n}.t_i = \lambda\overline{z_n}.s_i$.

Unification systems will also be called systems.

Before presenting Huet's procedure for pre-unification, we will repeat the definition of partial bindings as given in [32].

▶ **Definition 3** (Partial bindings). A partial binding of type $\alpha_1 \to \ldots \to \alpha_n \to \beta$ where $\beta \in \mathfrak{T}_o$ is a term of the form
$\lambda\overline{y_n}.a(\lambda\overline{z^1_{p_1}}.x_1(\overline{y_n}, \overline{z^1_{p_1}}), \ldots, \lambda\overline{z^m_{p_m}}.x_m(\overline{y_n}, \overline{z^m_{p_m}}))$ for some atom a where

- $\tau(y_i) = \alpha_i$ for $0 < i \leq n$.
- $\tau(a) = \gamma_1 \to \ldots \to \gamma_m \to \beta$ where $\gamma_i = \delta^i_1 \to \ldots \to \delta^i_{p_i} \to \gamma'_i$ for $0 < i \leq m$.
- $\tau(z^i_j) = \delta^i_j$ for $0 < i \leq m$ and $0 < j \leq p_i$.
- x_i is a fresh variable and $\tau(x_i) = \alpha_1 \to \ldots \to \alpha_n \to \delta^i_1 \to \ldots \to \delta^i_{p_i} \to \gamma'_i$ for $0 < i \leq m$.
- $\gamma'_1, \ldots, \gamma'_m \in \mathfrak{T}_o$.

Partial bindings fall into two categories, imitation bindings, which for a given atom a and type α, are denoted by $\mathtt{PB}(a, \alpha)$ and projection bindings, which for a given index $0 < i \leq n$ and a type α, are denoted by $\mathtt{PB}(i, \alpha)$ and in which the atom a is equal to the bound variable y_i. Since partial bindings are uniquely determined by a type and an atom (up to renaming of the fresh variables $\overline{x_m}$), this defines a particular term.

▶ **Definition 4** (Huet's Pre-unification Procedure). Huet's pre-unification procedure is given in Table 1.

$$\frac{S}{S \cup \{A \doteq A\}}\ \mathtt{(Delete)} \qquad \frac{S \cup \{\lambda\overline{z_k}.s_1 \doteq \lambda\overline{z_k}.t_1, \ldots, \lambda\overline{z_k}.s_n \doteq \lambda\overline{z_k}.t_n\}}{S \cup \{\lambda\overline{z_k}.f(\overline{s_n}) \doteq \lambda\overline{z_k}.f(\overline{t_n})\}}\ \mathtt{(Decomp)}$$

$$\frac{S\sigma \cup \{x \doteq \lambda\overline{z_k}.t\} \qquad x \notin \mathtt{fvars}(t) \wedge \sigma = [\lambda\overline{z_k}.t/x]}{S \cup \{\lambda\overline{z_k}.x(\overline{z_k}) \doteq \lambda\overline{z_k}.t\}}\ \mathtt{(Bind)}$$

$$\frac{S \cup \{x \doteq u, \lambda\overline{z_k}.x^\alpha(\overline{s_n}) \doteq \lambda\overline{z_k}.f(\overline{t_m})\} \qquad u \in \mathtt{PB}(f, \alpha)}{S \cup \{\lambda\overline{z_k}.x^\alpha(\overline{s_n}) \doteq \lambda\overline{z_k}.f(\overline{t_m})\}}\ \mathtt{(Imitate)}^1$$

$$\frac{S \cup \{x \doteq u, \lambda\overline{z_k}.x^\alpha(\overline{s_n}) \doteq \lambda\overline{z_k}.a(\overline{t_m})\} \qquad 0 < i \leq k, u = \mathtt{PB}(i, \alpha)}{S \cup \{\lambda\overline{z_k}.x^\alpha(\overline{s_n}) \doteq \lambda\overline{z_k}.a(\overline{t_m})\}}\ \mathtt{(Project)}^2$$

1. where $f \in \mathfrak{C}$.
2. where either $a \in \mathfrak{C}$ or $a = z_i$ for some $0 < j \leq k$.

Table 1 PUA- Huet's pre-unification procedure

The next theorem states the completeness of this procedure.

▶ **Theorem 5** ([32]). *Given a system* $\langle Q_\exists, Q_\forall, S, \mathtt{id}\rangle$ *and assume it is unifiable by* σ*, then there is a sequence of rule applications in Def. 4 resulting in* $\langle Q'_\exists, Q'_\forall, \emptyset, \theta\rangle$ *such that* $\theta \leq \sigma$.

2.3 Pattern Unification

In this section we describe the higher-order pattern unification algorithm in [20]. The notation used is similar to the one in [25]. This algorithm forms the basis for our algorithm.

▶ **Definition 6** (Pattern Systems). A system $\langle Q_\exists, Q_\forall, S, \sigma\rangle$ is called a pattern system if for all equations $e_i \in S$ and for all subterms $x\overline{z_n}$ in these equations such that $x \in Q_\exists$ we have that $\overline{z_n} \subseteq Q_\forall \cup \mathtt{bvars}(e_i)$ and $z_i \neq z_j$ for all $0 < i < j \leq n$.

▶ **Definition 7** (Pruning). Given a pattern system $\langle Q_\exists, Q_\forall, S, \sigma\rangle$ such that $\lambda\overline{z_n}.x\overline{z_n^1} \doteq r \in S$, r contains an occurrence y such that $y \in Q_\forall \cup \overline{z_n}$ and $y \notin \overline{z_n^1}$:

- if there is a subterm $w\overline{z_m^2}$ of r such that $y = z_i^2$ for $0 < i \leq m$, then return $\langle Q_\exists \cup \{w'\} \setminus \{w\}, Q_\forall, S\theta, \sigma \circ \theta\rangle$ where $\theta = [w \mapsto \lambda\overline{z_m^2}.w'\overline{z_{m-1}^3}]$ and $\overline{z_{m-1}^3} = \overline{z_m^2} \setminus \{z_i^2\}$.
- otherwise, return $\langle Q_\exists, Q_\forall, \bot, \mathtt{id}\rangle$.

