Efficient Full Higher-Order Unification (Technical Report)

Petar Vukmirović
Vrije Universiteit Amsterdam, De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands
p.vukmirovic@vu.nl
https://orcid.org/0000-0001-7049-6847

Alexander Bentkamp
Vrije Universiteit Amsterdam, De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands
a.bentkamp@vu.nl
https://orcid.org/0000-0002-7158-3595

Visa Nummelin
Vrije Universiteit Amsterdam, De Boelelaan 1081, 1081 HV Amsterdam, The Netherlands
visa.nummelin@vu.nl
https://orcid.org/0000-0003-0078-790X

Abstract

We developed a procedure to enumerate complete sets of higher-order unifiers based on work by Jensen and Pietrzykowski. Our procedure removes many redundant unifiers by carefully restricting the search space and tightly integrating decision procedures for fragments that admit a finite complete set of unifiers. We identify a new such fragment and describe a procedure for computing its unifiers. Our unification procedure is implemented in the Zipperposition theorem prover. Experimental evaluation shows a clear advantage over Jensen and Pietrzykowski's procedure.

2012 ACM Subject Classification Computing methodologies → Theorem proving algorithms

Keywords and phrases unification, higher-order logic, theorem proving, term rewriting, indexing data structures

1 Introduction

Unification is concerned with finding a substitution that makes two terms equal, for some notion of equality. Since the invention of Robinson's first-order unification algorithm [22], it has become an indispensable tool in many areas of computer science including theorem proving, logic programming, natural language processing, and programming language compilation.

Many of these applications are based on higher-order formalisms and require higher-order unification. Due to its undecidability and explosiveness, the higher-order unification problem is considered one of the main obstacles on the road to efficient higher-order tools.

One of the reasons for higher-order unification's explosiveness lies in *flex-flex pairs*, which consist of two applied variables, e.g., $F\,X \overset{?}{=} G\,\mathsf{a}$, where F, G, and X are variables and a is a constant. Even this seemingly simple problem has infinitely many incomparable unifiers. One of the first methods designed to combat this explosion is Huet's preunification [12]. Huet noticed that some logical calculi would remain complete if flex-flex pairs are not eagerly solved but postponed as constraints. If only flex-flex constraints remain, we know that a unifier must exist and we do not need to solve them. Huet's preunification has been used in many reasoning tools including Isabelle [20], Leo-III [26], and Satallax [5]. However, recent developments in higher-order theorem proving [2, 4] require enumeration of unifiers even for flex-flex problems, which is the focus of this report.

Jensen and Pietrzykowski's (JP) procedure [13] is the best known procedure for this purpose (Section 2). Given two terms to unify, it first identifies a position where the terms disagree. Then, in parallel branches of the search tree, it applies suitable substitutions, involving a variable either at the position of disagreement or above, and repeats this process on the resulting terms until they are equal or trivially nonunifiable.

Building on the JP procedure, we designed an improved procedure with the same completeness guarantees (Section 3). It addresses many of the issues that are detrimental to the performance of the JP procedure. First, the JP procedure does not terminate in many cases of obvious nonunifiability, e.g., for $X \overset{?}{=} \mathsf{f}\,X$, where X is a non-functional variable and f is a function constant. This example also shows that the JP procedure does not generalize Robinson's first-order procedure gracefully. To address this issue, our procedure detects whether a unification problem belongs to a fragment in which unification is decidable and finite complete sets of unifiers (CSUs) exist. We call algorithms that enumerate elements of the CSU for such fragments *oracles*. Noteworthy fragments with oracles are first-order terms, patterns [19], functions-as-constructors [16], and a new fragment we present in Section 5. The unification procedures of Isabelle [20] and Leo-III [26] check whether the unification problem belongs to a decidable fragment, but we take this idea a step further by checking this more efficiently and for every subproblem arising during unification.

Second, the JP procedure computes many redundant unifiers. Consider the example $F\,(G\,\mathsf{a}) \overset{?}{=} F\,\mathsf{b}$, where JP produces, in addition to the desired unifiers $\{F \mapsto \lambda x.\, H\}$ and $\{G \mapsto \lambda x.\, \mathsf{b}\}$, the redundant unifier $\{F \mapsto \lambda x.\, H,\ G \mapsto \lambda x.\, x\}$. The design of our procedure avoids computing many redundant unifiers, including this one. Additionally, as oracles usually return a small CSU, their integration reduces the number of redundant unifiers.

Third, the JP procedure repeatedly traverses the parts of the unification problem that have already been unified. Consider the problem $\mathsf{f}^{100}\,(G\,\mathsf{a}) \overset{?}{=} \mathsf{f}^{100}\,(H\,\mathsf{b})$, where the exponents denote repeated application. It is easy to see that this problem can be reduced to $G\,\mathsf{a} \overset{?}{=} H\,\mathsf{b}$. However, the JP procedure will wastefully retraverse the common context $\mathsf{f}^{100}[\,]$ after applying each new substitution. Since the JP procedure must apply substitutions to the variables occurring in the common context above the disagreement pair, it cannot be easily adapted to eagerly decompose unification pairs. By contrast, our procedure is designed to decompose the pairs eagerly, never traversing a common context twice.

Last, efficiently implemented algorithms for first-order [23] and pattern unification [19] apply substitutions and β-reduce lazily, i.e., only until the heads of the current unification pair are not mapped by the substitution and the terms are in head normal form. This is possible because in these algorithms each step is determined by the head symbols of the current unification pair. Since the JP procedure also needs to consider variables above the position of disagreement, it is unfit for optimizations of this kind. Our procedure depends only on the head of a current unification pair, enabling this optimization.

To filter out some of the terms that are not unifiable with a given query term from a set of terms, we developed a higher-order extension of fingerprint indexing (Section 6). We implemented our procedure, several oracles, and the fingerprint index in the Zipperposition prover (Section 7). Since a straightforward implementation of the JP procedure already existed in Zipperposition, we used it as a baseline to evaluate the performance of our procedure (Section 8). The results show substantial performance improvements over this baseline.

2 Background

Our setting is the simply typed λ-calculus. Types α, β, γ are either base types or functional types $\alpha \to \beta$. By convention, when we write $\alpha_1 \to \cdots \to \alpha_n \to \beta$, we assume β to be a base type. Basic terms are free variables (denoted $F, G, H, \ldots$), bound variables (x, y, z), and constants $(\mathsf{f}, \mathsf{g}, \mathsf{h})$. Complex terms are applications of one term to another $(s\,t)$ or λ-abstractions $(\lambda x.\, s)$. Following Nipkow [19], we use these syntactic conventions to distinguish free from bound variables. Bound variables with no enclosing binder, such as x in $\lambda y.\, x$, are called *loose bound variables*. We say that a term without loose bound variables is *closed* and a term without free variables is *ground*. Iterated λ-abstraction $\lambda x_1 \ldots \lambda x_n.\, s$ is abbreviated as $\lambda \overline{x}_n.\, s$ and iterated application $(s\,t_1)\ \ldots\ t_n$ as $s\,\overline{t}_n$, where $n \geq 0$. Similarly, we denote a sequence of terms $t_1, \ldots, t_n$ by $\overline{t}_n$, omitting its length $n \geq 0$ where it can be inferred or is irrelevant. Parameters and body for any term $\lambda \overline{x}.\, s$ are defined to be $\overline{x}$ and s respectively, where s is not a λ-abstraction. The *size* of a term is inductively defined as $\mathrm{size}(F) = 1$; $\mathrm{size}(x) = 1$; $\mathrm{size}(\mathsf{f}) = 1$; $\mathrm{size}(s\,t) = \mathrm{size}(s) + \mathrm{size}(t)$; $\mathrm{size}(\lambda x.\, s) = \mathrm{size}(s) + 1$.

We assume the standard notions of α-, β-, η-conversions. A term is in *head normal form* (*hnf*) if it is of the form $\lambda \overline{x}.\, a\,\overline{t}$, where a is a free variable, bound variable, or a constant. In this case, a is called the *head* of the term. By convention, a and b denote heads. If a is a variable, we call it a *flex* head; otherwise, we call it a *rigid* head. A term is called flex or rigid if its head is flex or rigid, respectively. By $s_{\downarrow \mathsf{h}}$ we denote the term obtained from a term s by repeated β-reduction of the leftmost outermost redex until it is in hnf. Unless stated otherwise, we view terms syntactically, as opposed to $\alpha\beta\eta$-equivalence classes. We write $s \leftrightarrow^*_{\alpha\beta\eta} t$ if s and t are $\alpha\beta\eta$-equivalent. Substitutions $(\sigma, \varrho, \theta)$ are functions from free and bound variables to terms; σt denotes application of σ to t, which α-renames t to avoid variable capture. The composition $\varrho\sigma$ of substitutions is defined by $(\varrho\sigma)t = \varrho(\sigma t)$. A variable F is mapped by σ if $\sigma F \not\leftrightarrow^*_{\alpha\beta\eta} F$. We write $\varrho \subseteq \sigma$ if for all variables F mapped by ϱ, $\varrho F \leftrightarrow^*_{\alpha\beta\eta} \sigma F$. Given a substitution ϱ, which maps F to s, we write $\varrho \setminus \{F \mapsto s\}$ to denote a substitution that does not map F and otherwise coincides with ϱ. Given substitutions ϱ and σ, which map disjoint sets of variables, we write $\varrho \cup \sigma$ to denote $\varrho\sigma$.

Deviating from the standard notion of higher-order subterm, we define subterms on β-reduced terms as follows: a term t is a subterm of t at position ε. If s is a subterm of u_i at position p, then s is a subterm of $a\ \overline{u}_n$ at position $i.p$. If s is a subterm of t at position p, then s is a subterm of $\lambda x.t$ at position $1.p$. Our definition of subterm is a graceful generalization of the corresponding first-order notion: a is a subterm of $\mathsf{f}\,\mathsf{a}\,\mathsf{b}$ at position 1, but f and $\mathsf{f}\,\mathsf{a}$ are not subterms of $\mathsf{f}\,\mathsf{a}\,\mathsf{b}$. A context is a term with zero or more subterms replaced by a hole $\square$. We write $C[\overline{u}_n]$ for the term resulting from filling in the holes of a context C with the terms $\overline{u}_n$ from left to right. The common context $\mathcal{C}(s, t)$ of two β-reduced η-long terms s and t of the same type is defined inductively as follows, assuming that $a \neq b$: $\mathcal{C}(\lambda x.\, s, \lambda y.\, t) = \lambda x.\mathcal{C}(s, \{y \mapsto x\}t)$; $\mathcal{C}(a\,\overline{s}_m, b\,\overline{t}_n) = \square$; $\mathcal{C}(a\,\overline{s}_m, a\,\overline{t}_m) = a\,\mathcal{C}(s_1, t_1)\ \ldots\ \mathcal{C}(s_m, t_m)$.

A *unifier* for terms s and t is a substitution σ, such that $\sigma s \leftrightarrow^*_{\alpha\beta\eta} \sigma t$. Following JP [13], a *complete set of unifiers* (*CSU*) of terms s and t is defined as a set U of unifiers for s and t such that for every unifier ϱ of s and t, there exists $\sigma \in U$ and substitution θ such that $\varrho \subseteq \theta\sigma$. A *most general unifier* (*MGU*) is a one-element CSU. We use $\subseteq$ instead of $=$ because a CSU element σ may introduce auxiliary variables not mapped by ϱ.

3 The Unification Procedure

To unify two terms s and t, our procedure builds a tree as follows. The nodes of the tree have the form (E, σ), where E is a multiset of unification constraints $\{(s_1 \overset{?}{=} t_1), \ldots, (s_n \overset{?}{=} t_n)\}$ and σ is the substitution constructed up to that point. A unification constraint $s \overset{?}{=} t$ is an unordered pair of two terms of the same type. The root node of the tree is $(\{s \overset{?}{=} t\}, \mathrm{id})$, where id is the identity substitution. The tree is then constructed applying the transitions listed below. The leaves of the tree are either a failure node $\bot$ or a substitution σ. Ignoring failure nodes, the set of all substitutions in the leaves forms a complete set of unifiers for s and t.

The transitions are parametrized by a mapping $\mathcal{P}$ that assigns a set of substitutions to a unification pair. Moreover, the transitions are parametrized by a selection function S mapping a multiset E of unification constraints to one of those constraints $S(E) \in E$, the *selected* constraint in E. The transitions, defined as follows, are only applied if the grayed constraint is selected.

Succeed $\quad (\varnothing, \sigma) \longrightarrow \sigma$

Normalize$_{\alpha\eta}$ $\quad (\{\lambda \overline{x}_m.\, s \overset{?}{=} \lambda \overline{y}_n.\, t\} \uplus E, \sigma) \longrightarrow (\{\lambda \overline{x}_m.\, s \overset{?}{=} \lambda \overline{x}_m.\, t'\, x_{n+1} \ldots x_m\} \uplus E, \sigma)$
where $m \geq n$, $\overline{x}_m \neq \overline{y}_n$, and $t' = \{y_1 \mapsto x_1, \ldots, y_n \mapsto x_n\}t$

Normalize$_{\beta}$ $\quad (\{\lambda \overline{x}.\, s \overset{?}{=} \lambda \overline{x}.\, t\} \uplus E, \sigma) \longrightarrow (\{\lambda \overline{x}.\, s_{\downarrow \mathsf{h}} \overset{?}{=} \lambda \overline{x}.\, t_{\downarrow \mathsf{h}}\} \uplus E, \sigma)$
where s or t is not in hnf

Dereference $\quad (\{\lambda \overline{x}.\, F\, \overline{s} \overset{?}{=} \lambda \overline{x}.\, t\} \uplus E, \sigma) \longrightarrow (\{\lambda \overline{x}.\, (\sigma F)\, \overline{s} \overset{?}{=} \lambda \overline{x}.\, t\} \uplus E, \sigma)$
where none of the previous transitions apply and F is mapped by σ

Fail $\quad (\{\lambda \overline{x}.\, a\, \overline{s}_m \overset{?}{=} \lambda \overline{x}.\, b\, \overline{t}_n\} \uplus E, \sigma) \longrightarrow \bot$
where none of the previous transitions apply, and a and b are different rigid heads

Delete $\quad (\{s \overset{?}{=} s\} \uplus E, \sigma) \longrightarrow (E, \sigma)$
where none of the previous transitions apply

OracleSucc $\quad (\{s \overset{?}{=} t\} \uplus E, \sigma) \longrightarrow (E, \varrho\sigma)$
where none of the previous transitions apply, some oracle found a finite CSU U for $\sigma(s) \overset{?}{=} \sigma(t)$, and $\varrho \in U$; if multiple oracles found a CSU, only one of them is considered

OracleFail $\quad (\{s \overset{?}{=} t\} \uplus E, \sigma) \longrightarrow \bot$
where none of the previous transitions apply, and some oracle determined $\sigma(s) \overset{?}{=} \sigma(t)$ has no solutions

Decompose $\quad (\{\lambda \overline{x}.\, a\, \overline{s}_m \overset{?}{=} \lambda \overline{x}.\, a\, \overline{t}_m\} \uplus E, \sigma) \longrightarrow (\{s_1 \overset{?}{=} t_1, \ldots, s_m \overset{?}{=} t_m\} \uplus E, \sigma)$
where none of the transitions Succeed to OracleFail apply

Bind $\quad (\{s \overset{?}{=} t\} \uplus E, \sigma) \longrightarrow (\{s \overset{?}{=} t\} \uplus E, \varrho\sigma)$
where none of the transitions Succeed to OracleFail apply, and $\varrho \in \mathcal{P}(s \overset{?}{=} t)$.

