Efficient Full Higher-Order Unification

Petar Vukmirović
Vrije Universiteit Amsterdam, The Netherlands
p.vukmirovic@vu.nl

Alexander Bentkamp
Vrije Universiteit Amsterdam, The Netherlands
a.bentkamp@vu.nl

Visa Nummelin
Vrije Universiteit Amsterdam, The Netherlands
visa.nummelin@vu.nl

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

Digital Object Identifier 10.4230/LIPIcs.FSCD.2020.5

Related Version A technical report contains further details and proofs. It is available at `http://matryoshka.gforge.inria.fr/pubs/hounif_report.pdf`.

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

Funding 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).

Acknowledgements 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, Dmitriy Traytel, and the anonymous reviewers for suggesting many improvements to this text.

1 Introduction

Unification is concerned with finding a substitution that makes two terms equal, for some notion of syntactic equality. Since the invention of Robinson's first-order unification algorithm [19], it has become an indispensable tool in theorem proving, logic programming, natural language processing, programming language compilation and other areas of computer science.

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 [10]. 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 [17], Leo-III [22], and Satallax [3]. However, recent developments in higher-order theorem proving [1,2] require full unification – i.e., enumeration of unifiers even for flex-flex pairs, which is the focus of this paper.

Jensen and Pietrzykowski's (JP) procedure [11] 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 for 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 [16], functions-as-constructors [13], and a new fragment we present in Section 4. The unification procedures of Isabelle and Leo-III 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 applies infinitely branching rule to flex-rigid pairs, whereas Huet's preunification procedure solves flex-rigid pairs using simpler, finitely branching rules. To gracefully generalize Huet's procedure, we show that his rules for flex-rigid pairs suffice to enumerate CSUs if combined with appropriate rules for flex-flex pairs.

Fourth, 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, optimizations such as lazy application of substitutions and lazy β-normalization cannot easily be integrated into the JP procedure. The rules of simpler procedures (e.g., first-order [9] and pattern unification [16]) depend only on the heads of the unification pair. Thus, to determine the next step, implementations of these procedures need to substitute and β-reduce only until the heads of the current unification pair are not mapped by the substitution and are not λ-abstractions. Since the JP procedure is not based on the decomposition of unification pairs, it is unfit for optimizations of this kind. We designed our procedure to allow for 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 [20] (Section 5). We implemented our procedure, several oracles, and the fingerprint index in the Zipperposition prover (Section 6). 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 7). The results show substantial performance improvements.

This paper lays out the main ideas behind our unification procedure. A separate technical report contains details and proofs of all statements [27].

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 [16], 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.

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 free 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, $\varrho F \leftrightarrow^*_{\alpha\beta\eta} \sigma F$.

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 gracefully generalizes the corresponding first-order notion: a is a subterm of $\mathsf{f}\, \mathsf{a}\, \mathsf{b}$, 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 η-long β-reduced 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 [11], 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 failure nodes $\bot$ or substitutions 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; this mapping abstracts the concept of unification rules present in other unification procedures. 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 $(\varnothing, \sigma) \longrightarrow \sigma$

Normalize$_{\alpha\eta}$ $(\{\lambda\overline{x}_m.\, s \overset{?}{=} \lambda\overline{y}_n.\, t\} \uplus E, \sigma) \longrightarrow (\{\lambda\overline{x}_m.\, s \overset{?}{=} \lambda\overline{x}_m.\, t'\, x_{n+1} \ldots x_m\} \uplus E, \sigma)$
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$_\beta$ $(\{\lambda\overline{x}.\, s \overset{?}{=} \lambda\overline{x}.\, t\} \uplus E, \sigma) \longrightarrow (\{\lambda\overline{x}.\, s_{\downarrow\mathsf{h}} \overset{?}{=} \lambda\overline{x}.\, t_{\downarrow\mathsf{h}}\} \uplus E, \sigma)$
where s or t is not in hnf

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

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

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

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

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

Bind $(\{s \overset{?}{=} t\} \uplus E, \sigma) \longrightarrow (\{s \overset{?}{=} t\} \uplus E, \varrho\sigma)$
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, OracleSucc and Bind must maintain the invariant that all substitutions are idempotent.

The OracleSucc and 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 check whether a unification constraint is inside the fragment, we need to fully apply the substitution and β-normalize the constraint. To avoid these expensive operations and enable efficient oracle integration, oracles must be redesigned to lazily discover whether the terms belong to their fragment. Most oracles contain a decomposition operation which requires only a partial application of the substitution and only partial β-normalization. If one of the constraints resulting from decomposition is not in the fragment, the original problem is not in the fragment. This allows us to detect that the problem is not in the fragment without fully applying the substitution and β-normalizing.