▶ **Definition 8** (Pattern Unification Algorithm). The pattern unification algorithm is the application of the rules from Table 2 such that before the application of rules (3) and (5) we apply exhaustively pruning.

$$\langle Q_\exists, Q_\forall, S \cup \{t \doteq t\}, \sigma\rangle \quad \to \quad \langle Q_\exists, Q_\forall, S\}, \sigma\rangle \tag{0}$$

$$\langle Q_\exists, Q_\forall, S \cup \{\lambda x.s \doteq \lambda x.t\}, \sigma\rangle \quad \to \quad \langle Q_\exists, Q_\forall \cup \{x\}, S \cup \{s \doteq t\}, \sigma\rangle \tag{1}$$

$$\langle Q_\exists, Q_\forall, S \cup \{f\overline{t_n} \doteq f\overline{s_n}\}, \sigma\rangle \quad \to \quad \langle Q_\exists, Q_\forall, S \cup \{t_1 \doteq s_1, \ldots, t_n \doteq s_n\}, \sigma\rangle \tag{2}$$

where $f \in \mathfrak{C} \cup Q_\forall$

$$\langle Q_\exists \cup \{x\}, Q_\forall, S \cup \{x\overline{z_n} \doteq f\overline{s_m}\}, \sigma\rangle \quad \to \quad \langle Q_\exists, Q_\forall, S\theta, \sigma \circ \theta\rangle \tag{3}$$

where $f \in \mathfrak{C}$, $x \notin \mathtt{fvars}(f\overline{s_m})$ and $\theta = [x \mapsto \lambda\overline{z_n}.f\overline{s_m}]$

$$\langle Q_\exists \cup \{x\}, Q_\forall, S \cup \{x\overline{z_n^1} \doteq x\overline{z_n^2}\}, \sigma\rangle \quad \to \quad \langle Q_\exists \cup \{w\}, Q_\forall, S\theta, \sigma \circ \theta\rangle \tag{4}$$

where $w \notin Q_\exists$, $\theta = [x \mapsto \lambda\overline{z_n^1}.w\overline{z_k^3}]$ and $\overline{z_k^3} = \{z_i^1 \mid z_i^1 = z_i^2\}$

$$\langle Q_\exists \cup \{y\}, Q_\forall, S \cup \{x\overline{z_n^1} \doteq y\overline{z_m^2}\}, \sigma\rangle \quad \to \quad \langle Q_\exists, Q_\forall, S\theta, \sigma \circ \theta\rangle \tag{5}$$

where $x \neq y$, $\theta = [y \mapsto \lambda\overline{z_m^2}.w\overline{z_{\phi(m)}^2}]$ and ϕ is a permutation (see pruning) such that $\phi(j) = i$ if $z_i^1 = z_j^2$ for $0 < i \leq n$ and $0 < j \leq m$

Table 2 Pattern Unification Algorithm

3 A Unification Algorithm for FC Higher-order Unification Problems

3.1 FC Higher-order Unification (FCU) Problems

The main difference between pattern and FCU problems is in the form of arguments of existentially quantified variables. While in pattern unification problems, these arguments must be a list of distinct universally quantified variables which occur in the scope of the existentially quantified one, we relax this requirement for FCU problems.

▶ **Definition 9** (Restricted Terms). Given $\mathfrak{C}$, $Q_\forall$ and an equation e, a restricted term in e is defined inductively as follows:

- $a \in Q_\forall \cup \mathtt{bvars}(e)$ is a restricted term.

- $f\overline{t_n}$ is a restricted term if $n > 0$, $f \in \mathfrak{C} \cup Q_\forall \cup \texttt{bvars}(e)$ and t_i is a restricted term for all $0 < i \leq n$.

▶ **Example 10.** the following are restricted terms over $\mathfrak{C} = \{f, g\}$ and $Q_\forall = \{a, b\}$: a, fab, $f(ga)b$ and $f(ga)a$.

▶ **Definition 11** (FCU Systems)**.** A system $\langle Q_\exists, Q_\forall, S, \sigma\rangle$ is an FCU system if

1. for all occurrences $x\overline{t_n}$ in S where $x \in Q_\exists$, t_i for all $0 < i \leq n$ is a restricted term and for each t_i and t_j such that $0 < j \leq n$ and $i \neq j$, $t_i \not\sqsubseteq t_j$.
2. for each two different occurrences $x\overline{t_n}$ and $y\overline{s_m}$ in S where $x, y \in Q_\exists$ and for each $0 < i \leq n$ and $0 < j \leq m$, $t_i \not\sqsubseteq s_j$.

▶ **Example 12.** Some FCU systems over $\mathfrak{C}$ and $Q_\forall$ defined as in the previous example are $\{xab \doteq f(ya)\}$ and $\{g(x(fa))(ga) \doteq f(y(fa)(gb))\}$. Note that only the first example is a pattern unification problem. An example of a non-FCU problem is $\{x(ga)a \doteq fb\}$

The next proposition is easy to verify.

▶ **Proposition 13.** *Pattern systems are FCU systems.*

3.2 The Existence of Most-general Unifiers

From this section on, an FCU problem will be referred simply as system, unless indicated otherwise.

In [32] it is claimed that the only "don't-know" non-determinism in the general higher-order procedure stems from the choice between the different applications of `(Imitate)` and `(Project)`. We prove, that in the case of FCU problems, this choice becomes deterministic.

We first prove a couple of auxiliary lemmas.

▶ **Lemma 14.** *let t be a restricted term, s a term containing the subterm $x\overline{r_n}$ and σ a substitution such that $t = s\sigma$, then there is a restricted term t' such that $t' = x\overline{r_n}\sigma$.*

Proof. Let k be the length of the position of $x\overline{r_n}$ in s, we prove by induction on k.