The transitions are designed so that only OracleSucc, Decompose, and Bind can introduce parallel branches in the constructed tree. OracleSucc can introduce branches using different unifiers of the CSU; Bind can introduce branches using different substitutions in $\mathcal{P}$; and Decompose can be applied in parallel with Bind.

Our approach is to apply substitutions and $\alpha\beta\eta$-normalize terms lazily. In particular, the transitions that modify the constructed substitution, OracleSucc and Bind, do not apply that substitution to the unification pairs directly. Instead, the transitions Normalize$_{\alpha\eta}$, Normalize$_{\beta}$, and Dereference partially normalize and partially apply the constructed substitution just enough to ensure that the heads are the ones we would get if the substitution was fully applied and the term was fully normalized. To support lazy dereferencing, $\mathsf{OracleSucc}$ and Bind must maintain the invariant that all substitutions are idempotent.

The $\mathsf{OracleSucc}$ and $\mathsf{OracleFail}$ transitions invoke oracles, such as pattern unification, to compute a CSU faster, produce fewer redundant unifiers, and discover nonunifiability earlier. In some cases, addition of oracles lets the procedure terminate more often.

In the literature, oracles are usually stated under the assumption that their input belongs to the appropriate fragment. To use oracles efficiently, they must be redesigned to lazily discover whether the terms belong to their fragment. Often it is sufficient to check if the terms belong to the fragment only when performing certain operations inside the oracle. For example, many oracles contain a decomposition operation, which usually does not depend on the terms belonging to a certain fragment. This allows us to extend our lazy dereferencing and normalization approach to the implementation of the oracles.

The core of the procedure lies in the Bind step, parameterized by the mapping $\mathcal{P}$ that determines which substitutions (called *bindings*) to create. The bindings are defined as follows:

Iteration for F Let F be a free variable of the type $\alpha_1 \to \cdots \to \alpha_n \to \beta_1$ and let some α_i be the type $\gamma_1 \to \cdots \to \gamma_m \to \beta_2$, where $n > 0$ and $m \geq 0$. Iteration for F at i is:

$$F \mapsto \lambda \overline{x}_n.\, H\, \overline{x}_n\, (\lambda \overline{y}.\, x_i\, (G_1\, \overline{x}_n\, \overline{y}) \ldots (G_m\, \overline{x}_n\, \overline{y}))$$

The free variables H and $G_1, \ldots, G_m$ are fresh, and $\overline{y}$ is an arbitrary-length sequence of bound variables of arbitrary types. All new variables (both free and bound) are of appropriate type. Due to indeterminacy of $\overline{y}$, this step is infinitely branching.

JP-style projection for F Let F be a free variable of type $\alpha_1 \to \cdots \to \alpha_n \to \beta$, where some α_i is equal to β and $n > 0$. Then the JP-style projection binding is

$$F \mapsto \lambda \overline{x}_n.\, x_i$$

Huet-style projection for F Let F be a free variable of type $\alpha_1 \to \cdots \to \alpha_n \to \beta$, where some $\alpha_i = \gamma_1 \to \cdots \to \gamma_m \to \beta$, $n > 0$ and $m \geq 0$. Huet-style projection is as follows:

$$F \mapsto \lambda \overline{x}_n.\, x_i\, (F_1\, \overline{x}_n)\, \ldots\, (F_m\, \overline{x}_n)$$

where the fresh free variables $\overline{F}_m$ and bound variables $\overline{x}_n$ are of appropriate types.

Imitation of g for F Let F be a free variable of type $\alpha_1 \to \cdots \to \alpha_n \to \beta$ and let g be a constant of type $\gamma_1 \to \cdots \to \gamma_m \to \beta$ where $n, m \geq 0$. The imitation binding is given by

$$F \mapsto \lambda \overline{x}_n.\, \mathsf{g}\, (F_1\, \overline{x}_n) \ldots (F_m\, \overline{x}_n)$$

where the fresh free variables $\overline{F}_m$ and bound variables $\overline{x}_n$ are of appropriate types.

Identification for F and G Let F and G be different free variables. Also, let the type of F be $\alpha_1 \to \cdots \to \alpha_n \to \beta$ and the type of G be $\gamma_1 \to \cdots \to \gamma_m \to \beta$, where $n, m \geq 0$. Then, identification binding binds F and G with

$$F \mapsto \lambda \overline{x}_n.\, H\, \overline{x}_n\, (F_1\, \overline{x}_n) \ldots (F_m\, \overline{x}_n) \qquad G \mapsto \lambda \overline{y}_m.\, H\, (G_1\, \overline{y}_m) \ldots (G_n\, \overline{y}_m)\, \overline{y}_m$$

where the fresh free variables $H, \overline{F}_m, \overline{G}_n$ and bound variables $\overline{x}_n$,$\overline{y}_m$ are of appropriate types. We call fresh variables emerging from this binding in the role of H *identification variables.*

Elimination for F Let F be a free variable of type $\alpha_1 \to \cdots \to \alpha_n \to \beta$, where $n > 0$. In addition, let $1 \leq j_1 < \cdots < j_i \leq n$ and $i < n$. Elimination for the sequence $(j_k)_{k=1}^{i}$ is

$$F \mapsto \lambda \overline{x}_n.\, G\, x_{j_1}\, \ldots\, x_{j_i}$$

where the fresh free variable G as well as all x_{j_k} are of appropriate type. We call fresh variables emerging from this binding in the role of G *elimination variables.*

We define $\mathcal{P}$ as follows, given a unification constraint $\lambda\overline{x}.\, s \overset{?}{=} \lambda\overline{x}.\, t$:

- If the constraint is rigid-rigid, $\mathcal{P}(\lambda\overline{x}.\, s \overset{?}{=} \lambda\overline{x}.\, t) = \varnothing$.
- If the constraint is flex-rigid, let $\mathcal{P}(\lambda\overline{x}.\, F\,\overline{s} \overset{?}{=} \lambda\overline{x}.\, a\,\overline{t})$ be
  - an imitation of a for F, if a is some constant g, and
  - all Huet-style projections for F, if F is not an identification variable.
- If the constraint is flex-flex and the heads are different, let $\mathcal{P}(\lambda\overline{x}.\, F\,\overline{s} \overset{?}{=} \lambda\overline{x}.\, G\,\overline{t})$ be
  - all identifications and iterations for both F and G, and
  - all JP-style projections for non-identification variables among F and G.
- If the constraint is flex-flex and the heads are identical we consider two cases:
  - if the head is an elimination variable, $\mathcal{P}(\lambda\overline{x}.\, s \overset{?}{=} \lambda\overline{x}.\, t) = \varnothing$;
  - otherwise, let $\mathcal{P}(\lambda\overline{x}.\, F\,\overline{s} \overset{?}{=} \lambda\overline{x}.\, F\,\overline{t})$ be all iterations for F at arguments of functional type and all eliminations for F.

Comparison with the JP Procedure In contrast to our procedure, the JP procedure constructs a tree with only one unification constraint per node and does not have a Decompose rule. Instead, at each node $(s \overset{?}{=} t, \sigma)$, the JP procedure computes the common context C of s and t, yielding term pairs $(s_1, t_1), \ldots, (s_n, t_n)$, called *disagreement pairs*, such that $s = C[s_1, \ldots, s_n]$ and $t = C[t_1, \ldots, t_n]$. The procedure heuristically chooses one of the disagreement pairs (s_i, t_i) and applies a binding to the heads of s_i and t_i or to a free variable occurring above the disagreement pair in the common context C. Due to this application of bindings above the disagreement pair, lazy normalization and dereferencing cannot easily be integrated into the JP procedure.

Our procedure uses many of the same binding rules as the JP procedure, but it explores the search space differently. In particular, the JP procedure allows iteration or elimination to be applied at a free variable in the common context of the unification constraint, even if bindings were already applied below that free variable. In contrast, we force the eliminations and iterations to be applied as soon as we observe a flex-flex pair with identical heads. After applying the Decompose transition, we can apply other bindings below this flex-flex pair, but we cannot resume applying eliminations or iterations to the flex-flex pair.

The bindings of our procedure contain further optimizations that are missing in the JP procedure: The JP procedure applies eliminations for only one parameter at a time, yielding multiple paths to the same unifier. It applies imitations to flex-flex pairs, which we found to be unnecessary. On flex-rigid pairs, it applies JP-style projections and iterations instead of the finitely branching Huet-style projections. Moreover, it does not keep track of which rule introduced which variable, i.e., iterations and eliminations are applied on elimination variables, and projections are applied on identification variables.

Examples We present some illustrative derivations. The displayed branches of the constructed trees are not necessarily exhaustive. We abbreviate JP-style projection as JP Proj, imitation as Imit, identification as Id, Decompose as Dc, Dereference as Dr, Normalize$_\beta$ as N$_\beta$, and Bind of a binding x as B(x). Transitions of the JP procedure are denoted by $\Longrightarrow$. For the JP transitions we implicitly apply the generated bindings and fully normalize terms, which significantly shortens JP derivations.

▶ **Example 1.** The JP procedure does not terminate on the problem $G \overset{?}{=} \mathsf{f}\, G$:

$$(G \overset{?}{=} \mathsf{f}\, G, \mathrm{id}) \overset{\mathsf{Imit}}{\Longrightarrow} (\mathsf{f}\, G' \overset{?}{=} \mathsf{f}^2\, G', \sigma_1) \overset{\mathsf{Imit}}{\Longrightarrow} (\mathsf{f}^2\, G'' \overset{?}{=} \mathsf{f}^3\, G'', \sigma_2) \overset{\mathsf{Imit}}{\Longrightarrow} \cdots$$

where $\sigma_1 = \{G \mapsto \lambda x.\, \mathsf{f}\, G'\}$ and $\sigma_2 = \{G' \mapsto \lambda x.\, \mathsf{f}\, G''\}\sigma_1$. By including any oracle that supports first-order occurs check, such as the pattern oracle or the fixpoint oracle described in Section 7, our procedure gracefully generalizes first-order unification:

$$(\{G \overset{?}{=} \mathsf{f}\, G\}, \mathrm{id}) \xrightarrow{\mathsf{OracleFail}} \bot$$

▶ **Example 2.** The following derivation illustrates the advantage of the Decompose rule.

$$\begin{aligned}
&(\{\mathsf{h}^{100}\,(F\,\mathsf{a}) \overset{?}{=} \mathsf{h}^{100}\,(G\,\mathsf{b})\}, \mathrm{id}) \xrightarrow{\mathsf{Dc}^{100}} (\{F\,\mathsf{a} \overset{?}{=} G\,\mathsf{b}\}, \mathrm{id}) \xrightarrow{\mathsf{B(Id)}} (\{F\,\mathsf{a} \overset{?}{=} G\,\mathsf{b}\}, \sigma_1)\\
&\xrightarrow{\mathsf{Dr{+}N}_\beta} (\{H\,\mathsf{a}\,(F'\,\mathsf{a}) \overset{?}{=} H\,(G'\,\mathsf{b})\,\mathsf{b}\}, \sigma_1) \xrightarrow{\mathsf{Dc}} (\{\mathsf{a} \overset{?}{=} G'\,\mathsf{b}, F'\,\mathsf{a} \overset{?}{=} \mathsf{b}\}, \sigma_1)\\
&\xrightarrow{\mathsf{B(Imit)}} (\{\mathsf{a} \overset{?}{=} G'\,\mathsf{b}, F'\,\mathsf{a} \overset{?}{=} \mathsf{b}\}, \sigma_2) \xrightarrow{\mathsf{Dr{+}N}_\beta} (\{\mathsf{a} \overset{?}{=} \mathsf{a}, F'\,\mathsf{a} \overset{?}{=} \mathsf{b}\}, \sigma_2) \xrightarrow{\mathsf{Delete}} (\{F'\,\mathsf{a} \overset{?}{=} \mathsf{b}\}, \sigma_2)\\
&\xrightarrow{\mathsf{B(Imit)}} (\{F'\,\mathsf{a} \overset{?}{=} \mathsf{b}\}, \sigma_3) \xrightarrow{\mathsf{Dr{+}N}_\beta} (\{\mathsf{b} \overset{?}{=} \mathsf{b}\}, \sigma_3) \xrightarrow{\mathsf{Delete}} (\varnothing, \sigma_3) \xrightarrow{\mathsf{Succeed}} \sigma_3
\end{aligned}$$

where $\sigma_1 = \{F \mapsto \lambda x.\, H\, x\, (F'\, x), G \mapsto \lambda y.\, H\,(G'\, y)\, y\}$; $\sigma_2 = \{G' \mapsto \lambda x.\, \mathsf{a}\}\sigma_1$; and $\sigma_3 = \{F' \mapsto \lambda x.\, \mathsf{b}\}\sigma_2$. The JP procedure produces the same intermediate substitutions σ_1 to σ_3, but since it does not decompose the terms, it retraverses the common context $\mathsf{h}^{100}\,[\,]$ at every step to identify the contained disagreement pair:

$$\begin{aligned}
&(\mathsf{h}^{100}\,(F\,\mathsf{a}) \overset{?}{=} \mathsf{h}^{100}\,(G\,\mathsf{b}), \mathrm{id}) \xRightarrow{\mathsf{Id}} (\mathsf{h}^{100}\,(H\,\mathsf{a}\,(F'\,\mathsf{a})) \overset{?}{=} \mathsf{h}^{100}\,(H\,(G'\,\mathsf{b})\,\mathsf{b}), \sigma_1)\\
&\xRightarrow{\mathsf{Imit}} (\mathsf{h}^{100}\,(H\,\mathsf{a}\,(F'\,\mathsf{a})) \overset{?}{=} \mathsf{h}^{100}\,(H\,\mathsf{a}\,\mathsf{b}), \sigma_2) \xRightarrow{\mathsf{Imit}} (\mathsf{h}^{100}\,(H\,\mathsf{a}\,\mathsf{b}) \overset{?}{=} \mathsf{h}^{100}\,(H\,\mathsf{a}\,\mathsf{b}), \sigma_3) \xRightarrow{\mathsf{Succeed}} \sigma_3
\end{aligned}$$

