Restricted Combinatory Unification

Ahmed Bhayat and Giles Reger

University of Manchester, Manchester, UK

Abstract. First-order theorem provers are commonly utilised as backends to proof assistants. In order to improve efficiency, it is desirable that such provers can carry out some higher-order reasoning. In his 1991 paper, Dougherty proposed a combinatory unification algorithm for higher-order logic. The algorithm removes the need to deal with λ-binders and α-renaming, making it attractive to implement in first-order provers. However, since publication it has garnered little interest due to its poor characteristics. It fails to terminate on many trivial instances and requires polymorphism. We present a restricted version of Dougherty's algorithm that is incomplete, terminating and does not require polymorphism. Further, we describe its implementation in the Vampire theorem prover, including a novel use of a substitution tree as a filtering index for higher-order unification. Finally, we analyse the performance of the algorithm on two benchmark sets and show that it is a significant step forward.

1 Introduction

Higher-order logic has many applications from the formalisation of mathematics through to uses in verifying the safety and security of computer systems. This has led to a growing interest in the *automation* of reasoning in higher-order logic (which is undecidable in general). A successful step in this direction has been via translation to first-order logic and utilisation of first-order theorem provers, made possible by the high level of maturity and sophistication of such provers. Proof assistants such as Isabelle [23] and Coq [9] along with automated provers such as Leo-III [29], interact with first-order provers by translating their native logic into first-order logic [21]. These translations tend to be incomplete and suffer from a number of problems of which two of the most important are highlighted below. This paper addresses these problems with a higher-order unification algorithm for combinatory logic and its pragmatic realisation within the first-order Vampire theorem prover [17].

The translation of nameless or λ-functions is often carried out using combinators. However, when translating to monomorphic first-order logic, supported by most first-order provers, an infinite set of combinators is required to guarantee completeness. Thus, most translation schemes suffice with including the combinators necessary to translate the λ-functions present in the input. Consider the somewhat contrived conjecture $\exists X : X\, b\, a = a$. On negation, this becomes $X\, b\, a \neq a$. As there are no 'λ's there would be no combinators present in the translation. Accordingly, the prover would be unable to synthesise the combinatory equivalent (which is $\boldsymbol{CK}$) of the λ-term $\lambda xy.y$ and would be unable to find a proof. Now consider the same conjecture, but assume that some combinator axioms are present in the first-order translation (this could be

via the heuristic addition of combinator axioms, an option in Vampire). In this case, the axioms can superpose amongst themselves. For example, the $\boldsymbol{K}$ combinator axiom $\boldsymbol{K}\,X\,Y = X$ could superpose onto the right hand side of the $\boldsymbol{S}$ combinator axiom $\boldsymbol{S}\,X\,Y\,Z = X\,Z(Y\,Z)$ to produce the equation $\boldsymbol{S}\,\boldsymbol{K}\,Y\,Z = Z$. A consequence of the combinator axioms has been derived that is of no use in proving the goal.

Both of these problems stem from attempting to achieve what the goal-oriented procedure of higher-order unification (HOU) does using the non-goal-oriented superposition calculus. Thus, there is a strong argument that introducing some form of higher-order unification into first-order provers would significantly improve their performance on problems generated by proof-assistants. This is particularly so if this can be achieved without harming performance on the first-order portion of the problems.

Indeed, replacing syntactic first-order unification with unification modulo the combinator axioms and adding an extra inference rule called 'extended narrowing' suffices to turn ordered resolution into a complete proof calculus for intensional HOL [13] [7]. Unfortunately, the required proof method does not easily extend to superposition. Unification modulo the combinator axioms would result in a complete superposition calculus for higher-order logic if a reduction ordering compatible with the combinator axioms could be found [31]. However, such an ordering is impossible as any ordering that partitions $\boldsymbol{K}\,X\,Y$ and X into an equivalence class must violate the subterm property.

Contribution The usage of λ-binders adds complications and subtleties to the implementation of higher-order unification. In particular, during the application of substitutions, bound variables require α-renaming. It is precisely to deal with such issues that explicit substitution calculi [12] [10] have been investigated.

First-order provers are generally not able to handle binders, so rather than explicit substitution calculi, we focus on higher-order unification in the setting of combinatory logic. The only existing algorithm in this setting is Dougherty's algorithm. The algorithm is a complete unification procedure for *polymorphic* higher-order terms. It is unattractive for implementation because it produces many redundant unifiers and does not terminate in many cases. Our main contributions in this paper are:

- A modification of Dougherty's algorithm that works on *monomorphic* higher order terms (Section 4). Our algorithm is incomplete, but terminating and has shown strong experimental results.
- A method of imperfect filtering that facilitates the implementation of higher-order unification without harming performance on first-order problems (Section 5).