- $k = 0$, then $t' = t$.
- $k > 0$, then $s = f\overline{s_m}$ and $f\overline{s_m}\sigma = f\overline{t_m} = t$. Since t is restricted, by definition so are $t_1, \ldots, t_m$. Assume $x\overline{r_n}$ occurs in s_i, then, according to the IH, there is a restricted term t' such that $t' = x\overline{r_n}\sigma$.

◀

▶ **Lemma 15.** *Given a unifiable equation $x\overline{t_n} \doteq r$, where r is a restricted term. Then, there is $0 < i \leq n$ such that t_i is a subterm of r.*

Proof. Assume on the contrary and let σ be the unifier. Then, $x\overline{t_n}\sigma = r$. By definition, r contains a symbol $a \in Q_\forall$ and we get a contradiction. ◀

▶ **Lemma 16.** *Let $t = t'\overline{t_k}$ and $s = f\overline{s_n}$ such that t' is a restricted term and $f \in \mathfrak{C} \cup Q_\forall$. If $t \doteq s$ is unifiable, then $t = f\overline{v_{n-k}}\overline{t_k}$ for restricted terms $\overline{v_{n-k}}$.*

Proof. Since t' is restricted, it does not contain abstractions and variables and as t is unifiable with s, it can be written as $f\overline{v_{n-k}}$. Since t' is restricted, all its subterms are restricted as well. ◀

The next two lemmas prove the determinism claim on applications of `(Project)` and `(Imitate)`.

▶ **Lemma 17.** *Given the equation* $x\overline{t_n} \doteq f\overline{s_m}$ *where* x *does not occur in* $f\overline{s_m}$ *and assume obtaining the following two equations by applying the substitutions* $\sigma_0 = x \mapsto \lambda\overline{z_n}.z_i\overline{x_m\overline{z_n}}$ *and* $\theta_0 = x \mapsto \lambda\overline{z_n}.z_j\overline{x_k\overline{z_n}}$ *for some* $0 < i < j \leq n$*:*

$$t_i\overline{x_l\overline{t_n}} \doteq f\overline{s_m} \tag{1}$$

and

$$t_j\overline{y_k\overline{t_n}} \doteq f\overline{s_m} \tag{2}$$

Then, there are no substitutions σ *and* θ *such that* σ *unifies equation 1 and* θ *unifies equation 2.*

Proof. Assume the existence of the two unifiers and obtain a contradiction. According to Lemma 16, we can rewrite the two equations as

$$f\overline{v_{m-l}}\overline{x_l\overline{t_n}} \doteq f\overline{s_m} \tag{3}$$

and

$$f\overline{u_{m-k}}\overline{y_k\overline{t_n}} \doteq f\overline{s_m} \tag{4}$$

for restricted terms $\overline{v_{m-l}}$ and $\overline{u_{m-k}}$. Assume, wlog, that $l \geq k$. Note also, that since $t_i \neq t_j$ and t_i, t_j have f as head symbol, $m \geq m - k > 0$. We consider two cases:

- $s_1, \ldots, s_{m-k}$ are all ground terms. In this case and since both equations are unifiable, we get the equation

  $$f\overline{v_{m-l}}\overline{x_{l-k}\overline{t_n}\sigma} = f\overline{s_{m-k}} = f\overline{u_{m-k}} \tag{5}$$

  Clearly, $k \neq l$ since otherwise $t_i = t_j$. We can now conclude that

  $$x_1\overline{t_n}\sigma = u_{m-l+1} \tag{6}$$

  Since u_{m-l+1} is a restricted term then, according to Lemma 15, there is $0 < k_1 \leq n$ such that t_{k_1} is a subterm of u_{m-l+1}. Since u_{m-l+1} is a strict subterm of t_j, we get that t_{k_1} is a strict subterm of t_j which contradicts the definition of FCU problems.
- There is s_{k_1} for $0 < k_1 \leq m - k$ which contains an occurrence of $z\overline{r_{k_2}}$. According to the definition of FCU problems, this must occur as a subterm of s_{k_1} as otherwise the subterm $z\overline{r_{k_2}}r'$ where r' is not a restricted term then violates the definition. Since $s_{k_1}\theta = u_{k_1}$ and u_{k_1} is a restricted term, we have, according to Lemma 14, that there exist a restricted term u' such that $z\overline{r_{k_2}}\theta = u'$. Using Lemma 15, we can conclude that there is $0 < k_3 \leq k_2$ such that r_{k_3} is a subterm of u', which is a subterm of u_{k_1} which is a strict subterm of t_j, which violates the definition of FCU problems.

◀

▶ **Lemma 18.** *Given the equation* $x\overline{t_n} \doteq f\overline{s_m}$ *where* x *does not occur in* $f\overline{s_m}$ *and assume obtaining the following two equations by applying the substitutions* $\sigma_0 = x \mapsto \lambda\overline{z_n}.f\overline{x_m\overline{z_n}}$ *and* $\theta_0 = x \mapsto \lambda\overline{z_n}.z_j\overline{x_k\overline{z_n}}$ *for some* $0 < j \leq n$*.*

$$f\overline{x_m\overline{t_n}} \doteq f\overline{s_m} \tag{7}$$

and

$$t_j\overline{y_k\overline{t_n}} \doteq f\overline{s_m} \tag{8}$$

Then, there are no substitutions σ *and* θ *such that* σ *unifies equation 7 and* θ *unifies equation 8.*

Proof. Assume the existence of the two unifiers and obtain a contradiction. Using Lemma 16, we can rewrite Eq. 8 as:

$$f\overline{v_{m-k}}\overline{y_k t_n} \doteq f\overline{s_m} \tag{9}$$

where $v_1, \ldots, v_{m-k}$ are restricted terms and strict subterms of t_j. Since f is imitated, it is not a restricted term and $f \neq t_j$ which implies that $m - k > 0$. Eq. 9 tells us that $v_1 = s_1\theta$ which implies that $s_1\theta$ is a strict subterm of t_j and a restricted term. On the other hand, we have that $x_1\overline{t_n} = s_1\sigma$ from Eq. 7. We consider two cases:

- s_1 is ground. In this case we can use Lemma 15 and the fact that s_1 is a restricted term to conclude that there is $0 < k_1 \leq n$ such that t_{k_1} is a subterm of s_1. On the other hand, we know that s_1 is a strict subterm of t_j and therefore we get that t_{k_1} is a strict subterm of t_j, which contradicts the definition of pattern problems.
- If s_1 is not ground, it must contain an occurrence $z\overline{r_l}$. This occurrence cannot occur as the subterm $z\overline{r_l}r'$ where r' is not a restricted term as it violates the definition of FCU problems. Therefore, $z\overline{r_l}$ is a subterm of s_1. Since $s_1\theta = v_1$ and since v_1 is a restricted term, we can use Lemma 14 to get that there is a restricted term v' such that $z\overline{r_l} = v'$. Now we use Lemma 15 and get that there is $0 < k_1 \leq l$ such that r_{k_1} is a subterm of v', which is a strict subterm of t_j. We get again a contradiction to the definition of FCU problems.

◀

▶ **Theorem 19** (The existence of most-general unifiers). *Given a unifiable FCU system S. Applying the procedure in Def. 4 to S terminates and returns a most-general unifier for S.*

Proof. The procedure in Def. 4 computes complete sets of unifiers and terminates with an element in this set [32]. Using the lemmas 17 and 18 we obtain that all transformations are deterministic. Therefore, the complete set contains only one element, which is the most-general unifier of S. ◀

3.3 The Unification Algorithm

For defining the unification algorithm, we need to slightly extend the definition of pruning.

▶ **Definition 20** (Covers). A cover for $x\overline{t_n}$ and a restricted term q is a substitution σ such that $x\overline{t_n}\sigma = q$.

Note, uniqueness of covers follows from Theorem 19

▶ **Example 21.** The following substitution $[x \mapsto \lambda z_1 \lambda z_2 . f(fz_1)z_2]$ is a cover for $xa(fb)$ and $f(fa)(fb)$.

▶ **Definition 22** (Pruning). Given a FCU system $\langle Q_\exists, Q_\forall, S, \sigma\rangle$ such that $x\overline{t_n} \doteq r \in S$ and r contains an occurrence of a maximal restricted term q such that $q \notin \overline{t_n}$:

- if there is a subterm $w\overline{s_m}$ of r such that $q = s_i$ for $0 < i \leq m$, then return $\langle Q_\exists \cup \{w'\} \setminus \{w\}, Q_\forall, S\theta, \sigma \circ \theta\rangle$ where $\theta = [w \mapsto \lambda\overline{z_m}.w'\overline{z'_{m-1}}]$ and $\overline{z'_{m-1}} = \overline{z_m} \setminus \{z_i\}$.
- else if there is no cover ρ for $x\overline{t_n}$ and q, then return $\langle Q_\exists, Q_\forall, \bot, \mathtt{id}\rangle$.
- else, do nothing.

▶ **Example 23.** Given the system $\langle\{x, y, u, w\}, \{a, b\}, \{x(fa)b \doteq yb(ga), u(fa)b \doteq g(wb(ga)), \mathtt{id}\}$, we can apply the following three prunings, $\sigma_1 = [w \mapsto \lambda z_1, z_2.w'z_1]$, $\sigma_2 = [y \mapsto \lambda z_1, z_2.y'z_1]$ and $\sigma_3 = [x \mapsto \lambda z_1, z_2.x'z_2]$ and obtain the system $\langle\{x', y', u, w'\}, \{a, b\}, \{x'b \doteq y'b, u(fa)b \doteq g(w'b), \sigma_1 \circ \sigma_2 \circ \sigma_3\}$

For the next definition, we will use the following replacement operator $r\,|^{\overline{t_n}}_{\overline{z_n}}$ to denote the replacement of each occurrence t_i in r with z_i for $0 < i \leq n$.

▶ **Definition 24** (Algorithm for FCU Systems)**.** The rules of an algorithm for the unification of FCU systems is given in Table 3 where before the application of rules (3) and (5) we apply exhaustively pruning.

$$\langle Q_\exists, Q_\forall, S \cup \{t \doteq t\}, \sigma\rangle \rightarrow \langle Q_\exists, Q_\forall, S, \sigma\rangle \quad (0)$$

$$\langle Q_\exists, Q_\forall, S \cup \{\lambda x.s \doteq \lambda x.t\}, \sigma\rangle \rightarrow \langle Q_\exists, Q_\forall \cup \{x\}, S \cup \{s \doteq t\}, \sigma\rangle \quad (1)$$

$$\langle Q_\exists, Q_\forall, S \cup \{f\overline{t_n} \doteq f\overline{s_n}\}, \sigma\rangle \rightarrow \langle Q_\exists, Q_\forall, S \cup \{t_1 \doteq s_1, \ldots, t_n \doteq s_n\}, \sigma\rangle \quad (2)$$

where $f \in \mathfrak{C} \cup Q_\forall$

$$\langle Q_\exists \cup \{x\}, Q_\forall, S \cup \{x\overline{t_n} \doteq f\overline{s_m}\}, \sigma\rangle \rightarrow \langle Q_\exists, Q_\forall, S\theta, \sigma \circ \theta\rangle \quad (3)$$

where $f \in \mathfrak{C}$, $x \notin \mathtt{fvars}(f\overline{s_m})$ and $\theta = [x \mapsto \lambda\overline{z_n}.f\overline{s_m}\,|^{\overline{t_n}}_{\overline{z_n}}]$

$$\langle Q_\exists \cup \{x\}, Q_\forall, S \cup \{x\overline{t_n} \doteq x\overline{s_n}\}, \sigma\rangle \rightarrow \langle Q_\exists \cup \{w\}, Q_\forall, S\theta, \sigma \circ \theta\rangle \quad (4)$$

where $w \notin Q_\exists$, $\overline{t_n} \neq \overline{s_n}$, $\theta = [x \mapsto \lambda\overline{z_n}.w\overline{z_{r_k}}]$ and $\overline{r_k} = \{i \mid 0 < i \leq n \wedge t_i = s_i\}$

$$\langle Q_\exists \cup \{y\}, Q_\forall, S \cup \{x\overline{t_n} \doteq y\overline{s_m}\}, \sigma\rangle \rightarrow \langle Q_\exists, Q_\forall, S\theta, \sigma \circ \theta\rangle \quad (5)$$

where $x \neq y$, $\theta = [y \mapsto \lambda\overline{z_m}.x\overline{z_{\phi(m)}}]$ and ϕ is a permutation (see Lemma 32) such that $\phi(j) = i$ if $t_i = s_j$ for $0 < i \leq n$ and $0 < j \leq m$