▶ **Example 3.** The search space restrictions also allow us to avoid returning some redundant unifiers. Consider the example $F\,(G\,\mathsf{a}) \overset{?}{=} F\,\mathsf{b}$, where a and b are of base type. Our procedure produces only one failing branch and the following two successful branches:

$$\begin{aligned}
&(\{F\,(G\,\mathsf{a}) \overset{?}{=} F\,\mathsf{b}\}, \mathrm{id}) \xrightarrow{\mathsf{Dc}} (\{G\,\mathsf{a} \overset{?}{=} \mathsf{b}\}, \mathrm{id}) \xrightarrow{\mathsf{B(Imit)}} (\{G\,\mathsf{a} \overset{?}{=} \mathsf{b}\}, \{G \mapsto \lambda x.\, \mathsf{b}\})\\
&\xrightarrow{\mathsf{Dr{+}N}_\beta} (\{\mathsf{b} \overset{?}{=} \mathsf{b}\}, \{G \mapsto \lambda x.\, \mathsf{b}\}) \xrightarrow{\mathsf{Delete}} (\varnothing, \{G \mapsto \lambda x.\, \mathsf{b}\}) \xrightarrow{\mathsf{Succeed}} \{G \mapsto \lambda x.\, \mathsf{b}\}
\end{aligned}$$

$$\begin{aligned}
&(\{F\,(G\,\mathsf{a}) \overset{?}{=} F\,\mathsf{b}\}, \mathrm{id}) \xrightarrow{\mathsf{B(Elim)}} (\{F\,(G\,\mathsf{a}) \overset{?}{=} F\,\mathsf{b}\}, \{F \mapsto \lambda x.\, F'\})\\
&\xrightarrow{\mathsf{Dr{+}N}_\beta} (\{F' \overset{?}{=} F'\}, \{F \mapsto \lambda x.\, F'\}) \xrightarrow{\mathsf{Delete}} (\varnothing, \{F \mapsto \lambda x.\, F'\}) \xrightarrow{\mathsf{Succeed}} \{F \mapsto \lambda x.\, F'\}
\end{aligned}$$

The JP procedure additionally produces the following redundant unifier:

$$\begin{aligned}
&(F\,(G\,\mathsf{a}) \overset{?}{=} F\,\mathsf{b}, \mathrm{id}) \xRightarrow{\mathsf{JP\ Proj}} (F\,\mathsf{a} = F\,\mathsf{b}, \{G \mapsto \lambda x.\, x\})\\
&\xRightarrow{\mathsf{Elim}} (F' = F', \{G \mapsto \lambda x.\, x, F \mapsto \lambda x.\, F'\}) \xRightarrow{\mathsf{Succeed}} \{G \mapsto \lambda x.\, x, F \mapsto \lambda x.\, F'\}
\end{aligned}$$

Moreover, the JP procedure does not terminate because an infinite number of iterations is applicable at the root. In contrast, our procedure terminates since, in this case, we only apply iteration binding for non base-type arguments, which F does not have.

Proof of Completeness Our completeness theorem is stated as follows:

▶ **Theorem 4.** *The procedure described above is complete, meaning that the substitutions on the leaves of the constructed tree form a CSU. In other words, for any unifier ϱ of a multiset of constraints E there exists a derivation $(E, \mathrm{id}) \longrightarrow^* \sigma$ and a substitution θ such that $\varrho \subseteq \theta\sigma$.*

The proof is given in Section 4.

Pragmatic Variant We structured our procedure so that most of the unification machinery is contained in the Bind step. Modifying $\mathcal{P}$, we can sacrifice completeness and obtain a pragmatic variant of the procedure that often performs better in practice. Our informal experiments showed that the following modification of $\mathcal{P}$ is a reasonable compromise between completeness and performance. It removes all iteration bindings to enforce a finitely branching procedure and replaces JP-style projections by Huet-style projections.

- If the constraint is rigid-rigid, $\mathcal{P}(\lambda\overline{x}.\, s \overset{?}{=} \lambda\overline{x}.\, t) = \varnothing$.
- If the constraint is flex-rigid, let $\mathcal{P}(\lambda\overline{x}.\, F\,\overline{s} \overset{?}{=} \lambda\overline{x}.\, a\,\overline{t})$ be
  - an imitation of a for F, if a is some constant g, and
  - all Huet-style projections for F if F is not an identification variable.
- If the constraint is flex-flex and the heads are different, let $\mathcal{P}(\lambda\overline{x}.\, F\,\overline{s} \overset{?}{=} \lambda\overline{x}.\, G\,\overline{t})$ be
  - an identification binding for F and G, and
  - all Huet-style projections for F if F is not an identification variable
- If the constraint is flex-flex and the heads are identical we consider two cases:
  - if the head is an elimination variable, $\mathcal{P}(\lambda\overline{x}.\, F\,\overline{s} \overset{?}{=} \lambda\overline{x}.\, F\,\overline{t}) = \varnothing$;
  - otherwise, $\mathcal{P}(\lambda\overline{x}.\, F\,\overline{s} \overset{?}{=} \lambda\overline{x}.\, F\,\overline{t})$ is the set of all eliminations bindings for F.

Moreover, the pragmatic variant imposes limits on the number of bindings applied, counting the applications of bindings locally, per constraint. It is useful to distinguish the Huet-style projection cases where α_i is a base type (called *simple projection*), which always reduces the problem size, and the cases where α_i is a functional type (called *functional projection*). We limit applications of the following bindings: functional projections, eliminations, imitations and identifications. In addition, a limit on the total number of applied bindings can be set. An elimination binding that removes k arguments counts as k elimination steps. Due to limits on application of bindings, the pragmatic variant terminates.

To fail as soon as any of the limits is reached, the pragmatic variant employs an additional oracle. If this oracle determines that the limits are reached and the constraint is of the form $\lambda\overline{x}.\, F\,\overline{s}_m \overset{?}{=} \lambda\overline{x}.\, G\,\overline{t}_n$, it returns a *trivial unifier* – a substitution $\{F \mapsto \lambda\overline{x}_m.\, H, G \mapsto \lambda\overline{x}_n.\, H\}$, where H is a fresh variable; if the limits are reached and the constraint is flex-rigid, the oracle fails; if the limits are not reached, it reports that terms are outside its fragment. The trivial unifier prevents the procedure from failing on easily unifiable flex-flex pairs.

Careful tuning of each limit optimizes the procedure for a specific class of problems. For problems coming from higher-order reasoning front-ends, shallow unification depth usually suffices. However, hard hand-crafted problems often need deeper unification.

4 Proof of Completeness

The proof of the completeness theorem (Theorem 4) is an adaptation of the proof presented by JP [13]. Definitions and lemmas are reused, but are combined together differently to suit our procedure. The "JP" labels in lemmas and definitions refer to the corresponding lemmas and definitions in the original proof.

The backbone of the proof is as follows. We incrementally define states (E_j, σ_j) and *remainder substitutions* ϱ_j starting with $(E_0, \sigma_0) = (E, \mathrm{id})$ and $\varrho_0 = \varrho$. These will satisfy the invariants that ϱ_j unifies E_j and $\varrho_0 \subseteq \varrho_j\sigma_j$. Intuitively, ϱ_j is what remains to be added to σ_j to reach a unifier subsuming ϱ_0. In each step, ϱ_j is used as a guide to choose the next transition $(E_j, \sigma_j) \longrightarrow (E_{j+1}, \sigma_{j+1})$.

To show that we eventually reach a state with an empty E_j, we employ a well-founded measure of (E_j, ϱ_j) that strictly decreases with each step. It is the lexicographic product of the syntactic size of $\varrho_j E_j$ and a measure on ϱ_j, which is taken from the JP proof.

In this section, we view terms as $\alpha\beta\eta$-equivalence classes, with the $\beta\eta$-normal form as their canonical representative. Moreover, we consider all substitutions to be fully applied. These assumptions are justified because replacing the lazy transitions $\mathsf{Normalize}_{\alpha\eta}$, $\mathsf{Normalize}_{\beta}$, and Dereference by eager counterparts only affects the efficiency but not the overall behavior of our procedure since all bindings depend only on the head of terms.

Fix a state (E_j, σ_j). If E_j is empty, then a unifier σ_j of E is found by Succeed and we are done because $\varrho_0 \subseteq \varrho_j \sigma_j$ by induction hypothesis. Otherwise, let $E_j = \{u \overset{?}{=} v\} \uplus E'_j$ where $u \overset{?}{=} v$ is selected. We must find a transition that reduces the measure and preserves the invariants. Fail and OracleFail cannot be applicable, because $\varrho_j u = \varrho_j v$ by the induction hypothesis. If applicable, Delete reduces the size of $\varrho_j E_j$ by removing a constraint.

OracleSucc has similar effect as Delete, but the remainder changes. Since ϱ_j is a unifier of $u \overset{?}{=} v$ and oracles compute CSUs, the oracle will find a unifier δ such that there exists a ϱ_{j+1} satisfying $\varrho_j \subseteq \varrho_{j+1}\, \delta$. Then $(E_{j+1}, \sigma_{j+1}) = \big(\delta E'_j, \delta\, \sigma_j\big)$ is a result of an OracleSucc transition. Observe that $\varrho_{j+1} E_{j+1} = \varrho_{j+1} \delta E'_j$ is a proper subset of $\varrho_j E_j$. Hence, the measure decreases and ϱ_{j+1} unifies E_{j+1}. The other invariant holds, because $\varrho_0 \subseteq \varrho_j\, \sigma_j \subseteq \varrho_{j+1}\, \delta\, \sigma_j = \varrho_{j+1}\, \sigma_{j+1}$.

If none of the previous transitions are applicable, we must find the right Decompose or Bind transition to apply. The choice is determined by the head a of u, the head b of v, and their values under ϱ_j. If $u \overset{?}{=} v$ is flex-rigid, then either $\varrho_j a$ has b as head symbol, enabling imitation, or $\varrho_j a$ has a parameter as head symbol, enabling Huet-style projection. In the flex-flex case, if $a \neq b$, we apply either iteration, identification, or JP-style projection based on the form of $\varrho_j a$ and $\varrho_j b$. Similarly, if $a = b$, we apply either iteration, elimination, or Decompose guided by the form of $\varrho_j a$. To show preservation of the induction invariants for Bind, we determine a binding δ that can be factored out of ϱ_j as $\varrho_j \subseteq \varrho_{j+1}\, \delta$ similar to the OracleSucc case. Here we have $\varrho_{j+1} E_{j+1} = \varrho_j E_j$; so we must ensure that the measure of ϱ_{j+1} is strictly smaller than that of ϱ_j. For Decompose, we set $\varrho_{j+1} = \varrho_j$ and show that $\varrho_{j+1} E_{j+1}$ is smaller than $\varrho_j E_j$.

In the following, we elaborate our proof in full detail.

▶ **Definition 5** (JP D1.6). Given two terms t and s and their common context C, we can write t as $C[\overline{t}]$ and s as $C[\overline{s}]$ for some $\overline{t}$ and $\overline{s}$. The pairs (s_j, t_j) are called *disagreement pairs.*

▶ **Definition 6** (JP D3.1). Given two terms t and s, let $\lambda\overline{x}.\, t'$ and $\lambda\overline{y}.\, s'$ be respective α-equivalent terms such that their parameters $\overline{x}$ and $\overline{y}$ are disjoint. Then the disagreement pairs of t' and s' are called *opponent pairs* in t and s.

▶ **Lemma 7** (JP L3.3 (1)). *Let ϱ be a substitution and X, Y be free variables such that $\varrho(X\, \overline{s}) = \varrho\big(Y\, \overline{t}\big)$ for some term tuples $\overline{s}$ and $\overline{t}$. Then for every opponent pair u, v in ϱX and ϱY (Definition 6), the head of u or v is a parameter of ϱX or ϱY.*

In contrast to first order, a higher order context $C[\,]$ need not to be injective as a mapping from terms to terms. As a result, eagerly decomposing $C[s] \overset{?}{=} C[t]$ into $s \overset{?}{=} t$ is not always complete. The concept of ω-simplicity captures an important subset of injective contexts $C[\,]$. Non-ω-simplicity on the other hand is the main trigger of Iteration—the most explosive binding of our procedure.

▶ **Definition 8** (JP D3.2). An occurrence of a parameter x of term t in the body of t is *ω-simple* if both

1. the arguments of x are distinct and are exactly (the η-long forms of) all of the variables bound in the body of t, and
2. this occurrence of x is not in an argument of any parameter of t.

This definition is slightly too restrictive for our purposes. It is unfortunate that condition 1 requires x to be applied to *all* instead of just some of the bound variables. The JP proof would probably work with such a relaxation, and the definition would be equivalent to injectivity with respect to the parameter for any values of other parameters. However, to reuse the JP lemmas, we stick to the original notion of ω-simplicity and introduce the following relaxation:

▶ **Definition 9.** An occurrence of a parameter x of term t in the body of t is *base-simple* if it is ω-simple or both

1. x is of base type, and
2. this occurrence of x is not in an argument of any parameter of t.

▶ **Lemma 10.** *Let s have parameters $\overline{x}$ and a subterm $x_j\,\overline{v}$ where this occurrence of x_j is base-simple. Then for any sequence $\overline{t}$ of (at least j) terms, the body of t_j is a subterm of $s\,\overline{t}$ (after normalization) at the position of $x_j\,\overline{v}$ up to renaming of the parameters of t_j. To compare positions of s and $s\,\overline{t}$, ignore the parameter count mismatch.*

Proof. Consider the process of β-normalizing $s\,\overline{t}$. After substituting terms $\overline{t}$ into the body of s, a further reduction can only take place when some t_k is an abstraction that gets arguments in s. The arguments $\overline{v}$ to the x_j are distinct variables bound in the body of s. This follows easily from either case of the definition of base-simplicity. So t_j is applied to the unmodified $\overline{v}$ after substituting terms $\overline{t}$ into the body of s. Base-simplicity also implies that $t_j\,\overline{v}$ does not occur in an argument to another t_k. Hence only the reduction of $t_j\,\overline{v}$ itself affects this subterm. The variables $\overline{v}$ match the parameter count of t_j because we consider the η-long form of t_j; so $t_j\,\overline{v}$ reduces to the body of t_j (modulo renaming). The position is obviously that of $x_j\,\overline{v}$. ◀

▶ **Lemma 11** (JP C3.4 strengthened)**.** *Let ϱ be a substitution and X a free variable. If $\varrho(\lambda\overline{x}.\,X\,\overline{s}) = \varrho\big(\lambda\overline{x}.\,X\,\overline{t}\big)$ and some occurrence of the i^{th} parameter of ϱX is base-simple, then $\varrho s_i = \varrho t_i$.*

Proof. By Lemma 10, ϱs_i occurs in $\varrho X(\varrho\overline{s})$ at certain position that depends only on ϱX. Similarly ϱt_i occurs in $\varrho X\big(\varrho\overline{t}\big) = \varrho X(\varrho\overline{s})$ at the same position, and hence $\varrho s_i = \varrho t_i$. ◀

We define more properties to determine which binding to apply to a given constraint. Roughly speaking, the simple comparison form will trigger identification bindings, projectivity will trigger Huet-style projections, and simple projectivity will trigger JP-style projections.

▶ **Definition 12** (JP D3.4)**.** We say that s and t are in *simple comparison form* if all ω-simple heads of opponent pairs in s and t are distinct, and each opponent pair has an ω-simple head.