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 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

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

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

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

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. Furthermore, 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.*

Given a unification constraint $\lambda \overline{x}.\, s \overset{?}{=} \lambda \overline{x}.\, t$, $\mathcal{P}$ is defined as follows:

- 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 distinguish 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.

The mapping $\mathcal{P}$ uses many binding rules of the JP procedure, but our procedure 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, our procedure forces the eliminations and iterations to be applied as soon as it observes a flex-flex pair with identical heads. After applying the Decompose transition, this flex-flex pair will be reduced to pairs representing the arguments of the identical heads. Therefore, unlike the JP procedure, it will not apply bindings to the flex-flex pair after bindings have been applied to its arguments.

The JP procedure can be modified to solve the preunification problem by making it choose only flex-rigid disagreement pairs and terminate with a preunifier when no flex-rigid pair remains. However, such a procedure would be less efficient than Huet's procedure because it would use the iteration binding instead of Huet-style projection to solve flex-rigid pairs. Our procedure applies Huet-style projections on flex-rigid pairs, which results in two important improvements over the JP procedure. First, our procedure terminates more often than the JP procedure because Huet-style projections cause only a finite branching, whereas iteration causes an infinite branching. Second, when our procedure is modified to solve the preunification problem by never selecting flex-flex pairs and stopping when only flex-flex pairs are left, it becomes an optimized variant of Huet's procedure that supports oracles as well as lazy substitution and β-reduction.

The bindings of our procedure contain further optimizations that are absent 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. Moreover, it does not keep track of which rules introduced which variables: 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 6, 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:

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

▶ **Example 3.** The search space restrictions also allow us to prune some redundant unifiers. Consider the problem $F\,(G\,\mathsf{a}) \overset{?}{=} F\,\mathsf{b}$, 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:

$$(F\,(G\,\mathsf{a}) \overset{?}{=} F\,\mathsf{b}, \mathrm{id}) \overset{\textsf{JP Proj}}{\Longrightarrow} (F\,\mathsf{a} = F\,\mathsf{b}, \{G \mapsto \lambda x.\, x\})$$

$$\overset{\textsf{Elim}}{\Longrightarrow} (F' = F', \{G \mapsto \lambda x.\, x, F \mapsto \lambda x.\, F'\}) \overset{\textsf{Succeed}}{\Longrightarrow} \{G \mapsto \lambda x.\, x, F \mapsto \lambda x.\, F'\}$$

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

Proof of Completeness. Like the JP procedure, our procedure misses no unifiers:

▶ **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 of Theorem 4 is an adaptation of the proof given by JP [11]. Definitions and lemmas are reused, but they are combined together differently. The full proof is given in our technical report [27]. 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 guides the choice of 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.

Contrary to our procedure, the proof assumes that all terms are in η-long β-reduced form and that all substitutions are fully applied. These assumptions are justified because all bindings depend only on the head of terms and hence replacing the lazy transitions $\textsf{Normalize}_{\alpha\eta}$, $\textsf{Normalize}_{\beta}$, and $\textsf{Dereference}$ by eager counterparts only affects the efficiency but not the overall behavior of our procedure.

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 the 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 bound variable 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 $\varrho_j \subseteq \varrho_{j+1}\,\delta$ similarly 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 $\varrho_j E_j$.

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 preliminary experiments showed that $\mathcal{P}$ defined as follows is a reasonable compromise between completeness and performance:

- 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 distinguish 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.

The pragmatic variant of our procedure removes all iteration bindings to enforce finite branching. Moreover, it 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 originating from proof assistants, shallow unification depth usually suffices. However, hard hand-crafted problems often need deeper unification.

4 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 PT procedure [18], 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 $E_n = \bot$. Unlike previously, in this section we consider all terms to be $\alpha\beta\eta$-equivalence classes with the η-long β-reduced form as their canonical representative and 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 $\{\boxed{s \overset{?}{=} s}\} \uplus E \Longrightarrow^{\mathrm{id}} E$

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

Failure $\{\boxed{\lambda\overline{x}.\, a\,\overline{s} \overset{?}{=} \lambda\overline{x}.\, b\,\overline{t}}\} \uplus E \Longrightarrow^{\mathrm{id}} \bot$
where a and b are different rigid heads

Solution $\{\boxed{\lambda\overline{x}.\, F\,\overline{x} \overset{?}{=} \lambda\overline{x}.\, t}\} \uplus E \Longrightarrow^{\varrho} \varrho(E)$
where F does not occur in t, t does not have a flex head, and $\varrho = \{F \mapsto \lambda\overline{x}.\, t\}$

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

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* 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 8 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. Lemma 5 lifts Prehofer's preunification result to solid terms:

▶ **Lemma 5.** *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.*

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 6.** *The unification problem $\{\lambda\overline{x}.\, F\,\overline{s}_m \overset{?}{=} \lambda\overline{x}.\, F\,\overline{t}_m\}$, 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 $s_{j_i} = t_{j_i}$.*

▶ **Lemma 7.** *Let $\{\lambda\overline{x}.\, F\,\overline{s}_m \overset{?}{=} \lambda\overline{x}.\, G\,\overline{t}_n\}$ be a solid unification problem where $F \neq G$. Then there is a finite CSU $\{\sigma_i^1, \ldots, \sigma_i^{k_i}\}$ of the problem $\{s_i \overset{?}{=} H_i\,\overline{t}_n\}$, where H_i is a fresh free variable. Let $\lambda\overline{y}_n.\, s_i^j = \lambda\overline{y}_n.\, \sigma_i^j(H_i)\,\overline{y}_n$. Similarly, there is a finite CSU $\{\tilde{\sigma}_i^1, \ldots, \tilde{\sigma}_i^{l_i}\}$ of the problem $\{t_i \overset{?}{=} \tilde{H}_i\,\overline{s}_m\}$, where $\tilde{H}_i$ is a fresh free variable. Let $\lambda\overline{x}_m.\, t_i^j = \lambda\overline{x}_m.\, \tilde{\sigma}_i^j(\tilde{H}_i)\,\overline{x}_m$. Let Z be a fresh free variable. An MGU σ for the given problem is*

$$F \mapsto \lambda\overline{x}_m.\, Z\, \underbrace{x_1 \ldots x_1}_{k_1 \textit{ times}} \ldots \underbrace{x_m \ldots x_m}_{k_m \textit{ times}}\, t_1^1 \ldots t_1^{l_1} \ldots t_n^1 \ldots t_n^{l_n}$$

$$G \mapsto \lambda\overline{y}_n.\, Z\, s_1^1 \ldots s_1^{k_1} \ldots s_m^1 \ldots s_m^{k_m}\, \underbrace{y_1 \ldots y_1}_{l_1 \textit{ times}} \ldots \underbrace{y_n \ldots y_n}_{l_n \textit{ times}}$$

Our proof that finite CSUs exist relies on Prehofer's proof that PT terminates without producing flex-flex pairs for the matching problem $\{\lambda \overline{x}_n.\, F\, \overline{s}_k \overset{?}{=} \lambda \overline{x}_n.\, t\}$ where $F\, \overline{s}_k$ is strictly solid and t is ground and second-order. His proof is easily generalized to the case where t is arbitrary order and $F\, \overline{s}_k$ is solid. Since PT is complete, we conclude that such problems have finite CSUs.

▶ **Theorem 8.** *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.*

This CSU is straightforward to compute. By Lemma 5, 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 E can be found as follows. Choose a constraint $(u \overset{?}{=} v) \in E$ and determine an MGU ϱ for it using Lemma 6 or 7. 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, let $\delta_E = \varrho'\varrho$. Then $\{\delta_E \sigma \mid \text{PT produces preunifier } \sigma \text{ with constraints } E\}$ is a finite CSU.

▶ **Example 9.** 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 7, 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 8 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.

5 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*.

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 [2,14]. A *fingerprint index* [20,28] 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 the 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 procedure 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 figure how to cope with λ-abstractions and bound variables. To that end, we define a function $\lfloor t \rfloor$, defined on β-reduced terms in De Bruijn [4] notation:

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

We let x_i be a bound variable of type α with De Bruijn index i, and db_i^α be a fresh constant corresponding to this variable. All constants db_i^α must be fresh. Effectively, $\lfloor\ \rfloor$ transforms a η-long β-reduced higher-order term to an untyped first-order term. Let $t_{\downarrow\beta\eta}$ be the η-long β-reduced 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

$$\mathrm{fp}_{\mathsf{ho}}(s) = \mathrm{fp}(\lfloor \langle s_{\downarrow\beta\eta} \rangle_{\mathsf{db}} \rfloor) = \mathrm{fp}(\mathsf{g}(\mathsf{db}_0^\alpha)) = (\mathsf{db}_0^\alpha, \mathsf{N}, \mathsf{N})$$
$$\mathrm{fp}_{\mathsf{ho}}(t) = \mathrm{fp}(\lfloor \langle t_{\downarrow\beta\eta} \rangle_{\mathsf{db}} \rfloor) = \mathrm{fp}(\mathsf{f}(\mathsf{db}_1^\alpha, \mathsf{db}_0^\alpha)) = (\mathsf{db}_1^\alpha, \mathsf{N}, \mathsf{db}_0^\alpha)$$

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 terms using the method described here as well.