Table 3 An algorithm for FCU problems

▶ **Example 25.** The following problem is contained in one of the classes of problems discussed in the introduction: $\exists C \exists D \lambda l_1 \lambda l_2.C(\mathrm{fst}l_1)(\mathrm{fst}(\mathrm{snd}(l_1)) \doteq \lambda l_1 \lambda l_2.f(D(\mathrm{fst}l_2)(\mathrm{fst}l_1))$ Table 4 gives a full execution of the algorithm on it.

$$\begin{array}{ll}
\langle\{C, D\}, \emptyset, \{\lambda l_1 \lambda l_2.C(\mathrm{fst}l_1)(\mathrm{fst}(\mathrm{snd}(l_1))) \doteq \lambda l_1 \lambda l_2.f(D(\mathrm{fst}l_2)(\mathrm{fst}l_1))\}, \mathtt{id}\rangle & \rightarrow^{(1)\times 2} \\
\langle\{C, D\}, \{l_1, l_2\}, \{C(\mathrm{fst}l_1)(\mathrm{fst}(\mathrm{snd}(l_1))) \doteq f(D(\mathrm{fst}l_2)(\mathrm{fst}l_1))\}, \mathtt{id}\rangle & \rightarrow^{\mathtt{prun}} \\
\langle\{C, D'\}, \{l_1, l_2\}, \{C(\mathrm{fst}l_1)(\mathrm{fst}(\mathrm{snd}(l_1))) \doteq f(D'(\mathrm{fst}l_1))\}, [D \mapsto \lambda z_1 \lambda z_2.D'z_2]\rangle & \rightarrow^{(3)} \\
\langle\{D'\}, \{l_1, l_2\}, \{f(D'(\mathrm{fst}l_1)) \doteq f(D'(\mathrm{fst}l_1))\}, [D \mapsto \lambda z_1 \lambda z_2.D'z_2, C \mapsto \lambda z_1 \lambda z_2.f(D'z_1)]\rangle & \rightarrow^{(0)} \\
\langle\{D'\}, \{l_1, l_2\}, \emptyset, [D \mapsto \lambda z_1 \lambda z_2.D'z_2, C \mapsto \lambda z_1 \lambda z_2.f(D'z_1)]\rangle &
\end{array}$$

Table 4 An example of a reduction on an FCU

Notice that this algorithm can also work with terms that are essentially untyped: it is the presence or absence of constructors and bound variables that matters in this algorithm and not the types of those constructors and variables. Rich typing can, of course, be used to disallow unifiers that are created by considering terms to be type-less.

4 Correctness of the Algorithm

The unification algorithm transforms systems by the application of substitutions and by the elimination of equations. We prove next that the application of rules of the algorithm in Def. 24 on FCU problems results in FCU problems as well.

▶ **Lemma 26.** *Given a FCU problem, then the application of rules from Def. 24 results in a FCU problem.*

Proof. Removing equations from the system clearly preserves the restrictions of FCU problems. This result is also immediate when applying substitutions as the only change to the arguments of the variables in the problem is to eliminate them. ◀

The following lemma states that projected arguments of variables on one side of the equation must always match arguments on the other side.

▶ **Lemma 27.** *Let $x\overline{t_n} \doteq r$ be an equation such that r contains an occurrence of $y\overline{s_m}$ where $r \neq y\overline{s_m}$ and let σ be a unifier of this equation such that $\sigma y = \lambda\overline{z_m}.s$. Then, for each occurrence z_i in s for $0 < i \leq m$, there is $0 < j \leq n$ such that $s_i = t_j$.*

Proof. We prove by induction on the number of occurrences. If s does not contain such occurrence, then the lemma clearly holds. Assume s contains an occurrence z_i for $0 < i \leq m$ and that there is no $0 < j \leq n$ such that $s_i = t_j$. In case there is more than one such occurrence in s, choose this occurrence to be in a minimal such subterm, i.e. z_i occurs in a subterm $z_i\overline{v_k}$ such that all occurrences of $z \in \overline{z_m}$ in $\overline{v_k}$ fulfill the requirement that there is $t_j = z$ for some $0 < j \leq n$. Let $\lambda\overline{z_m}.z_i\overline{v_k}(\overline{s_m}) = s_i\overline{v'_k}$. Since $r \neq y\overline{s_m}$ and the problem being FCU problem, we have that $y\overline{s_m} \sqsubset r$ Since $x\overline{t_n}\sigma = r\sigma$, we get that $s_i\overline{v'_k} \sqsubset x\overline{t_n}\sigma$. Since s_i is a restricted term, we get that there is $0 < j \leq n$ such that either

- $s_i\overline{v'_k} \sqsubseteq t_j$. By the minimality assumption, if $\overline{v'_k}$ contains a restricted term, then it must be equal to some t_l $0 < l \leq n$ and therefore, that $t_l \sqsubset t_j$, which contradicts the system being FCU. Therefore, since t_j is a restricted term, $k = 0$. We obtain that $s_i \sqsubseteq t_j$ and since $s_i \neq t_j$ by assumption, we get, again, a contradiction to the system being FCU.
- $t_j \sqsubseteq s_i$. Again, since $s_i \neq t_j$, we get a contraction to the system being FCU.

◀

We now prove, for each rule in the FCU algorithm, a relative completeness result. We start by the non-unifiability of problems with a positive occur check.