▶ **Definition 13** (JP D3.5)**.** A term t is called *projective* if the head of t is a parameter of t. If the whole body is just the parameter, then t is called *simply projective.*

A central part of the proof is to find a suitable measure for the remaining problem size. Showing that the measure is strictly decreasing and well-founded guarantees that the procedure finds a suitable substitution in finitely many steps. We reuse the measure for remainder substitutions from JP [13], but embed it into a lexicographic measure to handle the decomposition steps and oracles of our procedure.

▶ **Definition 14** (JP D3.7)**.** The *free weight* of a term t is the total number of occurrences of free variables and constants in t. The *bound weight* of t is the total number of occurrences (excluding occurrences λx) of bound variables in t, but with the particular exemption: if a prefix variable u has one or more ω-simple occurrences in the body, then one such occurrence and its arguments are not counted. It does not matter which occurrence is not counted because in η-long form the bound weight of the arguments of an ω-simple variable is the same for all occurrences of that variable.

▶ **Definition 15** (JP D3.8)**.** For multisets E of unification constraints and substitutions ϱ, our measure on pairs (E, ϱ) is the lexicographic comparison of

A the sum of the sizes of the terms in E
B the sum over the free weight of ϱF, for all variables F mapped by ϱ
C the sum over the bound weight of ϱF, for all variables F mapped by ϱ
D the sum over the number of parameters of ϱF, for all variables F mapped by ϱ

We denote the quadruple containing these numbers as $\mathrm{ord}(E, \varrho)$. We denote the triple containing only the last three components of $\mathrm{ord}(E, \varrho)$ as $\mathrm{ord}\,\varrho$. We write $<$ for the lexicographic comparison of these tuples.

The next six lemmas correspond to the bindings of our procedure and sufficient conditions for the binding to bring us closer to a given solution. This is expressed as decrease of the ord measure of the remainder. In each of these lemmas, let u be a term with a variable head a and v a term with an arbitrary head b. Let ϱ be a unifier of u and v. The conclusion, let us call it $\boldsymbol{C}$, is always the same: there exists a binding δ applicable to the problem $u \overset{?}{=} v$, and there exists a substitution ϱ' such that $\varrho \subseteq \varrho'\,\delta$ and $\mathrm{ord}\,\varrho' < \mathrm{ord}\,\varrho$. For most of these lemmas, we refer to JP [13] for proofs. However, our bindings have more preconditions, yielding additional orthogonal hypotheses in our lemmas.

▶ **Lemma 16** (JP L3.9)**.** *If $a = b$ is not an elimination variable and ϱa discards any of its parameters, then $\boldsymbol{C}$ by elimination. Moreover, for the elimination variable G introduced by this elimination, $\varrho' G$ discards none of its parameters and has the body of ϱa.*

Proof. Let $\varrho a = \lambda \overline{x}.\, t$ and let $(x_{j_k})_{k=1}^{i}$ be the subsequence of $\overline{x}$ consisting of those variables which occur in the body t. It is a strict subsequence, since ϱa is assumed to discard some parameter. As equal heads $a = b$ of the constraint $u \overset{?}{=} v$ are not elimination variables, elimination for $(j_k)_{k=1}^{i}$ can be applied. Let $\delta = \{a \mapsto \lambda \overline{x}.\, G\, x_{j_1}\, \ldots\, x_{j_i}\}$ be the corresponding binding. Define ϱ' to be like ϱ except

$$\varrho' a = a \quad \text{and} \quad \varrho' G = \lambda x_{j_1} ... x_{j_i}.\, t.$$

Obviously $\varrho' G$ is a closed term and $\varrho \subseteq \varrho'\,\delta$ holds. Moreover $\mathrm{ord}\,\varrho' < \mathrm{ord}\,\varrho$, because free and bound weights stay the same (ϱa and $\varrho' G$ have the same body t) whereas the number of parameters strictly decreases. The definition of $(j_k)_{k=1}^{i}$ implies that $\varrho' G$ discards none of its parameters. ◀

▶ **Lemma 17** (JP L3.10)**.** *Assume that there exists a parameter x of ϱa such that x has a non-ω-simple (Definition 8) occurrence in ϱa, which is not below another parameter, or such that x has at least two ω-simple occurrences in ϱa. Moreover, if $a = b$, to make iteration applicable, a must not be an elimination variable, and x must be of functional type. Then $\boldsymbol{C}$ is achieved by iteration.*

▶ **Lemma 18** (JP L3.11). *Assume that a and b are different free variables. If ϱa is simply projective (Definition 13) and a is not an identification variable, then $\boldsymbol{C}$ by JP-style projection.*

▶ **Lemma 19** (JP L3.12). *If ϱa is not projective and b is rigid, then $\boldsymbol{C}$ by imitation.*

▶ **Lemma 20** (JP L3.13). *Let $a \neq b$. Assume that $\varrho a \neq a$ and $\varrho b \neq b$ are in simple comparison form (Definition 12) and neither is projective. Then $\boldsymbol{C}$ by identification. Moreover, $\varrho' H$ is not projective, where H is the identification variable introduced by this application of the identification binding.*

Proof. This is JP's Lemma 3.13, plus the claim that $\varrho' H$ is not projective. Inspecting the proof of that lemma, it is obvious that $\varrho' H$ cannot be projective because ϱa and ϱb are not projective. ◀

▶ **Lemma 21.** *Assume that ϱa is projective (Definition 13), a is not an identification variable, and b is rigid. Then $\boldsymbol{C}$ by Huet-style projection.*

Proof. Since ϱa is projective, we have $\varrho a = \lambda \overline{x}_n.\, x_k\, \overline{t}_m$ for some k and some terms $\overline{t}_m$. If ϱa is also simply projective, then x_k must be non-functional and since Huet-style projection and JP-style projection coincide in that case, Lemma 18 applies. Hence, in the following we may assume that ϱa is not simply projective, i.e., that $m > 0$.

Let δ be the Huet-style projection binding:

$$\delta = \{a \mapsto \lambda \overline{x}_n.\, x_i\, (F_1\, \overline{x}_n)\, \ldots\, (F_m\, \overline{x}_n)\}$$

for fresh variables $F_1, \ldots, F_m$. This binding is applicable because b is rigid. Let ϱ' be the same as ϱ except that we set $\varrho' a = a$ and for each $1 \leq j \leq m$ we set

$$\varrho' F_j = \lambda \overline{x}_n.\, t_j$$

It remains to show that $\operatorname{ord} \varrho' < \operatorname{ord} \varrho$. The free weight of ϱa is the same as the sum of the free weights of $\varrho' F_j$ for $1 \leq j \leq m$. Thus, the free weight is the same for ϱ and ϱ'. The bound weight of ϱa however is exactly 1 larger than the sum of the bound weights of $\varrho' F_j$ for $1 \leq j \leq m$ because of the additional occurrence of x_k in ϱa. The exemption for ω-simple occurrences in the definition of the bound weight cannot be triggered by this occurrence of x_k because $m > 0$ and thus x_k is not ω-simple. It follows that $\operatorname{ord} \varrho' < \operatorname{ord} \varrho$. ◀

We are now ready to prove the completeness theorem (Theorem 4).

Proof. Let $E_0 = E$ and $\sigma_0 = \mathrm{id}$. Let $\varrho_0 = \tau\varrho$ for some renaming τ, such that the free variables of $\varrho_0 E$ and E are disjoint. Then ϱ_0 unifies E_0 because ϱ unifies E by assumption. Moreover, $\varrho_0 = \varrho_0\sigma_0$. We proceed to inductively define E_j, σ_j and ϱ_j until we reach some j such that $E_j = \varnothing$. To guarantee well-foundedness, we ensure that the measure $\operatorname{ord}(\varrho_j, E_j)$ decreases with each step. We maintain the following invariants for all j:

- $(E_j, \sigma_j) \longrightarrow (E_{j+1}, \sigma_{j+1})$;
- $\operatorname{ord}(\varrho_j, E_j) > \operatorname{ord}(\varrho_{j+1}, E_{j+1})$;
- ϱ_j unifies E_j;
- $\varrho_0 \subseteq \varrho_j\sigma_j$;
- the free variables in $\varrho_j E_j$ and E_j are disjoint;
- for every identification variable X, $\varrho_j X$ is not projective; and
- for every elimination variable X, each parameter of $\varrho_j X$ has occurrences in $\varrho_j X$, all of which are base-simple.

If $E_j \neq \varnothing$, let $u \overset{?}{=} v$ be the selected constraint $S(E_j)$ in E_j.

First assume that an oracle is able to find a CSU for the constraint $u \overset{?}{=} v$. Since ϱ_j unifies u and v, by the definition of a CSU, the CSU discovered by the oracle contains a unifier δ of u and v such that there exists ϱ_{j+1} satisfying $\varrho_j \subseteq \varrho_{j+1}\,\delta$. Thus, an OracleSucc transition is applicable and yields the node $(E_{j+1}, \sigma_{j+1}) = (\delta(E_j \setminus \{u \overset{?}{=} v\}), \delta\,\sigma_j)$. The strict containment $\varrho_{j+1}E_{j+1} \subset \varrho_{j+1}\,\delta\,E_j = \varrho_j E_j$ implies $\mathrm{ord}(E_{j+1}, \varrho_{j+1}) < \mathrm{ord}(E_j, \varrho_j)$. It also shows that the constraints $\varrho_{j+1}E_{j+1}$ are unified when $\varrho_j E_j$ are. By direct computation $\varrho_0 \subseteq \varrho_j\sigma_j \subseteq \varrho_{j+1}\,\delta\,\sigma_j = \varrho_{j+1}\sigma_{j+1}$. Variable disjointness follows by picking ϱ_{j+1} in the right way. New identification or elimination variables are not introduced so their properties are preserved. Hence all invariants are preserved.

Otherwise we proceed by a case distinction on the form of $u \overset{?}{=} v$. Typically, one of the Lemmas 16–21 is going to apply. Any one of them gives substitutions ϱ' and δ with properties that let us define $E_{j+1} = \delta E_j$, $\sigma_{j+1} = \delta\,\sigma_j$ and $\varrho_{j+1} = \varrho'$. The problem size always strictly decreases, because these lemmas imply $\varrho_{j+1}E_{j+1} = \varrho_{j+1}\delta E_j = \varrho_j E_j$ and $\mathrm{ord}\,\varrho_{j+1} = \mathrm{ord}\,\varrho' < \mathrm{ord}\,\varrho_j$. Regarding the other invariants, the former equation guarantees that ϱ_{j+1} unifies E_{j+1}, and $\varrho_0 \subseteq \varrho_j\,\sigma_j \subseteq \varrho_{j+1}\,\delta\,\sigma_j = \varrho_{j+1}\,\sigma_{j+1}$ holds by computation. The conditions on variables must be checked separately when new ones are introduced. Let a be the head of $u = \lambda\overline{x}.\, a\,\overline{u}$ and b be the head of $v = \lambda\overline{x}.\, b\,\overline{v}$. Consider the following cases:

$\underline{u \text{ and } v \text{ have the same head symbol } a = b}$:

(1) Suppose that $\varrho_j a$ has a parameter with non-base-simple occurrence. By one of the induction invariants, a is not an elimination variable. Among all non-base-simple occurrences of parameters in $\varrho_j a$, choose the leftmost one, which we call x. This occurrence of x cannot be below another parameter, because having x occur in one of its arguments would make that other parameter non-base-simple, contradicting the occurrence of x being leftmost. Thus x is neither base-simple nor below another parameter; so x is of functional type. Moreover, non-base-simplicity implies non-ω-simplicity. Hence, we can apply Lemma 17 (iteration).

(2) Otherwise suppose that $\varrho_j a$ discards some of its parameters. By one of the induction invariants, $\varrho_j a$ is not an elimination variable. Hence Lemma 16 (elimination) applies. The newly introduced elimination variable G satisfies the required invariants, because Lemma 16 guarantees that $\varrho_{j+1}G$ uses its parameters and shares the body with $\varrho_j a$ which by assumption of this case contains only base-simple occurrences.

(3) Otherwise every parameter of $\varrho_j a$ has occurrences and all of them are base-simple. We are going to show that Decompose is a valid transition and decreases $\varrho_j E_j$. By Lemma 11 we conclude from $\varrho_j u = \varrho_j v$ that $\varrho_j u_i = \varrho_j v_i$ for every i. Hence the new constraints $E_{j+1} = E_j \setminus \{u \overset{?}{=} v\} \cup \{u_i \overset{?}{=} v_i \mid \text{for all } i\}$ after Decompose are unified by ϱ_j. This allows us to define $\varrho_{j+1} = \varrho_j$ and $\sigma_{j+1} = \sigma_j$. To check that $\varrho_{j+1}E_{j+1} = \varrho_j E_{j+1}$ is smaller than $\varrho_j E_j$ it suffices to check that constraints $\varrho_j u_i \overset{?}{=} \varrho_j v_i$ together are smaller than $\varrho_j u \overset{?}{=} \varrho_j v$. Since all parameters of $\varrho_j a$ have base-simple occurrences, $\varrho_j u_i$ is a subterm of $\varrho_j u = \lambda\overline{x}.\, \varrho_j a\,(\varrho_j\overline{u})$ by Lemma 10. Similarly for $\varrho_j v$. It follows that $\varrho_{j+1}E_{j+1}$ is smaller than $\varrho_j E_j$. Since $\varrho_{j+1} = \varrho_j$ and $\sigma_{j+1} = \sigma_j$, the other invariants are obviously preserved.

$\underline{u \text{ and } v \text{ is a flex-flex pair with different heads:}}$

(4) First, suppose that $\varrho_j a$ or $\varrho_j b$ is simply projective (Definition 13). By the induction hypothesis, the simply projective head cannot be an identification variable. Thus Lemma 18 (JP-style projection) applies.

(5) Otherwise suppose that $\varrho_j a$ is projective but not simply. Then the head of $\varrho_j a$ is its some parameter x_k. But this occurrence cannot be ω-simple because it has arguments which

cannot be bound above the head x_k. Thus Lemma 17 (iteration) applies. If $\varrho_j b$ is projective but not simply, the same argument applies.

(6) Otherwise suppose that $\varrho_j a$, $\varrho_j b$ are in simple comparison form (Definition 12). By one of the the induction invariants, the free variables in $\varrho_j E_j$ and E_j are disjoint. Thus $\varrho_j a \neq a$ and $\varrho_j b \neq b$. Then Lemma 20 (identification) applies.

(7) Otherwise $\varrho_j a$, $\varrho_j b$ are not in simple comparison form. By Lemma 7 and by the definition of simple comparison form, there is some opponent pair $x_k \overline{r}$, b in $\varrho_j a$ and $\varrho_j b$ (after possibly swapping u and v) where either the occurrence of x_k is not ω-simple (Definition 8) or else x_k has another ω-simple occurrence in the body of $\varrho_j a$. Then $\varrho_j a$ satisfies the hypotheses of Lemma 17 (iteration) and we are done.

u and v is a flex-rigid pair: Without loss of generality, we assume that a is flex and b is rigid.