These techniques are implemented in the Vampire theorem prover [17] (along with other extensions reported elsewhere for higher-order reasoning) and experimental results (Section 6) show that combinatory unification can help solve new problems.

2 Preliminaries

In this paper some knowledge of first-order unification and substitution tree indexing is assumed. The reader is referred to [15] and [14] for further details. We present the logical terminology used throughout the rest of the paper. We work with the combinatory-logic (CL) first developed by Schönfinkel, but popularised by Curry. As Dougherty's

original algorithm works with polymorphic terms and our modification works with monomorphic terms, both are presented here.

Terms are built over a set of types. Let S be a set of sort symbols that act as syntactic identifiers for the base types of the logic and V_{ty} be a set of sort variables. The set of types is defined as:

Monomorphic Types	$\tau ::= \sigma \mid \tau \to \tau$	where $\sigma \in S$
Polymorphic Types	$\tau ::= \sigma \mid \alpha \mid \tau \to \tau$	where $\sigma \in S, \alpha \in V_{ty}$

A *polymorphic* type declaration is of the form $\Pi\overline{\alpha_m}.\tau$ where each α_i is a type variable and τ is a potentially polymorphic type containing type variables from $\overline{\alpha_m}$ ($\overline{\alpha_m}$ is a list of type variables). A *monomorphic* type declaration is simply τ for some monomorphic type τ.

For each type τ let V_τ be a set of term variables of type τ and let $V = \bigcup_{\tau \in T} V_\tau$. Further, let Σ be a set of typed constant symbols. When working in monomorphic CL, for every type τ, there exists a constant $\boldsymbol{I} : \tau \to \tau \in \Sigma$. For every pair of types τ, ρ, there exists a constant $\boldsymbol{K} : \tau \to \rho \to \tau \in \Sigma$ and for every triple of types τ, ρ, σ, there exists a constant $\boldsymbol{S} : (\tau \to \rho \to \sigma) \to (\tau \to \rho) \to \tau \to \sigma \in \Sigma$. The constants $\boldsymbol{I}, \boldsymbol{K}$ and $\boldsymbol{S}$ are known as *basic combinators*. When working in polymorphic CL, the existence of only three polymorphic basic combinators is required. From now on, unless required for clarity, type subscripts are omitted.

We define monomorphic and polymorphic terms together as follows. Let $f : \Pi\overline{\alpha_m}.\tau$ be a member of Σ. In the monomorphic case, $m = 0$. Then, f applied to m type arguments: $f\langle\overline{\sigma_m}\rangle$ is a term of type $\tau\{\overline{\alpha_m} \to \overline{\sigma_m}\}$ for some tuple of types $\overline{\sigma_m}$. For all $X \in V_\tau$, X is a term of type τ. If t is a term of type $\tau \to \sigma$ and t' is a term of type τ, then tt' is a term of type σ. Where type arguments are irrelevant, they are dropped from the presentation.

Terms of the form tt' are called *applications*. Non-applicative terms $X, \zeta \in V \cup \Sigma$ are called heads. A term can be decomposed uniquely into a head and n arguments e.g. $\zeta\, t_1 \ldots t_n$ or in shorter form $\zeta\ \overline{t_n}$. By $head(t)$ the unique head of t is intended e.g. $head(f\, a\, b) = f$. A head is first-order if it is not a variable or combinator. A term is *passive* if it does not have a combinator head. The positions $pos(t)$ of term t are defined in the standard fashion; we write $t|_p$ for the subterm of t at position p. Recall the partial ordering $<$ on positions such that $p < p'$ if $t|_{p'}$ is a subterm of $t|_p$. A *higher-order subterm*[1] is a subterm with a variable or combinator head. A term that contains no higher-order subterms is called first-order. The set of *first-order positions* over a term t is defined as all $p \in pos(t)$ such that, for all $p' < p$, $head(t|_{p'})$ is not a variable or combinator. A term is *linear* if it contains no repeated variables. In a term $\zeta\,\overline{t_n}$, subterms of the form $\zeta\,\overline{t_i}$ for $i < n$ are known as *prefix* subterms.

In what follows, capital letters such as $X, Y, Z \ldots$ are used to denote variables, s, t, u denote arbitrary terms, $a, b, c \ldots$ denote constants.