▶ **Lemma 28.** *Let $\langle Q_\exists, Q_\forall, S \cup \{x\overline{t_n} \doteq f\overline{s_m}\}, \sigma\rangle$ be a system such that x occurs in $\overline{s_m}$, then the system is not unifiable.*

Proof. Assume it is unifiable by θ and $\theta x = \lambda\overline{z_n}.s$. Consider two cases:

- s does not contain any occurrence of a variable z_i for $0 < i \leq n$. Let $\#t$ be the number of occurrences of symbols from $\mathfrak{C}$ in t. Then, $\#(x\overline{t_n}\theta) = \#(\theta x) \leq \#(\overline{s_m}\theta) < \#(f\overline{s_m}\theta)$ and we get a contradiction to $x\overline{t_n}\theta = f\overline{s_m}\theta$.
- In case s contains such an occurrence, then, according to Lemma 27, for all occurrences of z_i in s for $0 < i \leq n$, there is $0 < j \leq n$. Let ρ be the mapping between indices defined as above such that $\rho(i) = j$. Let $\overline{r_k}$ be the set of indices $0 < i \leq n$ which occurs in s for some $k \leq n$. Let p' be the non-trivial position of $x\overline{r_n}$ in $f\overline{s_m}$ and let p be the maximal position of a z_i in s for $i \in \overline{r_k}$. But then, there is an occurrence of $s_{\rho(i)}$ in $f\overline{s_m}\theta$ at position $p' \circ p$. Since $f\overline{s_m}\theta = x\overline{t_n}\theta$ and since $s_{\rho(i)}$ is a restricted term, there is an occurrence of z_i in s at position $p' \circ p$, in contradiction to the maximality of p.

◀

▶ **Lemma 29.** *Given a system $\langle Q_\exists, Q_\forall, S, \rho\}$ where $x\overline{t_n} \doteq r \in S$ and assume applying pruning in order to obtain system $\langle Q_\exists, Q_\forall, S', \rho'\}$ as defined in Def. 22, then, if S is unifiable by substitution σ, then there is a substitution σ', such that $\sigma = \theta \circ \sigma'$. If S is not unifiable, then S' is not unifiable.*

Proof. The rule is applicable only if there is such an occurrence q. Otherwise, $S = S'$ and $\theta = \mathtt{id}$. We consider the two cases in the lemma:

- there is a subterm $y\overline{s_m}$ of r such that $q = s_i$ for $0 < i \leq m$. If S is not unifiable, then assume S' is unifiable by σ' and since $S' = S\theta$, we get that S is unifiable by $\theta \circ \sigma'$, a contradiction. Assume the system is unifiable and let $\sigma y = \lambda\overline{z_m}.s$. Then,

according to Lemma 27, either there is $0 < j \leq n$, such that $t_j = s_i$, which contradicts the assumption, or s does not contain an occurrence of s_i. In the second case, by taking $\sigma' = \sigma|_{\mathfrak{V}(S)\setminus\{y\}}[w \mapsto \lambda\overline{z_{r_{m-1}}}.y\overline{z_m}\sigma]$ where $\overline{z_m} \setminus \overline{z'_{m-1}} \not\subseteq Q_\forall$, we get that $y\overline{s_m}\sigma = w\overline{s_{r_{m-1}}}\sigma' = y\overline{s_m}\theta \circ \sigma'$.

- If there is no such cover, then there is no substitution which unifies this equation.

◀

▶ **Lemma 30.** *Given a system $\langle Q_\exists, Q_\forall, S, \rho\rangle$ where $x\overline{t_n} \doteq s \in S$ and x does not occur in s and assume applying the substitution θ as defined in rule (3) in Table 3. Then, if σ a unifier of S, then there is σ' such that $\sigma = \theta \circ \sigma'$.*

Proof. We prove by induction on the structure of s. Note that two base cases are also defined in the last two cases below for $m = 0$.

- $s = y\overline{s_m}$ for $0 \leq m$. Note, that in this case, since $t_i \not\sqsubset s_j$ for $0 < i \leq n$ and $0 < j \leq m$ and that since $m \leq n$ due to pruning, we get that $x\overline{t_n}\theta = y\overline{t_{\phi(m)}}$ for ϕ defined as in rule (5) in Table 3. The rest of the proof is similar to the proof of Lemma 32.
- $s = t_i\overline{s_m}$ for some $0 < i \leq n$ and $0 \leq m$ and therefore $\theta x = \lambda\overline{z_n}.z_i\overline{s'_m}$ for some $\overline{s'_m}$. Assume applying `(Project)` with the substitution $\theta' = \lambda\overline{z_n}.z_i\overline{x_m\overline{z_n}}$ as defined in Def. 4. After applying `(Bind)` and possibly also several `(Decomp)`, we get the problem, since x cannot occur in s,

$$S'\theta' \cup \{x_1\overline{t_n} \doteq s_1, \ldots, x_m\overline{t_n} \doteq s_m\} \tag{10}$$

  Since, by assumption, $x\overline{t_n} \doteq t_i\overline{s_m}$ is unifiable and t_i is a restricted term, it follows, using an argument similar to the one in the proof of Lemma 17, that also each of the $x_j\overline{t_n} \doteq s_j$ is unifiable by some σ^o_j for $0 < j \leq m$. By following lemmas 17 and 18, we have that applying any other projection or imitation to $x\overline{t_n} \doteq t_i\overline{s_m}$ will render it non-unifiable. By using Theorem 5, we have that σ^o_j can be extended into a unifier σ_j of $S'\theta' \cup \{x_j\overline{t_n} \doteq s_j\}$ for all $0 < j \leq m$ and $\sigma = \theta' \circ \sigma_1 \circ \ldots \circ \sigma_m$. By applying the induction hypothesis, we get that there are substitutions σ'_j unifying $S'\theta' \cup \{x_j\overline{t_n} \doteq s_j\}$ for all $0 < j \leq m$ such that $\sigma_j = \theta_j \circ \sigma'_j$ for $\theta_j x_j = \lambda\overline{z_n}.s'_j$. I.e. that $\sigma = \theta' \circ \theta_1 \circ \sigma'_1 \circ \ldots \circ \theta_m \circ \sigma'_m$. Since each σ_j is a unifier of the above equations and σ is a unifier of S, we get that for each $m \geq j' > j$, $\sigma'_{j'} \leq \sigma'_j|_{\mathtt{dom}(\sigma'_{j'})}$. From this, together with the fact that the domain and range of each σ'_j do not contain variables from the domain of each $\theta_{j'}$ for $0 < j < j' \leq m$, we get that $\sigma = \theta' \circ \theta_1 \circ \ldots \circ \theta_m \circ \sigma'_1 \circ \ldots \circ \sigma'_m$. On the other hand, by applying θ, we get the problem