(8) Suppose first that $\varrho_j a$ is projective. By one of the induction invariants, a cannot be an identification variable. Thus Lemma 21 (Huet-style projection) applies.

(9) Otherwise $\varrho_j a$ is not projective. The head of $\varrho_j a$ must be b because b is rigid, and ϱ_j unifies u and v. Since $\varrho_j a$ is not projective, that means that b is not a bound variable. Therefore, b must be a constant. Then Lemma 19 (imitation) applies.

We have now constructed a run $(E_0, \sigma_0) \longrightarrow (E_1, \sigma_1) \longrightarrow (E_2, \sigma_2) \longrightarrow \cdots$ of the procedure. This run cannot be infinite because because the measure $\mathrm{ord}(E_j, \varrho_j)$ strictly decreases as j increases. Hence, at some point we reach a j such that $E_j = \varnothing$ and $\varrho_0 \subseteq \varrho_j\, \sigma_j$. Therefore, $(E, \mathrm{id}) \longrightarrow^* (\varnothing, \sigma_j) \longrightarrow \sigma_j$ and $\varrho \subseteq \tau^{-1}\, \varrho_j\, \sigma_j$, completing the proof. ◀

5 A New Decidable Fragment

We discovered a new fragment that admits a finite CSU and a simple oracle. The oracle is based on work by Prehofer and the procedure PT [21], a modification of Huet's procedure. PT transforms an initial multiset of constraints E_0 by applying bindings ϱ. If there is a sequence $E_0 \Longrightarrow^{\varrho_1} \cdots \Longrightarrow^{\varrho_n} E_n$ such that E_n has only flex-flex constraints, we say that PT produces a preunifier $\sigma = \varrho_n \ldots \varrho_1$ with constraints E_n. A sequence fails if $E_n = \bot$. As in the previous section, we consider all terms to be $\alpha\beta\eta$-equivalence classes with the β-reduced η-long form as their canonical representative. Unlike previously, in this section we view unification constraints $s \overset{?}{=} t$ as ordered pairs.

The following rules, however, are stated modulo orientation. The PT transition rules, adapted for our presentation style, are as follows:

Deletion $\{s \overset{?}{=} s\} \uplus E \Longrightarrow^{\mathrm{id}} E$

Decomposition $\{\lambda\overline{x}.\, a\, \overline{s}_m \overset{?}{=} \lambda\overline{x}.\, a\, \overline{t}_m\} \uplus E \Longrightarrow^{\mathrm{id}} \{s_1 \overset{?}{=} t_1, \ldots, s_m \overset{?}{=} t_m\} \uplus E$

where a is rigid

Failure $\{\lambda\overline{x}.\, a\, \overline{s} \overset{?}{=} \lambda\overline{x}.\, b\, \overline{t}\} \uplus E \Longrightarrow^{\mathrm{id}} \bot$

where a and b are different rigid heads

Solution $\{\lambda\overline{x}.\, F\, \overline{x} \overset{?}{=} \lambda\overline{x}.\, t\} \uplus E \Longrightarrow^{\varrho} \varrho(E)$

where F does not occur in t, t does not have a flex head, and $\varrho = \{F \mapsto \lambda\overline{x}.\, t\}$

Imitation $\{\lambda\overline{x}.\, F\, \overline{s}_m \overset{?}{=} \lambda\overline{x}.\, \mathsf{f}\, \overline{t}_n\} \uplus E \Longrightarrow^{\varrho} \varrho(\{G_1\, \overline{s}_m \overset{?}{=} t_1, \ldots, G_n\, \overline{s}_m \overset{?}{=} t_n\} \uplus E)$

where $\varrho = \{F \mapsto \lambda\overline{x}_m.\, \mathsf{f}\, (G_1\, \overline{x}_m) \ldots (G_n\, \overline{x}_m)\}$, $\overline{G}_n$ are fresh variables of appropriate types

Projection $\{\lambda\overline{x}.\, F\, \overline{s}_m \overset{?}{=} \lambda\overline{x}.\, a\, \overline{t}\} \uplus E \Longrightarrow^{\varrho} \varrho(\{s_i\, (G_1\, \overline{s}_m) \ldots (G_j\, \overline{s}_m) \overset{?}{=} a\, \overline{t}\} \uplus E)$

where $\varrho = \{F \mapsto \lambda\overline{x}_m.\, x_i\, (G_1\, \overline{x}_m) \ldots (G_j\, \overline{x}_m)\}$, $\overline{G}_j$ are fresh variables of appropriate types

The grayed constraints are required to be selected by a given selection function S. We call S *admissible* if it prioritizes selection of constraints applicable for Failure and Decomposition, and of descendant constraints of Projection transitions with $j = 0$ (i.e., for x_i of base type), in that order of priority. In the remainder of this section we consider only admissible selection functions, an assumption that Prehofer also makes implicitly in his thesis.

The following Lemma states that PT is *complete for preunification*:

▶ **Lemma 22.** *Let ϱ be a unifier of a multiset of constraints E_0. Then PT produces a preunifier σ with constraints E_n, and there exists a unifier θ of E_n such that $\varrho \subseteq \theta\sigma$.*

Proof. This lemma is a refinement of Lemma 4.1.7 from Prehofer's PhD thesis [21], and this proof closely follows the proof of that lemma.

We prove the lemma by induction using a well-founded measure on (ϱ, E_0). The ordering is the lexicographic comparison of the following properties:

A sum of the abstraction-free sizes of the bindings in ϱ
B multiset of the sizes of constraints in E_0

Here, the *abstraction-free size* is inductively defined by $\mathrm{afsize}(F) = 1$; $\mathrm{afsize}(x) = 1$; $\mathrm{afsize}(\mathsf{f}) = 1$; $\mathrm{afsize}(s\,t) = \mathrm{afsize}(s) + \mathrm{afsize}(t)$; $\mathrm{afsize}(\lambda x.\, s) = \mathrm{afsize}(s)$.

If E_0 consists only of flex-flex constraints, then we can take an empty transition sequence (i.e., $\sigma = \mathrm{id}$) and $\theta = \varrho$. Otherwise, there exists a constraint that is selected by an admissible selection function. We show that for each such constraint, there is going to be a PT transition bringing us closer to the desired preunifier.

Let $E_0 = \{\, s \overset{?}{=} t \,\} \uplus E_0'$. Since $s \overset{?}{=} t$ is selected, at least one of s and t must have a rigid head. We distinguish several cases based on the form of $s \overset{?}{=} t$ (modulo the constraint order):

- $s \overset{?}{=} s$: in this case, Deletion applies. This transition does not alter ϱ, but removes an equation obtaining E_1 which is smaller than E_0 and still unifiable by ϱ. By induction hypothesis, the preunifier is reachable.
- $\lambda \overline{x}.\, a\, \overline{s}_m \overset{?}{=} \lambda \overline{x}.\, b\, \overline{t}_n$, where a and b are rigid: since ϱ is a unifier, then $a = b$ and $n = m$. Therefore, Decomposition applies. Similarly to the above case, we conclude that the preunifier is reachable.
- $\lambda \overline{x}.\, F\, \overline{x} \overset{?}{=} \lambda \overline{x}.\, t$ where t has a rigid head and F does not occur in t: Solution applies. Huet showed [12, proof of L5.1] that in this case $\varrho = \varrho\sigma_1$, where $\sigma_1 = \{F \mapsto \lambda \overline{x}.\, t\}$. Let $\varrho_1 = \varrho \setminus \{F \mapsto \varrho(F)\}$. Then $\varrho = \varrho_1\sigma_1$. Since ϱ unifies E_0', ϱ_1 unifies $E_1 = \sigma_1(E_0')$. Clearly, ϱ_1 is smaller than ϱ. Now we can apply induction hypothesis on ϱ_1 and E_1, from which we obtain a sequence $E_1 \Longrightarrow^{\sigma_2} \cdots \Longrightarrow^{\sigma_n} E_n$, θ which unifies E_n and $\sigma' = \sigma_n \ldots \sigma_2$, such that $\varrho_1 \subseteq \theta\sigma'$. Finally, it is clear that sequence $E_0 \Longrightarrow^{\sigma_1} E_1 \Longrightarrow^{\sigma_2} \cdots \Longrightarrow^{\sigma_n} E_n$, θ, and $\sigma = \sigma'\sigma_1$ are as wanted in the lemma statement.
- $\lambda \overline{x}.\, F\, \overline{s}_m \overset{?}{=} \lambda \overline{x}.\, a\, \overline{t}$ where a is rigid: depending on the value of ϱ, we can either take an Imitation or Projection step. In either case, we show that the order reduces. Let $\varrho(F) = \lambda \overline{x}_m.\, b\, \overline{u}_n$ where b is either a constant or a bound variable. If b is a constant, we choose the Imitation step; otherwise we choose the Projection step. In either case, $\sigma_1 = \{F \mapsto \lambda \overline{x}_m.\, b\, (G_1\, \overline{x}_m) \ldots (G_n\, \overline{x}_m)\}$. Then it is easy to check that $\varrho_1 = \varrho \setminus \{F \mapsto \lambda \overline{x}_m.\, b\, \overline{u}_n\} \cup \{G_1 \mapsto \lambda \overline{x}_m.\, u_1, \ldots, G_n \mapsto \lambda \overline{x}_m.\, u_n\}$ unifies E_1 created as the result of imitation or projection. Clearly, ϱ_1 has smaller abstraction-free size than ϱ. Therefore, by the induction hypothesis we obtain the transitions $E_1 \Longrightarrow^{\sigma_2} \cdots \Longrightarrow^{\sigma_n} E_n$ and substitutions θ and $\sigma' = \sigma_n \ldots \sigma_2$, where $\varrho_1 \subseteq \theta\sigma'$ and θ is a unifier of E_n. Our goal is to prove $\varrho \subseteq \theta\sigma'\sigma_1$. The only variable that ϱ maps and on which ϱ and ϱ_1 differ is F. Since ϱ_1 and $\theta\sigma'$ agree on $G_1, \ldots, G_n$, we conclude that ϱ and $\theta\sigma'\sigma_1$ agree on F. Therefore, by taking $\sigma = \sigma'\sigma_1$, and prepending E_0 to the sequence from the induction hypothesis, we show that the preunifier is reachable. ◀

Prehofer showed that PT terminates for some classes of constraints. We call a term *linear* if no free variable has repeated occurrences in it. We call a term *solid* if its free variables are applied either to bound variables or ground base-type terms. We call it *strictly solid*[1] if its free variables are applied either to bound variables or second-order ground base-type terms. For example, if G, a, and x are of base type, and F, H, g, and y are binary, the terms $F\,G\,\mathsf{a}$, and $H\,(\lambda x.\,x)\,\mathsf{a}$ are not solid; $\lambda x.\,F\,x\,x$ is strictly solid; $F\,\mathsf{a}\,(\mathsf{g}\,(\lambda y.\,y\,\mathsf{a}\,\mathsf{a})\,\mathsf{a})$ is solid, but not strictly. Prehofer's thesis states that PT terminates on $\{s \overset{?}{=} t\}$ if s is linear, s shares no free variables with t, s is strictly solid, and t is second-order.

We extend this result in Theorem 28 along two axes: we create an oracle for the full unification problem, and we lift some order constraints by requiring s and t to be solid.

As a first step, we consider the matching problem. A *matching problem* is a unification problem $\{s \overset{?}{=} t\}$ where t is ground. In what follows, we establish some useful properties of matching problems in which both s and t are solid (*solid matching problems*). In the remaining of this section, whenever we compare multisets, we use the multiset ordering defined by Dershowitz and Manna [9].

▶ **Lemma 23.** *PT terminates on a solid matching problem* $\{s \overset{?}{=} t\}$.

Proof. We show termination by designing a measure on a matching problem E that decreases with each application of a PT transition (possibly followed by Decomposition or Failure steps). Our measure function compares the following properties lexicographically:

A multiset of the sizes of right-hand sides of the constraints in E
B number of free variables in E

Clearly, when applying PT transitions, the problem stays a matching problem because the applied bindings do not introduce free variables on the right-hand sides. It is also easy to check that each applied binding keeps the terms in the solid fragment. Namely, bindings for both Imitation and Projection transitions are patterns, which means that, after applying the binding, fresh free variables are applied only to bound variables or ground base-type terms. The Solution transition effectively replaces the variable with a ground term, which is obviously solid. We show each transition either trivially terminates or reduces the measure:

Deletion A decreases.
Decomposition A decreases.
Failure Trivial – represents a terminal node.
Solution This transition applies on constraints of the form $F \overset{?}{=} t$. This reduces A, since the constraint $F \overset{?}{=} t$ is removed and F cannot appear on the right-hand side.
Imitation The rule is applicable only on $\lambda\overline{x}.\,F\,\overline{s}_m \overset{?}{=} \lambda\overline{x}.\,\mathsf{f}\,\overline{t}_n$. The Imitation transition replaces the constraint with $\{H_i\,\overline{s}_m \overset{?}{=} \overline{t}_i \mid 1 \leq i \leq n\}$. This reduces A, as F does not appear on any right-hand side.
Projection The rule is applicable only on $\lambda\overline{x}.\,F\,\overline{s}_m \overset{?}{=} \lambda\overline{x}.\,a\,\overline{t}_n$. If a is a bound variable, and we project F to argument s_i different than a, Failure applies and PT trivially terminates. If we project F to a, we apply Decomposition, which reduces A. If a is a constant, for Failure not to apply, we have to project to a base-type ground term s_i. This does not increase A, since no variables appear on right-hand sides, but removes the variable F from E, reducing B by one. ◀

[1] In Prehofer's thesis, striclty solid terms are called weakly second-order.

We say σ is a grounding substitution if for every variable F mapped by σ, σF is ground.

▶ **Lemma 24.** *All unifiers produced by PT for the solid matching problem $\{s \overset{?}{=} t\}$ are grounding substitutions.*

Proof. Closely following the proof of Lemma 5.2.5 in Prehofer's PhD thesis [21], we prove our claim by induction on the length of the PT transition sequence that leads to the unifier. We know this sequence is finite by Lemma 23. The base case of induction, for the empty sequence, is trivial. The induction step is made using one of the following transitions:

Deletion Trivial.
Decomposition Trivial.
Failure This rule is not relevant, since it will not lead to the unifier.
Solution This rule applies the substitution $\{F \mapsto t\}$, which is ground since t is ground.
Imitation This transition applies on constraints of the form

$$\lambda\overline{x}.\, F\,\overline{s}_k \overset{?}{=} \lambda\overline{x}.\, \mathsf{f}\,\overline{t}_l$$

The binding for Imitation is $\varrho = \{F \mapsto \lambda\overline{x}_k.\, \mathsf{f}\,(G_1\,\overline{s}_k)\ldots(G_l\,\overline{s}_k)\}$, and reduces the problem to $\{G_i\,\overline{s}_k \overset{?}{=} t_i \mid 1 \leq i \leq l\}$. Since the right-hand sides are ground, any unifier σ produced by PT must map all of the variables $G_1,\ldots,G_l$. By induction hypothesis, $\sigma(G_i)$ is ground. Therefore, $\sigma\varrho$ must map F to a ground term.