6 Implementation

Zipperposition [5, 6] is an open-source[1] 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 [1], 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. In 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 sets containing a unifier or empty sets). 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 unification problems $E_1, E_2, \ldots$ to be solved. Solutions to each of those problems are lazy lists $p_1, p_2, \ldots$ containing subsingletons of unifiers. To avoid postponing some unifier indefinitely, we use the dovetailing technique: we first take one subsingleton from p_1, then one from each of p_1 and p_2. We continue with one subsingleton from each of 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 [10] is concerned with problems of the form $\{F \stackrel{?}{=} t\}$. 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 $F \stackrel{?}{=} t$ has no solutions. Otherwise, the fixpoint oracle is not applicable.

7 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 [1], Bentkamp implemented a straightforward version of JP procedure. This version is faithful to the original description, with a check as

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

	jp	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.

to whether a (sub)problem can be solved using a first-order oracle as the only optimization. Our evaluations were performed on StarExec Miami [23] 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 [25] and 832 monomorphic higher-order Sledgehammer (SH) generated problems [24] as our benchmarks[2]. 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 (**jp**) 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 to the values of a variant of a well-performing configuration we used for the CASC-27 theorem proving competition [26]. 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.

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

Counterintuitively, in some cases, using oracles can hurt the performance of Zipperposition. Using oracles typically results in generating smaller CSUs, whose elements are more general substitutions than the ones we obtain without oracles. These more general substitutions usually contain more applied variables, which Zipperposition's heuristics avoid due to their explosive nature. This can make Zipperposition postpone necessary inferences for too long. Configuration n benefits from this effect and therefore solves 18 TPTP problems that no other configuration in Figure 2 solves. The same effect also gives configurations with only one oracle an advantage over configurations with multiple oracles on some problems.

The evaluation sheds some light on how often solid unification problems appear in practice. The raw data show that configuration s solves 5 TPTP problems that neither f nor p solve. Configuration f solves 8 TPTP problems that neither s nor p solve, while p solves 9 TPTP problems that two other configurations do not. This suggests that the solid oracle is slightly less beneficial than the fixpoint or pattern oracles, but still presents a useful addition the set of available oracles.

A subset of 11 TPTP benchmarks, concerning operations on Church numerals, is designed to test the efficiency of higher-order unification. Our procedure performs exceptionally well on these problems – it solves all of them, usually faster than other competitive higher-order provers. In particular, on two of these problems, neither Leo-III 1.4 nor Satallax 3.4 produce a proof within a 60 seconds CPU limit, while the cv configuration proves each of them in less than 4.5 s. A full list of these problems is in our technical report [27].

8 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 [11]. Snyder and Gallier [21] 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 [7]. 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 problems 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 [10]. The Snyder–Gallier procedure restricted not to solve flex-flex pairs is a version of the PT procedure presented in Section 4. It improves Huet's procedure by featuring a Solution rule.

The third line of research gives up the expressiveness of the full λ-calculus and focuses on decidable fragments. Patterns [16] are arguably the most important such fragment in practice, with implementations in Isabelle [17], Leo-III [22], Satallax [3], λProlog [15], and other systems. Functions-as-constructors [13] unification subsumes pattern unification but is significantly more complex to implement. Prehofer [18] 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 Farmer's discovery [8] that higher-order unification of terms with unary function symbols is decidable.

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, it can integrate any oracle for problems with finite CSUs – including the one we discovered.

9 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 evaluation 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 [12] and Zaionc [29], we would like to consider the use of regular grammars to finitely present infinite CSUs. For example, the grammar $G ::= \lambda x.\, x \mid \lambda x.\, \mathsf{f}\,(G\,x)$ represents all elements of the CSU for the problem $\lambda x.\, G\,(\mathsf{f}\,x) \overset{?}{=} \lambda x.\, \mathsf{f}\,(G\,x)$.

References

1 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.

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

3 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.

4 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.

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

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

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

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

9 Krystof Hoder and Andrei Voronkov. Comparing unification algorithms in first-order theorem proving. In Bärbel Mertsching, Marcus Hund, and Muhammad Zaheer Aziz, editors, *KI 2009: Advances in Artificial Intelligence, 32nd Annual German Conference on AI, Paderborn, Germany, September 15-18, 2009. Proceedings*, volume 5803 of *Lecture Notes in Computer Science*, pages 435–443. Springer, 2009.

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

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

12 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.

13 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.

14 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.

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

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

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

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

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

20 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.

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

22 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.

23 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.

24 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.

25 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.

26 Geoff Sutcliffe. The CADE-27 automated theorem proving system competition - CASC-27. *AI Commun.*, 32(5-6):373–389, 2019.

27 Petar Vukmirović, Alexander Bentkamp, and Visa Nummelin. Efficient full higher-order unification (technical report), 2020. URL: `http://matryoshka.gforge.inria.fr/pubs/hounif_report.pdf`.

28 Petar Vukmirović, 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.

29 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.