$$S'\theta \cup \{s'_1 \doteq s_1, \ldots, s'_m \doteq s_m\} \tag{11}$$

  But, since $\theta = \theta' \circ \theta_1 \circ \ldots \circ \theta_m$, we just need to choose $\sigma' = \sigma'_1 \circ \ldots \circ \sigma'_m$ and we have $\sigma = \theta \circ \sigma'$.
- $s = f\overline{s_m}$ for $0 \leq m$ and therefore $\theta x = \lambda\overline{z_n}.f\overline{s'_m}$. Assume applying `(Imitate)` with the substitution $\theta' = \lambda\overline{z_n}.f\overline{x_m\overline{z_n}}$. After applying `(Bind)` and a `(Decomp)`, we get the problem, since x cannot occur in s,

$$S'\theta' \cup \{x_1\overline{t_n} \doteq s_1, \ldots, x_m\overline{t_n} \doteq s_m\} \tag{12}$$

  From here we follow as before and use again Theorem 5 and Lemma 18.

◀

▶ **Lemma 31.** *Given a system $\langle Q_\exists, Q_\forall, S' \cup \{x\overline{t_n} \doteq x\overline{s_n}\}, \rho\rangle$ and assume applying the substitution θ as defined in rule (4) in Table 3. Then, if the system is unifiable by a substitution σ, then there is σ' such that $\sigma = \theta \circ \sigma'$.*

Proof. Assume that $\sigma x = \lambda\overline{z_n}.s$, we first prove that there is no occurrence z_i in s such that there is $0 < j \leq k$ where $i = r_j$ and $t_i \neq s_i$. Assume on the contrary, then $x\overline{t_n}\sigma = x\overline{s_n}\sigma$ which implies that $t_i = s_i$. Now we can define $\sigma' = \sigma|_{\mathfrak{V}(S)\setminus\{x\}}[w \mapsto \lambda\overline{z_{r_k}}.x\overline{z_n}\sigma]$ where $\overline{z_n} \setminus \overline{z_{r_k}} \not\subseteq Q_\forall$ are new variables. ◀

▶ **Lemma 32.** *Given a system $\langle Q_\exists, Q_\forall, S' \cup \{x\overline{t_n} \doteq y\overline{s_m}\}, \rho\rangle$ where $x \neq y$ and assume applying the substitution θ as defined in rule (5) in Table 3. Then, if the system is unifiable by a substitution σ, then there is σ' such that $\sigma = \theta \circ \sigma'$.*

Proof. First note that since we apply pruning beforehand (in a symmetric way), $n = m$ and ϕ is indeed a permutation. Assume, wlog, that $\sigma x = \lambda\overline{z_n}.s$. We know that for each occurrence z_i in s for $0 < i \leq n$, $s_i = t_{\phi(i)}$. By choosing $\sigma' = \sigma|_{\mathfrak{V}(S)\setminus\{y\}}$, we get that $y\overline{s_m}\sigma = x\overline{s_{\phi(m)}}\sigma = x\overline{s_{\phi(m)}}\sigma' = y\overline{s_m}\theta \circ \sigma'$. Therefore, $\sigma = \theta \circ \sigma'$. ◀

▶ **Theorem 33** (Termination). *Given a system $\langle Q_\exists, Q_\forall, S, \sigma\rangle$, the algorithm in Def. 24 always terminates.*

Proof. Let the tuple $m = \langle m_1, m_2\rangle$ where m_1 is the size of the set $Q_\exists$ and m_2 is the number of all symbols except $\doteq$ in S. Consider its lexicographical order, it is clear that m is well founded. We show that it decreases with every rule application of the algorithm:

- rules (0), (1) and (2) decrease m_2 and do no decrease m_1.
- rule (3) decreases m_1.
- rule (4) decreases m_2 and does not increase m_1.
- rule (5) decreases m_1.
- pruning decreases m_2 and does not increase m_1.

◀

▶ **Theorem 34** (Completeness). *Given a system $\langle Q_\exists, Q_\forall, S, \mathtt{id}\rangle$ and assume it is unifiable by σ, there there is a sequence of rule applications in Def. 24 resulting in $\langle Q'_\exists, Q'_\forall, \emptyset, \theta\rangle$ such that $\theta \leq \sigma$.*

Proof. Since the algorithm terminates and since it has a rule application for each unifiable equation, we obtain at the end the above system. Lemmas 28, 29, 30, 31 and 32 then give us that for any unifier σ of S, there is a substitution σ' such that $\sigma = \theta \circ \sigma'$. Therefore, $\theta \leq \sigma$. ◀

The next theorem is an easy corollary of the completeness theorem.

▶ **Theorem 35** (Most-general unifier). *Given a system $\langle Q_\exists, Q_\forall, S, \mathtt{id}\rangle$, if the algorithm defined in Def. 24 terminates with system $\langle Q'_\exists, Q'_\forall, \emptyset, \sigma\rangle$, then σ is an mgu of S.*

Proof. Since the algorithm in Def. 24 is deterministic, then we can use Theorem 34 in order to prove that σ is an mgu. ◀

The next theorem is proved by simulating the algorithm in Def. 24 using the procedure in Def. 4.