Projection This transition applies on constraints of the form

$$\lambda\overline{x}.\, F\,\overline{s}_k \overset{?}{=} \lambda\overline{x}.\, t$$

The binding for Projection is $\varrho = \{F \mapsto \lambda\overline{x}_k.\, x_i\,(G_1\,\overline{x}_k)\ldots(G_j\,\overline{x}_k)\}$, and reduces the problem to $\{s_i\,(G_1\,\overline{s}_k)\ldots(G_j\,\overline{s}_k) \overset{?}{=} a\,\overline{t}_l\}$. Since the right-hand side is ground, any unifier σ produced by PT must map all of the variables $G_1,\ldots,G_j$. By induction hypothesis, $\sigma(G_i)$ is ground. Therefore, $\sigma\varrho$ must map F to a ground term. ◀

▶ **Lemma 25.** *If s and t are solid, s is linear and shares no free variables with t, PT terminates for the preunification problem $\{s \overset{?}{=} t\}$, and all remaining flex-flex constraints are solid.*

Proof. This lemma is a modification of Lemma 5.2.1 and Theorem 5.2.6 from Prehofer's PhD thesis [21]. Correspondingly, the following proof closely follows the proofs for these two lemmas. We also adopt the notion of an *isolated* variable from Prehofer's thesis: a variable is isolated in a multiset E of constraints if it appears exactly once in E.

First, similarly to the proof of previous lemma we conclude each transition maintains the condition that the terms remain solid. Second, we have to show that variables on left-hand sides remain isolated. For Imitation and Projection rules, the preservation of this invariant is obvious. Later, we also show that Solution preserves it. Third, since s and t share no variables, and s is linear, no rule can introduce a variable from the right-hand side to the left-hand side.

To prove termination of PT, we devise a measure that decreases with each application of a PT transition. The measure lexicographically compares the following properties:

A number of occurrences of constant symbols and bound variables that are not below free variables on left-hand sides
B number of free variables on right-hand sides
C multiset of the sizes of right-hand sides

We show that each transition either trivially terminates or reduces the measure:

Deletion A does not increase and at least one of A or B reduces.

Decomposition A reduces.

Failure Trivial.

Solution This rule applies in two cases:

- $F \overset{?}{=} t$: since F is isolated, A is unchanged, B is not increased, and C reduces. All variables on left-hand sides remain isolated since they are unaffected by this substitution.
- $t \overset{?}{=} F$: since F does not occur on any left-hand side and the head of t must be a constant or bound variable (otherwise the rule would not be applicable), A reduces. Even though t might contain some free variables and F can have multiple occurrences on right-hand sides, all free variables on left-hand sides remain isolated. Namely, all free variables in t occur exactly once in the multiset, and since $t \overset{?}{=} F$ is removed, all of them either disappear or end up on right-hand sides.

Imitation We distinguish the following two forms of the selected constraint:

- $\lambda\overline{x}.\, F\,\overline{v}_n \overset{?}{=} \lambda\overline{x}.\, \mathsf{f}\,\overline{u}_m$: We replace this constraint by $\{H_i\,\overline{v}_n \overset{?}{=} u_i \mid 1 \leq i \leq m\}$ and apply the Imitation binding $F \mapsto \lambda\overline{x}_n.\, \mathsf{f}\,(H_1\,\overline{x}_n)\ldots(H_m\,\overline{x}_n)$. As F is isolated, A and B do not increase. Since F is isolated, it does not occur on any right-hand side, and hence C decreases.
- $\lambda\overline{x}.\, \mathsf{f}\,\overline{u}_m \overset{?}{=} \lambda\overline{x}.\, F\,\overline{v}_n$: applying the Imitation transition as above will reduce A since F cannot appear on any left-hand side.

Projection Similarly to the previous transition rule, we have two cases:

- $\lambda\overline{x}.\, F\,\overline{v}_n \overset{?}{=} \lambda\overline{x}.\, a\,\overline{u}_m$: if a is a bound variable, then for Failure not to be applicable afterwards, we have to project F onto argument v_i equal to a. Then we proceed like for Imitation. If a is not a bound variable, then we have to project to some base-type term v_j (otherwise Failure would be applicable afterwards). This reduces the problem to $\lambda\overline{x}.\, v_j \overset{?}{=} \lambda\overline{x}.\, a\,\overline{u}_m$. This is a solid matching problem, whose solutions computed by PT are grounding substitutions (see Lemma 24). Applying one of those solutions will eliminate all the free variables in $\lambda\overline{x}.\, a\,\overline{u}_m$. Since PT is parametrized by an admissible selection function, we know that there are no constraints descending from a simple projection in E since the constraint $\lambda\overline{x}.\, F\,\overline{v}_n \overset{?}{=} \lambda\overline{x}.\, a\,\overline{u}_m$ was chosen, which, due to solidity restrictions, cannot be such a descendant. Therefore, we know that PT will transform the descendants of the matching problem $\lambda\overline{x}.\, v_j \overset{?}{=} \lambda\overline{x}.\, a\,\overline{u}_m$ until either Failure is observed (making PT trivially terminating) or until no descendant exists and the grounding matcher is computed (see Lemmas 24 and 23). This results in removal of the original constraint $\lambda\overline{x}.\, F\,\overline{v}_n \overset{?}{=} \lambda\overline{x}.\, a\,\overline{u}_m$ and application of the computed grounding matcher which will either remove all the free variables in the right-hand side of the constraint (not increasing A and reducing B) or not increasing A and B, reduce C if no free variables occur in the right-hand side.
- $\lambda\overline{x}.\, a\,\overline{v}_n \overset{?}{=} \lambda\overline{x}.\, F\,\overline{u}_m$: if a is a bound variable, projecting F onto argument u_i will either enable application of Decomposition as the next step reducing A, or it will result in Failure, trivially terminating. If a is a constant, then projecting F onto some u_j will either yield Failure or enable Decomposition, reducing A. ◀

Enumerating a CSU for a solid flex-flex pair may seem as hard as for any other flex-flex pair; however, the following two lemmas show that solid pairs admit an MGU:

▶ **Lemma 26.** *The unification problem $\{\lambda\overline{z}.\, F\,\overline{s}_m \overset{?}{=} \lambda\overline{z}.\, F\,\overline{s}'_{m'}\}$, where both terms are solid, has an MGU of the form $\sigma = \{F \mapsto \lambda\overline{x}_m.\, G\,x_{j_1}\ldots x_{j_r}\}$ where G is a fresh variable, and $1 \leq j_1 < \cdots < j_r \leq m$ are exactly those indices j_i for which $s_{j_i} = s'_{j_i}$.*

Proof. Let ϱ be a unifier for the given unification problem. Let $\lambda\overline{x}.\, u = \varrho(F)$. Take an arbitrary subterm of u whose head is a bound variable x_i. If x_i is of function type, it corresponds to either s_i or s'_i which, due to solidity restrictions, has to be a bound variable. Furthermore, since ϱ is a unifier, s_i and s'_i have to be the syntactically equal. Similarly, if x_i is of base type, it corresponds to two ground terms s_i and s'_i which have to be syntactically equal. We conclude that ϱ can use variables from $\overline{x}_n$ only if they correspond to syntactically equal terms. Therefore, there is a substitution θ such that $\varrho \subseteq \theta\sigma$. Due to arbitrary choice of ϱ, we conclude that σ is an MGU. ◀

▶ **Lemma 27.** *Let $\{\lambda\overline{x}.\, F\,\overline{s}_m \overset{?}{=} \lambda\overline{x}.\, F'\,\overline{s}'_{m'}\}$ be a solid unification problem where $F \neq F'$. Then there is a finite CSU $\{\sigma_i^1, \ldots, \sigma_i^{k_i}\}$ of the problem $\{s_i \overset{?}{=} H_i\,\overline{s}'_{m'}\}$, where H_i is a fresh free variable. Let $\lambda\overline{y}_{m'}.\, s_i^j = \lambda\overline{y}_{m'}.\, \sigma_i^j(H_i)\,\overline{y}_{m'}$. Similarly, there is a finite CSU $\{\tilde{\sigma}_i^1, \ldots, \tilde{\sigma}_i^{l_i}\}$ of the problem $\{s'_i \overset{?}{=} \tilde{H}_i\,\overline{s}_m\}$, where $\tilde{H}_i$ is a fresh free variable. Let $\lambda\overline{x}_m.\, s'^j_i = \lambda\overline{x}_m.\, \tilde{\sigma}_i^j(\tilde{H}_i)\,\overline{x}_m$. These finite CSUs exist by Lemma 23. Let Z be a fresh free variable. An MGU σ for the given problem is*

$$F \mapsto \lambda\overline{x}_m.\, Z\, \underbrace{x_1\,\ldots\,x_1}_{k_1\ \textit{times}} \ldots \underbrace{x_m\,\ldots\,x_m}_{k_m\ \textit{times}}\; s'^1_1 \ldots s'^{l_1}_1 \;\ldots\; s'^1_{m'} \ldots s'^{l_{m'}}_{m'}$$

$$F' \mapsto \lambda\overline{y}_{m'}.\, Z\, s_1^1 \ldots s_1^{k_1} \ldots s_m^1 \ldots s_m^{k_m}\; \underbrace{y_1\,\ldots y_1}_{l_1\ \textit{times}} \ldots \underbrace{y_{m'}\,\ldots y_{m'}}_{l_{m'}\ \textit{times}}$$

Proof. Let ϱ be an arbitrary unifier for the problem $\lambda\overline{x}.\, F\,\overline{s}_m \overset{?}{=} \lambda\overline{x}.\, F'\,\overline{s}'_{m'}$. We prove σ is an MGU by showing that there exists a substitution θ such that $\varrho \subseteq \theta\sigma$. Without loss of generality, we can focus only on the values of ϱF and $\varrho F'$; F and F' are the only free variables that appear in the original problem and for all other variables mapped by ϱ, θ can be chosen to coincide with ϱ.

Let $\lambda\overline{x}_m.\, u = \varrho F$ and $\lambda\overline{y}_{m'}.\, u' = \varrho F'$, where the bound variables $\overline{x}_m$ and $\overline{y}_{m'}$ have been α-renamed apart. We also assume that the names of variables bound inside u and u' are α-renamed so that they are different from $\overline{x}_m$ and $\overline{y}'_m$. Finally, bound variables from the definition of σ have been α-renamed to match $\overline{x}_m$ and $\overline{y}_{m'}$. We define θ to be the substitution

$$\theta = \{Z \mapsto \lambda z_1^1 \ldots z_1^{k_1} \ldots z_m^1 \ldots z_m^{k_m}\, w_1^1 \ldots w_1^{l_1} \ldots w_{m'}^1 \ldots w_{m'}^{l_{m'}}.\, \mathsf{diff}(u, u')\}$$

where $\mathsf{diff}(v, v')$ is defined recursively as

$$\mathsf{diff}(\lambda x.\, v, \lambda y.\, v') = \lambda x.\, \mathsf{diff}(v, \{y \mapsto x\}v') \quad (1)$$
$$\mathsf{diff}(a\,\overline{v}_n, a\,\overline{v'}_n) = a\,\mathsf{diff}(v_1, v'_1) \ldots \mathsf{diff}(v_n, v'_n) \quad (2)$$
$$\mathsf{diff}(x_i, v') = z_i^k, \text{ if } v' = s_i^k \quad (3)$$
$$\mathsf{diff}(v, y_i) = w_i^l, \text{ if } v = s'^l_i \quad (4)$$
$$\mathsf{diff}(x_i\,\overline{v}_n, y_j\,\overline{v'}_n) = z_i^k\,\mathsf{diff}(v_1, v'_1) \ldots \mathsf{diff}(v_n, v'_n), \text{ if } y_j = s_i^k \quad (5)$$

From diff's definition it is clear that there are terms v, v' for which it is undefined. However, we will show that for each u and u' which are bodies of bindings from a unifier ϱ, diff is defined and has the desired property. In equations 3, 4, and 5, if there are multiple numbers k or l that fulfill the condition, choose an arbitrary one. We need to show that

$\varrho \subseteq \theta\sigma$, i.e., $\varrho F = \theta\sigma F$ and $\varrho F' = \theta\sigma F'$. By the definitions of u, u', θ and σ and β-reduction, this is equivalent to

$$\lambda\overline{x}_m.\, u = \lambda\overline{x}_m.\, \{z_i^k \mapsto x_i, w_i^l \mapsto {s'}_i^l \text{ for all } k,l,i\}\, \mathsf{diff}(u,u')$$
$$\lambda\overline{y}'_m.\, u' = \lambda\overline{y}'_m.\, \{z_i^k \mapsto s_i^k, w_i^l \mapsto y_i \text{ for all } k,l,i\}\, \mathsf{diff}(u,u')$$

We will show by induction that for any $\lambda\overline{x}_m.\, v$, $\lambda\overline{y}_m.\, v'$, such that

$$\{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\}v = \{y_1 \mapsto s'_1, \ldots, y_{m'} \mapsto s'_{m'}\}v' \tag{$\star$}$$

we have

$$\begin{aligned} v &= \{z_i^k \mapsto x_i, w_i^l \mapsto {s'}_i^l \text{ for all } k,l,i\}\, \mathsf{diff}(v,v') \\ v' &= \{z_i^k \mapsto s_i^k, w_i^l \mapsto y_i \text{ for all } k,l,i\}\, \mathsf{diff}(v,v') \end{aligned} \tag{$\dagger$}$$

The equation ($\star$) holds for $v = u$ and $v' = u'$ because ϱ is a unifier of $\lambda\overline{x}.\, F\,\overline{s}_m \overset{?}{=} \lambda\overline{x}.\, F'\,\overline{s}'_{m'}$. Therefore, once we have shown that ($\star$) implies ($\dagger$), we know that ($\dagger$) holds for $v = u$ and $v' = u'$ and we are done.

We prove that ($\star$) implies ($\dagger$) by induction on the size of v and v'. We consider the following cases:

$v = \lambda x.\, v_1$. For ($\star$) to hold v and v' must be of the same type. Therefore, the λ-prefixes of their η-long representatives must have the same length and we can apply equation 1. By the induction hypothesis, ($\dagger$) holds.

$v = x_i$. In this case, $\{x_1 \mapsto s_1, \ldots, x_m \mapsto s_m\}v = s_i$. Since ($\star$) holds, v' must be an instance of a unifier from the CSU of $s_i = H_i\,\overline{s}'_{m'}$. However, since s_i and all terms in $\overline{s}'_{m'}$ are ground, $\lambda\overline{y}_m.\, v' = \sigma_i^k(H_i)$, for some k. Then, $\mathsf{diff}(x_i, v') = z_i^k$, and it is easy to check that ($\dagger$) holds.