Unification Unification involves substituting terms for (free) variables in order to make two or more terms equal. This equality could be syntactic equality, as is generally the

[1] Note that the definition of higher-order subterm here is different to its usage in [3]

case in first-order theorem proving, or equality modulo a set of axioms. In classic HOU, the goal is to find substitution(s) $\theta_1 \dots \theta_n$ for terms $t_1, t_2 \dots t_n$ such that $t_1\theta_i =_{\beta\eta} t_2\theta_i =_{\beta\eta} \dots =_{\beta\eta} t_n\theta_i$ where $=_{\beta\eta}$ is equality modulo the axioms of β and η reduction. In this paper, we are interested in the relationship $=_c$ (defined below) on terms of the combinatory logic. A substitution that unifies two or more terms is known as a *unifier*. When working with polymorphic terms, a substitution θ is a *pair*, a term substitution θ_0 and a type substitution θ_1. By an abuse of notation, the same symbols are used to refer to these dual substitutions and standard monomorphic substitutions.

For two unifiers σ and θ, σ is more general than θ ($\sigma \leq \theta$) iff there exists a substitution γ such that $\sigma\gamma = \theta$. In this case, θ is *redundant*. If neither $\sigma \leq \theta$, nor $\theta \leq \sigma$ then σ and θ are *independent*. In syntactic first-order unification, if two terms have a unifier then they have a unique (up to variable naming) *most general unifier* (mgu). This is not the case with HOU and the notion of mgu is generalised to that of *complete set of unifiers* (csu). Let Γ be a set of unifiers of terms $t_1 \dots t_n$. Γ is a csu iff forall $\sigma \in unifers(\overline{t_n})$ such that $\sigma \notin \Gamma$, we have $\exists \sigma' \in \Gamma$ such that $\sigma' \leq \sigma$. Γ is a minimal csu iff forall $\sigma_1, \sigma_2 \in \Gamma$, σ_1 and σ_2 are independent. With respect to HOU, the minimal csu may be infinite.

The HOU problem is undecidable and any complete algorithm must produce redundant unifiers [16]. Let $\stackrel{up}{=}$ be the least congruence relation on combinatory terms which contains $\{(t_1, t_2) | head(t_1), head(t_2) \in V\}$. The pre-unification problem is to find substitutions $\theta_1 \dots \theta_n$ such that for terms $t_1 \dots t_n$, $t_i\theta_k \stackrel{up}{=} t_j\theta_k$ forall i, j and k. Huet devised a famous complete algorithm for pre-unification [16] that is irredundant.

Definition 1. *For CL terms t_1 and t_2, $t_1 =_c t_2$ or equivalently t_1 is C-equal to t_2, iff $\Lambda(t_1) =_{\beta\eta} \Lambda(t_2)$ where Λ is the following translation between combinatory terms and terms of the λ-calculus.*

$$\begin{array}{ll} \Lambda(a) = a \text{ for a not a combinator} & \Lambda(\boldsymbol{S}) = \lambda XYZ.XZ(YZ) \\ \Lambda(\boldsymbol{I}) = \lambda X.X & \Lambda(t_1t_2) = \Lambda(t_1)\Lambda(t_2) \\ \Lambda(\boldsymbol{K}) = \lambda XY.X & \end{array}$$

The translation Λ can be used to derive unifiers of λ-terms from unifiers of CL terms. The details can be found in Dougherty's paper [11]. If θ is a substitution such that θ unifies two or more terms with respect to $=_c$, θ is referred to as a C-unifier.

Following Dougherty, a *system* is defined as a multiset of pairs of CL terms. A pair is *trivial* if its components are identical and it is valid if its components are C-equal. The definition of flex-flex, flex-rigid and rigid-rigid pairs is as common in HOU literature [27]. The definitions of *trivial* and *valid* are extended to systems in the obvious way. The *extensional combinatory unification* problem is to find, for any given system $\mathcal{S}$, a set of unifiers U such that $\forall \langle t_1, t_2 \rangle \in \mathcal{S}$ and $\forall \theta \in U$ $t_1\theta =_c t_2\theta$. A system $\mathcal{S}$ is *simple* if for all pairs $\langle t, t' \rangle$ in $\mathcal{S}$, terms t and t' do not have identical rigid heads and both are passive.

Definition 2 (Solved System). *For some system $\mathcal{S}$, a pair $\langle X, t_2 \rangle \in \mathcal{S}$ is* solved *if X doesn't occur in t_2 or in any other pair in $\mathcal{S}$. A system $\mathcal{S}$ is solved if $\forall p \in \mathcal{S}$, either p is trivial or solved.*

The importance of the concept of solved systems can be seen from the fact the solved pairs of a solved system $\mathcal{S}$ form a *most general unifier* of $\mathcal{S}$ (see [11]).