▶ **Theorem 36** (Soundness). *Given a system $\langle Q_\exists, Q_\forall, S, \mathtt{id}\rangle$ and assume there is a sequence of rule applications in Def. 24 resulting in $\langle Q'_\exists, Q'_\forall, \emptyset, \theta\rangle$, then θ is a unifier of S.*

Proof. It is obvious we can simulate each of the rules (0), (1), (2) and (3) using the procedure in Def. 4. We get the required result by using Theorem 19. For rules (4), (5) and the first case of pruning, assume there is another substitution ρ such that ρ unifies the problem and $\rho \not< \theta$. This can only happen if $\rho x = \lambda\overline{z_n}.w'\overline{r_{k'}}$ such that $\overline{r_k} \subset \overline{r'_{k'}}$. Lemma 31 states that there is no unifier ω and a substitution γ such that $\omega = \rho \circ \gamma$. Since the second case of pruning results in failure, we are done. ◀

5 Conclusion

We have described an extension of pattern unification called *function-as-constructor* unification. Such unification problems typically show up in situations where functions are applied to bound variables and where such functions are treated as term constructors (at least during the process of unification). We have shown that the properties that make first-order and pattern unification desirable for implementation—decidability and the existence of mgus for unifiable pairs—also hold for this class of unification problems. We are planning an implementation within the Leo-III theorem prover [34] and we then plan to compare this approach to unification with the implementation of Huet's pre-unification algorithm available in Leo-III when exercised again the THF set of problems within TPTP [8].

Acknowledgments. We thank Enrico Tassi for discussions related to this paper. This work has been funded by the ERC Advanced Grant ProofCert.

References

1 The Twelf project, 2016. `http://twelf.org/`.
2 Andreas Abel and Brigitte Pientka. Higher-order dynamic pattern unification for dependent types and records. In *Typed Lambda Calculi and Applications*, pages 10–26. Springer, 2011.
3 P. B. Andrews, F. Pfenning, S. Issar, and C. P. Klapper. The TPS theorem proving system. In Jörg H. Siekmann, editor, CADE 8, LNCS 230, pages 663–664. Springer, July 1986.
4 Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Crafting a proof assistant. In *Types for Proofs and Programs*, pages 18–32. Springer, 2006.
5 D. Baelde, K. Chaudhuri, A. Gacek, D. Miller, G. Nadathur, A. Tiu, and Y. Wang. Abella: A system for reasoning about relational specifications. *J. of Formalized Reasoning*, 2014.
6 Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, New York, revised edition, 1984.
7 Christoph Benzmüller and Michael Kohlhase. LEO – a higher order theorem prover. In *15th Conf. on Automated Deduction (CADE)*, pages 139–144, 1998.
8 Christoph Benzmüller, Florian Rabe, and Geoff Sutcliffe. THF0–the core of the TPTP language for higher-order logic. In *Automated Reasoning*, pages 491–506. Springer, 2008
9 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda-a functional language with dependent types. In *TPHOLs*, volume 5674, pages 73–78. Springer, 2009.
10 Chad E Brown. Satallax: An automatic higher-order prover. In *Automated Reasoning*, pages 111–117. Springer, 2012.
11 A. Church. A formulation of the Simple Theory of Types. *J. of Symbolic Logic*, 1940.
12 Mary Dalrymple, Stuart M. Shieber, and Fernando C. N. Pereira. Ellipsis and higher-order unification. *Linguistics and Philosophy*, 14:399–452, 1991.
13 Dominic Duggan. Unification with extended patterns. *Theoretical Computer Science*, 206(1):1–50, 1998.
14 R. Fettig and B. Löchner. Unification of higher-order patterns in a simply typed lambda-calculus with finite products and terminal type. In RTA 1996, LNCS 1103, pages 347–361.

15 Warren Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13:225–230, 1981.

16 Gérard Huet. The undecidability of unification in third order logic. *Information and Control*, 22:257–267, 1973.

17 Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

18 Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

19 Raymond McDowell and Dale Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. on Computational Logic*, 3(1):80–136, 2002.

20 Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. of Logic and Computation*, 1(4):497–536, 1991.

21 Dale Miller and Gopalan Nadathur. A computational logic approach to syntax and semantics. Presented at the Tenth Symposium of the Mathematical Foundations of Computer Science, IBM Japan, May 1985.

22 Dale Miller and Gopalan Nadathur. Some uses of higher-order logic in computational linguistics. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, pages 247–255, 1986.

23 Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.

24 Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus — A compiler and abstract machine based implementation of λProlog. CADE 16, LNAI 1632, pp. 287–291, 1999.

25 Tobias Nipkow. Functional unification of higher-order patterns. In M. Vardi, editor, *8th Symp. on Logic in Computer Science*, pages 64–74. IEEE, June 1993.

26 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Number 2283 in LNCS. Springer, 2002.

27 Frank Pfenning. Unification and anti-unification in the Calculus of Constructions. In G. Kahn, editor, *6th Symp. on Logic in Computer Science*, pages 74–85. IEEE, July 1991.

28 Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. CADE 16, LNAI 1632, pages 202–206, Trento, 1999.

29 Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In J. Giesl and R. Hähnle, editors, IJCAR, LNCS 6173, pages 15–21, 2010.

30 Xiaochu Qi, Andrew Gacek, Steven Holte, Gopalan Nadathur, and Zach Snow. The Teyjus system – version 2, 2015. `http://teyjus.cs.umn.edu/`.

31 Helmut Schwichtenberg. Minlog. In Freek Wiedijk, editor, *The Seventeen Provers of the World*, volume 3600 of *LNCS*, pages 151–157. Springer, 2006.

32 Wayne Snyder and Jean H. Gallier. Higher order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8(1-2):101–140, 1989.

33 Alwen F. Tiu. An extension of L-lambda unification. `http://www.ntu.edu.sg/home/atiu/llambdaext.pdf`, September 2002.

34 Max Wisniewski, Alexander Steen, and Christoph Benzmüller. The Leo-III project. In Alexander Bolotov and Manfred Kerber, editors, *Joint Automated Reasoning Workshop and Deduktionstreffen*, page 38, 2014.

35 Beta Ziliani and Matthieu Sozeau. A unification algorithm for Coq featuring universe polymorphism and overloading. ICFP 2015, pp. 179–191.