$v = x_i\,\overline{v}_n, n > 0$. In this case, x_i is mapped to s_i which, due to solidity restrictions, has to be a functional bound variable. Since ($\star$) holds, we conclude that the head of $\{y_1 \mapsto s'_1, \ldots, y_{m'} \mapsto s'_{m'}\}v'$ must be s'_j, such that $s'_j = s_i$; this also means that $v' = y_j\,\overline{v'}_n$. Therefore, it is easy to check that some $\tau = \{H_i \mapsto \lambda\overline{y}_{m'}.\, y_j\}$ is a matcher for the problem $s_i = H_i\,\overline{s}'_{m'}$. For some k, $\sigma_i^k = \tau$, i.e., $\mathsf{diff}(v,v') = z_i^k\, \mathsf{diff}(v_1,v'_1)\ldots\mathsf{diff}(v_n,v'_n)$. By induction hypothesis, we get that ($\dagger$) holds.

$v = a\,\overline{v}_n$, where a is a free variable, a loose bound variable different than $x_1, \ldots, x_m$, or a constant. If a is a free variable or a loose bound variable different than $x_1, \ldots, x_m$, then $v' = a\,\overline{v'}_n$, since ($\star$) holds, all of $\overline{s}'_{m'}$ are ground, and bound variables different than $x_1, \ldots, x_m$ and $y_1, \ldots, y_{m'}$ are renamed to match by equation 1. By the induction hypothesis and by equation 2, we obtain ($\dagger$). If a is a constant, we consider two cases: either $v' = a\,\overline{v'}_n$ which allows us to apply the induction hypothesis and obtain ($\dagger$) as above, or $v' = y_j\,\overline{v'}_m$. Since a is a constant, y_j cannot be a functional bound variable, since then it would be mapped to s_j, which due to solidity restrictions also has to be a functional bound variable and ($\star$) would not hold. Therefore $v' = y_j$. In this case, we proceed as in the case $v = x_i$ with the roles of v and v' swapped. ◀

▶ **Theorem 28.** *Let s and t be solid terms that share no free variables, and let s be linear. Then the unification problem $\{s \overset{?}{=} t\}$ has a finite CSU.*

Proof. By Lemma 25, PT terminates on $\{s \overset{?}{=} t\}$ with a finite set of preunifiers σ, each associated with a multiset E of solid flex-flex pairs.

An MGU δ_E of the remaining multiset E of solid flex-flex constraints can be found as follows. Choose one constraint $(u \overset{?}{=} v) \in E$ and determine an MGU ϱ for it using Lemma 26 or 27. Then the set $\varrho(E \setminus \{u \overset{?}{=} v\})$ also contains only solid flex-flex constraints, and we iterate this process by choosing a constraint from $\varrho(E \setminus \{u \overset{?}{=} v\})$ next until there are no constraints left, eventually yielding an MGU ϱ' of $\varrho(E \setminus \{u \overset{?}{=} v\})$. Finally, we obtain the MGU $\delta_E = \varrho'\varrho$ of E.

Let $U = \{\delta_E\sigma \mid \text{PT produces preunifier } \sigma \text{ with constraints } E\}$. By termination of PT, U is finite. We show that U is a CSU. Let ϱ be an arbitrary unifier for $\{s \overset{?}{=} t\}$. By Lemma 22, PT produces a preunifier σ with flex-flex constraints E such that there is a unifier θ of E and $\varrho \subseteq \theta\sigma$. Since δ_E is an MGU of E, there exists a substitution θ' such that $\theta \subseteq \theta'\delta_E$. If we make sure that the fresh variables introduced by σ and δ_E are distinct, it follows that $\varrho \subseteq \theta'\delta_E\sigma$. Therefore, U is a CSU. ◀

The proof of Theorem 28 provides an effective way to calculate a CSU using PT and the results of Lemmas 26 and 27.

▶ **Example 29.** For example, let $\{F\,(\mathsf{f}\,\mathsf{a}) \overset{?}{=} \mathsf{g}\,\mathsf{a}\,(G\,\mathsf{a})\}$ be the unification problem to solve. Projecting F onto the first argument will lead to a nonunifiable problem, so we perform imitation of g building a binding $\sigma_1 = \{F \mapsto \lambda x.\,\mathsf{g}\,(F_1\,x)\,(F_2\,x)\}$. This yields the problem $\{F_1\,(\mathsf{f}\,\mathsf{a}) \overset{?}{=} \mathsf{a}, F_2\,(\mathsf{f}\,\mathsf{a}) \overset{?}{=} G\,\mathsf{a}\}$. Again, we can only imitate a for F_1 – building a new binding $\sigma_2 = \{F_1 \mapsto \lambda x.\,\mathsf{a}\}$. Finally, this yields the problem $\{F_2\,(\mathsf{f}\,\mathsf{a}) \overset{?}{=} G\,\mathsf{a}\}$. According to Lemma 27, we find CSUs for the problems $J_1\,\mathsf{a} = \mathsf{f}\,\mathsf{a}$ and $I_1\,(\mathsf{f}\,\mathsf{a}) \overset{?}{=} \mathsf{a}$ using PT. The latter problem has a singleton CSU $\{I_1 \mapsto \lambda x.\,\mathsf{a}\}$, whereas the former has a CSU containing $\{J_1 \mapsto \lambda x.\,\mathsf{f}\,x\}$ and $\{J_1 \mapsto \lambda x.\,\mathsf{f}\,\mathsf{a}\}$. Combining these solutions, we obtain an MGU $\sigma_3 = \{F_2 \mapsto \lambda x.\,H\,x\,x\,\mathsf{a},\ G \mapsto \lambda x.\,H\,(\mathsf{f}\,\mathsf{a})\,(\mathsf{f}\,x)\,x\}$ for $F_2\,(\mathsf{f}\,\mathsf{a}) \overset{?}{=} G\,\mathsf{a}$. Finally, we get the MGU $\sigma = \sigma_3\sigma_2\sigma_1 = \{F \mapsto \lambda x.\,\mathsf{g}\,\mathsf{a}\,(H\,x\,x\,\mathsf{a}),\ G \mapsto \lambda x.\,H\,(\mathsf{f}\,\mathsf{a})\,(\mathsf{f}\,x)\,x\}$ of the original problem.

Small examples that violate conditions of Theorem 28 and admit only infinite CSUs can be found easily. The problem $\{\lambda x.\,F\,(\mathsf{f}\,x) \overset{?}{=} \lambda x.\,\mathsf{f}\,(F\,x)\}$ violates variable distinctness and is a well-known example of a problem with only infinite CSUs. Similarly, $\lambda x.\,\mathsf{g}\,(F\,(\mathsf{f}\,x))\,F \overset{?}{=} \lambda x.\,\mathsf{g}\,(\mathsf{f}\,(G\,x))\,G$, which violates linearity, reduces to the previous problem. Only ground arguments to free variables are allowed because $\{F\,X \overset{?}{=} G\,\mathsf{a}\}$ has only infinite CSUs. Finally, it is crucial that functional arguments to free variables are only bound variables: the problem $\{\lambda y.\,X(\lambda x.\,x)\,y \overset{?}{=} \lambda y.\,y\}$ has only infinite CSUs.

6 An Extension of Fingerprint Indexing

A fundamental building block for almost all automated reasoning tools is the operation of retrieving term pairs that satisfy certain conditions, e.g., unifiable terms, instances or generalizations. Indexing data structures are used to implement this operation efficiently. If the data structure retrieves precisely the terms that satisfy the condition it is called *perfect*; otherwise, it is called *imperfect*.

Higher-order indexing has received little attention, compared to its first-order counterpart. However, recent research in higher-order theorem proving increased the interest in higher-order indexing [4,17]. A *fingerprint index* [24,30] is an imperfect index based on the idea that the skeleton of the term consisting of all non-variable positions is not affected by substitutions. Therefore, we can easily determine that terms are not unifiable (or matchable) if they disagree on a fixed set of sample positions.

More formally, when we sample an untyped first-order term t on a sample position p, the *generic fingerprinting function* gfpf distinguishes four possibilities:

$$\mathrm{gfpf}(t,p) = \begin{cases} \mathsf{f} & \text{if } t|_p \text{ has a symbol head } \mathsf{f} \\ \mathsf{A} & \text{if } t|_p \text{ is a variable} \\ \mathsf{B} & \text{if } t|_q \text{ is a variable for some proper prefix } q \text{ of } p \\ \mathsf{N} & \text{otherwise} \end{cases}$$

We define the *fingerprinting function* $\mathrm{fp}(t) = (\mathrm{gfpf}(t, p_1), \ldots, \mathrm{gfpf}(t, p_n))$, based on a fixed tuple of positions $\overline{p}_n$. Determining whether two terms are compatible for a given retrieval operation reduces to checking their fingerprints' componentwise compatibility. The following matrices determine the compatibility for retrieval operations:

	f_1	f_2	A	B	N
f_1		✗			✗
A					✗
B					
N	✗	✗	✗		

	f_1	f_2	A	B	N
f_1		✗	✗	✗	✗
A				✗	✗
B					
N	✗	✗	✗	✗	

The left matrix determines unification compatibility, while the right matrix determines compatibility for matching term s (rows) onto term t (columns). Symbols f_1 and f_2 stand for arbitrary distinct constants. Incompatible features are marked with ✗. For example, given a tuple of term positions $(1, 1.1.1, 2)$, and terms $\mathsf{f}(\mathsf{g}(X), \mathsf{b})$ and $\mathsf{f}(\mathsf{f}(\mathsf{a}, \mathsf{a}), \mathsf{b})$, their fingerprints are $(\mathsf{g}, \mathsf{B}, \mathsf{b})$ and $(\mathsf{f}, \mathsf{N}, \mathsf{b})$, respectively. Since the first fingerprint component is incompatible, terms are not unifiable.

Fingerprints for the terms in the index are stored in a trie data structure. This allows us to efficiently filter out terms that are not compatible with a given retrieval condition. For the remaining terms, a unification or matching algorithm must be invoked to determine whether they satisfy the condition or not.

The fundamental idea of first-order fingerprint indexing carries over to higher-order terms – application of a substitution does not change the rigid skeleton of a term. However, to extend fingerprint indexing to higher-order terms, we must address the issues of $\alpha\beta\eta$-normalization and the fact that we can sample two new kinds of terms – λ-abstractions and bound variables. To that end, we define a function $\lfloor t \rfloor$, defined on β-reduced terms in De Bruijn [6] notation:

$$\lfloor F\,\overline{s} \rfloor = F \qquad \lfloor x_i\,\overline{s}_n \rfloor = \mathsf{db}^{\alpha}_{i}(\lfloor s_1 \rfloor, \ldots, \lfloor s_n \rfloor) \qquad \lfloor \mathsf{f}\,\overline{s}_n \rfloor = \mathsf{f}(\lfloor s_1 \rfloor, \ldots, \lfloor s_n \rfloor) \qquad \lfloor \lambda\overline{x}.\, s \rfloor = \lfloor s \rfloor$$

We let x_i be a bound variable of type α with De Bruijn index i, and db^{α}_{i} be a fresh constant corresponding to this variable. All db^{α}_{i} must be different from constants that do not represent De Bruijn indices. Effectively, $\lfloor\ \rfloor$ transforms a β-reduced η-long higher-order term to an untyped first-order term. Let $t_{\downarrow\beta\eta}$ be the β-reduced η-long form of t; the higher-order generic fingerprinting function $\mathrm{gfpf}_{\mathsf{ho}}$, which relies on conversion $\langle t \rangle_{\mathsf{db}}$ from named to De Bruijn representation, is defined as

$$\mathrm{gfpf}_{\mathsf{ho}}(t, p) = \mathrm{gfpf}(\lfloor \langle t_{\downarrow\beta\eta} \rangle_{\mathsf{db}} \rfloor, p)$$

If we define $\mathrm{fp}_{\mathsf{ho}}(t) = \mathrm{fp}(\lfloor \langle t_{\downarrow\beta\eta} \rangle_{\mathsf{db}} \rfloor)$, we can support fingerprint indexing for higher-order terms with no changes to the compatibility matrices. For example, consider the terms $s = (\lambda xy.\, x\, y)\, \mathsf{g}$ and $t = \mathsf{f}$, where g has the type $\alpha \to \beta$ and f has the type $\alpha \to \alpha \to \beta$. For the tuple of positions $(1, 1.1.1, 2)$ we get

$$\mathrm{fp}_{\mathsf{ho}}(s) = \mathrm{fp}(\lfloor \langle s_{\downarrow\beta\eta} \rangle_{\mathsf{db}} \rfloor) = \mathrm{fp}(\mathsf{g}(\mathsf{db}^{\alpha}_{0})) = (\mathsf{db}^{\alpha}_{0}, \mathsf{N}, \mathsf{N})$$
$$\mathrm{fp}_{\mathsf{ho}}(t) = \mathrm{fp}(\lfloor \langle t_{\downarrow\beta\eta} \rangle_{\mathsf{db}} \rfloor) = \mathrm{fp}(\mathsf{f}(\mathsf{db}^{\alpha}_{1}, \mathsf{db}^{\alpha}_{0})) = (\mathsf{db}^{\alpha}_{1}, \mathsf{N}, \mathsf{db}^{\alpha}_{0})$$

Since the first and third fingerprint component are incompatible, the terms are not unifiable.

Other first-order indexing techniques such as feature vector indexing and substitution trees can probably be extended to higher-order logic using the method described here as well.

7 Implementation

Zipperposition [7, 8] is an open-source[2] theorem prover written in OCaml. It is a versatile testbed for prototyping extensions to superposition-based theorem provers. It was initially designed as a prover for polymorphic first-order logic and then extended to higher-order logic. The most recent addition is a complete mode for Boolean-free higher-order logic [2], which depends on a unification procedure that can enumerate a CSU. We implemented our procedure in Zipperposition.

We used OCaml's functors to create a modular implementation. The core of our procedure is implemented in a module which is parametrized by another module providing oracles and implementing the Bind step. This way we can obtain the full or pragmatic procedure and seamlessly integrate oracles while reusing as much common code as possible.

To enumerate all elements of a possibly infinite CSU, we rely on lazy lists whose elements are subsingletons of unifiers (either one-element set containing a unifier or an empty set). The search space must be explored in a *fair* manner, meaning that no branch of the constructed tree is indefinitely postponed.

Each Bind step will give rise to new a unification problem $p_1, p_2, \ldots$ to be solved. Solutions to each of those problems are new lazy lists. To avoid postponing some unifier indefinitely, we first take one subsingleton from p_1, then one from each of p_1 and p_2. We continue with one subsingleton from p_1, p_2 and p_3, and so on. Empty lazy lists are ignored in the traversal. To ensure we do not remain stuck waiting for a unifier from a particular lazy list, the procedure will periodically return an empty set, indicating that the next lazy list should be probed.

The implemented selection function for our procedure prioritizes selection of rigid-rigid over flex-rigid pairs, and flex-rigid over flex-flex pairs. However, since the constructed substitution σ is not applied eagerly, heads can appear to be flex, even if they become rigid after dereferencing and normalization. To mitigate this issue to some degree, we dereference the heads with σ, but do not normalize, and use the resulting heads for prioritization.

We implemented oracles for the pattern, solid, and fixpoint fragment. Fixpoint unification [12] is concerned with problems of the form $\{F \overset{?}{=} t\}$. If F does not occur in t, $\{F \mapsto t\}$ is an MGU for the problem. If there is a position p in t such that $t|_p = F\,\overline{u}_m$ and for each prefix $q \neq p$ of p, $t|_q$ has a rigid head and either $m = 0$ or t is not a λ-abstraction, then we can conclude that $F \overset{?}{=} t$ has no solutions. Otherwise, the fixpoint oracle is not applicable.

8 Evaluation

We evaluated the implementation of our unification procedure in Zipperposition, assessing a complete variant and a pragmatic variant, the latter with several different combinations of limits for number of bindings. As part of the implementation of the complete mode for Boolean-free higher-order logic in Zipperposition [2], Bentkamp implemented a straightforward version of JP procedure. This version is faithful to the original description, with a check as to whether a (sub)problem can be solved using a first-order oracle as the only optimization.

[2] `https://github.com/sneeuwballen/zipperposition`

	old	cv	pv^{12}_{6666}	pv^{6}_{3333}	pv^{4}_{2222}	pv^{2}_{1222}	pv^{2}_{1121}	pv^{2}_{1020}
TPTP	1551	1717	1722	**1732**	**1732**	1715	1712	1719
SH	242	**260**	253	255	255	254	259	257

Figure 1 Proved problems, per configuration

	n	f	p	s	fp	fs	ps	fps
TPTP	1658	1717	1717	1720	1719	**1724**	1720	1723
SH	245	255	**260**	259	255	254	258	254

Figure 2 Proved problems, per used oracle

Our evaluations were performed on StarExec Miami [27] servers with Intel Xeon E5-2620 v4 CPUs clocked at 2.10 GHz with 60 s CPU limit.

Contrary to first-order unification, there is no widely available corpus of benchmarks dedicated solely to evaluating performance of higher-order unification algorithms. Thus, we used all 2606 monomorphic higher-order theorems from the TPTP library [29] and 832 monomorphic higher-order Sledgehammer (SH) generated problems [28] as our benchmarks[3]. Many TPTP problems require synthesis of complicated unifiers, whereas Sledgehammer problems are only mildly higher-order – many of them are solved with first-order unifiers.

We used the naive implementation of the JP procedure (**old**) as a baseline to evaluate the performance of our procedure. We compare it with the complete variant of our procedure (**cv**) and pragmatic variants (**pv**) with several different configurations of limits for applied bindings. All other Zipperposition parameters have been fixed. The cv configuration and all of the pv configurations use only pattern unification as an underlying oracle. To test the effect of oracle choice, we evaluated the complete variant in 8 combinations: with no oracles (**n**), with only fixpoint (**f**), pattern (**p**), or solid (**s**) oracle, and with their combinations: **fp**, **fs**, **ps**, **fps**.

Figure 1 compares different variants of the procedure with the naive JP implementation. Each pv configuration is denoted by pv^{a}_{bcde} where a is the limit on the total number of applied bindings, and b, c, d, and e are the limits of functional projections, eliminations, imitations, and identifications, respectively. Figure 2 summarizes the effects of using different oracles.

The configuration of our procedure with no oracles outperforms the JP procedure with the first-order oracle. This suggests that the design of the procedure, in particular lazy normalization and lazy application of the substitution, already reduces the effects of the JP procedure's main bottlenecks. Raw evaluation data shows that on TPTP benchmarks, complete and pragmatic configurations differ in the set of problems they solve – cv solves 19 problems not solved by pv^{4}_{2222}, whereas pv^{4}_{2222} solves 34 problems cv does not solve. Similarly, comparing the pragmatic configurations with each other, pv^{6}_{3333} and pv^{4}_{2222} each solve 13 problems that the other one does not. The overall higher success rate of pv^{2}_{1020} compared to pv^{2}_{1222} suggests that solving flex-flex pairs by trivial unifiers often suffices for superposition-based theorem proving.

Counterintuitively, in some cases the success rate does not increase if oracles are combined. Although oracles yield smaller CSUs, which in turn yield less clauses, these clauses typically contain many applied free variables, which can harm the performance of Zipperposition.

A subset of TPTP benchmarks is designed to test the efficiency of higher-order unification.

[3] An archive with raw results, scripts for running each configuration, and all used problems is available at `http://matryoshka.gforge.inria.fr/pubs/hounif_data.zip`

	CVC4	Leo-III	Satallax	Vampire	cv
`NUM020^1`	–	0.46	–	–	**0.03**
`NUM021^1`	–	–	–	–	**4.10**
`NUM415^1`	45.80	0.34	0.21	0.42	**0.03**
`NUM416^1`	47.37	0.92	0.21	0.41	**0.07**
`NUM417^1`	–	49.73	**0.30**	0.40	0.45
`NUM418^1`	–	0.40	1.29	0.38	**0.03**
`NUM419^1`	–	0.42	23.33	0.37	**0.03**
`NUM798^1`	46.29	0.35	4.01	0.38	**0.03**
`NUM799^1`	–	5.05	–	–	**0.03**
`NUM800^1`	–	–	–	**0.37**	3.15
`NUM801^1`	–	0.73	38.77	–	**0.50**

Figure 3 Time needed to prove a problem, in seconds.

It consists of problems concerning operations on Church numerals [3]. Our procedure performs exceptionally well on these problems – it solves all of them, usually faster than other competitive higher-order provers. There are 11 benchmarks in `NUM` category of TPTP that contain conjectures about Church numerals: `NUM020^1`, `NUM021^1`, `NUM415^1`, `NUM416^1`, `NUM417^1`, `NUM418^1`, `NUM419^1`, `NUM798^1`, `NUM799^1`, `NUM800^1`, and `NUM801^1`. We evaluated those problems using the same CPU nodes and the same time limits as above. In addition to Zipperposition, we used all higher-order provers that took part in the 2019 CASC THF category for this evaluation: CVC4 1.7 [1], Leo-III 1.4 [26], Satallax 3.4 [5], Vampire 4.4 THF [14]. Figure 3 shows the CPU time needed to solve a problem or "–" if the prover timed out.

9 Discussion and Related Work

The problem addressed in this paper is that of finding a complete and efficient higher-order unification procedure. Three main lines of research dominated the research field of higher-order unification over the last forty years.

The first line of research went in the direction of finding procedures that enumerate CSUs. The most prominent procedure designed for this purpose is the JP procedure [13]. Snyder and Gallier [25] also provide such a procedure, but instead of solving flex-flex pairs systematically, their procedure blindly guesses the head of the necessary binding by considering all constants in the signature and fresh variables of all possible types. Another approach, based on higher-order combinators, is given by Dougherty [10]. This approach blindly creates (partially applied) S-, K-, and I-combinator bindings for applied variables, which results in returning many redundant unifiers, as well as in nonterminating behavior even for simple examples such as $X\ \mathsf{a} = \mathsf{a}$.

The second line of research is concerned with enumerating preunifiers. The most prominent procedure in this line of research is Huet's [12]. The Snyder–Gallier procedure restricted not to solve flex-flex pairs is a version of PT procedure presented in Section 5. It improves Huet's procedure by featuring the Solution rule.

The third line of research gives up the expressiveness of the full λ-calculus and focuses on decidable fragments. Patterns [19] are arguably the most important such fragment in practice, with implementations in Isabelle [20], Leo-III [26], Satallax [5], λProlog [18], and other systems. Functions-as-constructors [16] unification subsumes pattern unification but is significantly more complex to implement. Prehofer [21] lists many other decidable fragments, not only for unification but also preunification and unifier existence problems. Most of these algorithms are given for second-order terms with various constraints on their variables. Finally, one of the first decidability results is the decidability of higher-order unification of terms with unary function symbols [11].

Our procedure draws inspiration from and contributes to all three lines of research. Accordingly, its advantages over previously known procedures can be laid out along those three lines. First, our procedure mitigates many issues of the JP procedure. Second, it can be modified not to solve flex-flex pairs, and become a version of Huet's procedure with important built-in optimizations. Third, our procedure can integrate any oracle for problems with finite CSUs – including the one we discovered.

The implementation of our procedure in Zipperposition was one of the reasons this prover evolved from proof-of-concept prover for higher-order logic to competitive higher-order prover. In the 2019 edition of CASC[4], Zipperposition featured a less efficient version of our procedure. Nevertheless, it finished neck-and-neck with state-of-the-art prover Leo-III and has beaten all the newcomers in higher-order competition category.

10 Conclusion

We presented a procedure for enumerating a complete set of higher-order unifiers that is designed for efficiency. Due to design that restricts search space and tight integration of oracles it reduces the number of redundant unifiers returned and gives up early in cases of nonunifiability. In addition, we presented a new fragment of higher-order terms that admits finite CSUs. Our implementation shows a clear improvement over previously known procedure.

In future work, we will focus on designing intelligent heuristics that automatically adjust unification parameters according to the type of the problem. For example, we should usually choose shallow unification for mostly first-order problems and deeper unification for hard higher-order problems. We plan to investigate other heuristic choices, such as the order of bindings and the way in which search space is traversed (breadth- or depth-first). We are also interested in further improving the termination behavior of the procedure, without sacrificing completeness. Finally, following the work of Libal [15] and Zaionc [31], we would like to consider the use of regular grammars to finitely present infinite CSUs. For example, the grammar $X ::= \lambda x.\, x \mid \lambda x.\, \mathsf{f}\,(X\,x)$ represents all elements of the CSU for the problem $\lambda x.\, X\,(\mathsf{f}\,x) \overset{?}{=} \lambda x.\, \mathsf{f}\,(X\,x)$.

Acknowledgment

We are grateful to the maintainers of StarExec for letting us use their service. We thank Ahmed Bhayat, Jasmin Blanchette, Daniel El Ouraoui, Mathias Fleury, Pascal Fontaine, Predrag Janičić, Robert Lewis, Femke van Raamsdonk, Hans-Jörg Schurr, Sophie Tourret, and Dmitriy Traytel for suggesting many improvements to this text. Vukmirović and Bentkamp's research has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 713999, Matryoshka). Nummelin has received funding from the Netherlands Organization for Scientific Research (NWO) under the Vidi program (project No. 016.Vidi.189.037, Lean Forward).

[4] `http://tptp.cs.miami.edu/CASC/27/WWWFiles/DivisionSummary1.html`

References

1 Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV 2011*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

2 Alexander Bentkamp, Jasmin Blanchette, Sophie Tourret, Petar Vukmirović, and Uwe Waldmann. Superposition with lambdas. In Pascal Fontaine, editor, *CADE-27*, volume 11716 of *LNCS*, pages 55–73. Springer, 2019.

3 Christoph Benzmüller and Chad E. Brown. A structured set of higher-order problems. In Joe Hurd and Thomas F. Melham, editors, *TPHOLs 2005*, volume 3603 of *LNCS*, pages 66–81. Springer, 2005.

4 Ahmed Bhayat and Giles Reger. Restricted combinatory unification. In Pascal Fontaine, editor, *CADE-27*, volume 11716 of *LNCS*, pages 74–93. Springer, 2019.

5 Chad E. Brown. Satallax: An automatic higher-order prover. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR 2012*, volume 7364 of *LNCS*, pages 111–117. Springer, 2012.

6 Nicolaas G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *J. Symb. Log.*, 40(3):470–470, 1975.

7 Simon Cruanes. *Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond.* PhD thesis, École polytechnique, 2015.

8 Simon Cruanes. Superposition with structural induction. In Clare Dixon and Marcelo Finger, editors, *FroCoS 2017*, volume 10483 of *LNCS*, pages 172–188. Springer, 2017.

9 Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Commun. ACM*, 22(8):465–476, 1979.

10 Daniel J. Dougherty. Higher-order unification via combinators. *Theor. Comput. Sci.*, 114(2):273–298, 1993.

11 William M. Farmer. A unification algorithm for second-order monadic terms. *Ann. Pure Appl. Logic*, 39(2):131–174, 1988.

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

13 Don C. Jensen and Tomasz Pietrzykowski. Mechanizing omega-order type theory through unification. *Theor. Comput. Sci.*, 3(2):123–171, 1976.

14 Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In Natasha Sharygina and Helmut Veith, editors, *CAV 2013*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.

15 Tomer Libal. Regular patterns in second-order unification. In Amy P. Felty and Aart Middeldorp, editors, *CADE-25*, volume 9195 of *LNCS*, pages 557–571. Springer, 2015.

16 Tomer Libal and Dale Miller. Functions-as-constructors higher-order unification. In Delia Kesner and Brigitte Pientka, editors, *FSCD 2016*, volume 52 of *LIPIcs*, pages 26:1–26:17. Schloss Dagstuhl, 2016.

17 Tomer Libal and Alexander Steen. Towards a substitution tree based index for higher-order resolution theorem provers. In Pascal Fontaine, Stephan Schulz, and Josef Urban, editors, *IJCAR 2016*, volume 1635 of *CEUR-WS*, pages 82–94. CEUR-WS, 2016.

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

19 Tobias Nipkow. Functional unification of higher-order patterns. In E. Best, editor, *LICS '93*, pages 64–74. IEEE Computer Society, 1993.

20 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

21 Christian Prehofer. *Solving higher order equations: from logic to programming.* PhD thesis, Technical University Munich, Germany, 1995.

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

23 Stephan Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2-3):111–126, 2002.

24 Stephan Schulz. Fingerprint indexing for paramodulation and rewriting. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *IJCAR 2012*, volume 7364 of *LNCS*, pages 477–483. Springer, 2012.

25 Wayne Snyder and Jean H. Gallier. Higher-order unification revisited: Complete sets of transformations. *J. Symb. Comput.*, 8(1/2):101–140, 1989.

26 Alexander Steen and Christoph Benzmüller. The higher-order prover Leo-III. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *IJCAR 2018*, volume 10900 of *LNCS*, pages 108–116. Springer, 2018.

27 Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. Starexec: A cross-community infrastructure for logic solving. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *IJCAR 2014*, volume 8562 of *LNCS*, pages 367–373. Springer, 2014.

28 Nik Sultana, Jasmin Christian Blanchette, and Lawrence C. Paulson. LEO-II and Satallax on the Sledgehammer test bench. *J. Applied Logic*, 11(1):91–102, 2013.

29 Geoff Sutcliffe. The TPTP problem library and associated infrastructure - from CNF to TH0, TPTP v6.4.0. *J. Autom. Reasoning*, 59(4):483–502, 2017.

30 Petar Vukmirovic, Jasmin Christian Blanchette, Simon Cruanes, and Stephan Schulz. Extending a brainiac prover to lambda-free higher-order logic. In Tomás Vojnar and Lijun Zhang, editors, *TACAS 2019*, volume 11427 of *LNCS*, pages 192–210. Springer, 2019.

31 Marek Zaionc. The set of unifiers in typed lambda-calculus as regular expression. In Jean-Pierre Jouannaud, editor, *RTA-85*, volume 202 of *LNCS*, pages 430–440. Springer, 1985.