3 Dougherty's Combinatory Unification Algorithm

Dougherty's algorithm is a complete, finitely-branching, polymorphic HOU algorithm. It is presented here as a set of non-determinsitic transformation rules that act on a system of unification pairs. Let $\Longrightarrow$ represent the application of a transformation rule to a system, $\Longrightarrow^+$ the transitive closure of $\Longrightarrow$ and $\Longrightarrow^*$ its reflexive transitive closure. We use $\uplus$ for the multiset sum of two multisets. It assumed that the order of the terms in the pairs is immaterial.

Polymorphism is an essential feature in Dougherty's algorithm even if the original terms are monomorphic. In the SXX'-narrow transform (given below) an applied variable head is replaced with the term $\boldsymbol{S}\ X'\ X''$ and then reduced. Applying this transform to the system $\{\langle X_{\iota\to\iota}\ a_\iota, b_\iota\rangle\}$ results in $\{\langle X'a(X''a), b\rangle\}$. Because the type of a is ι the type of the fresh variable X'' must be $\iota \to ?$, but there is no way to determine what type $?$ should be. Similarly the type of X' must be $\iota \to ? \to \iota$, but again $?$ cannot be determined. Thus, in both cases $?$ is set to a type variable which may be instantiated during subsequent unification. In only this transformation are the types of the introduced variables not deducible.

Our presentation differs from that of Dougherty's by having WEAKREDUCE as a separate transformation, rather than a special case of HEADNARROW.

1. ADDARG
$$\{\langle t_1, t_2\rangle\} \uplus \mathcal{S} \implies \{\langle t_1\theta\ d, t_2\theta\ d\rangle\} \uplus \mathcal{S}\theta$$
Where d is a fresh constant and θ is the type-unifier of the types of t_1, t_2 and $\alpha \to \tau$ for fresh type variables α and τ (in case t_1 and t_2 are both of atomic type).
2. SPLIT:
$$\{\langle X\ \overline{t_n}, h\ \overline{s'_m}\ \overline{s''_n}\rangle\} \uplus \mathcal{S} \implies \{\langle h\ \overline{X'_m}\ \overline{t_n}, h\ \overline{s'_m}\ \overline{s''_n}\rangle\}\theta \uplus \mathcal{S}\theta$$
Where $\theta = (\theta_0, \theta_1)$, $\theta_0 = \{X \to h\ \overline{X'_m}\}$ and θ_1 is the mgu of $type(X)$ and $type(h\ \overline{s'_m})$. Each X'_i is a fresh variable.
3. WEAKREDUCE:
$$\{\langle \boldsymbol{I}\ t\ \overline{t_n}, s'\rangle\} \uplus \mathcal{S} \implies \{\langle t\ \overline{t_n}, s'\rangle\} \uplus \mathcal{S} \quad \text{(I-reduce)}$$
$$\{\langle \boldsymbol{K}\ t\ t'\ \overline{t_n}, s'\rangle\} \uplus \mathcal{S} \implies \{\langle t\ \overline{t_n}, s'\rangle\} \uplus \mathcal{S} \quad \text{(K-reduce)}$$
$$\{\langle \boldsymbol{S}\ t\ t'\ t''\ \overline{t_n}, s'\rangle\} \uplus \mathcal{S} \implies \{\langle t\ t''\ (t't'')\ \overline{t_n}, s'\rangle\} \uplus \mathcal{S} \quad \text{(S-reduce)}$$
4. HEADNARROW: The variables introduced by the rules are assumed to be fresh for the system in all cases. In all rules, $\theta = (\theta_0, \theta_1)$ where θ_0 is the syntactic unifier of a non-variable prefix subterm and the left-hand side of a suitably renamed

combinator axiom and θ_1 is the relevant type unifier. For example, in the first rule $\theta_0 = \{X \to \boldsymbol{I}\}$ and $\theta_1 = mgu(type(X), type(\boldsymbol{I}))$.

$$\{\langle X\ t\ \overline{t_n}, s'\rangle\} \uplus\ \mathcal{S} \qquad \Longrightarrow \{\langle t\ \overline{t_n}, s'\rangle\}\theta\ \uplus\ \mathcal{S}\theta \qquad \text{(I-narrow)}$$

$$\{\langle X\ t\ \overline{t_n}, s'\rangle\} \uplus\ \mathcal{S} \qquad \Longrightarrow \{\langle X'\ \overline{t_n}, s'\rangle\}\theta\ \uplus\ \mathcal{S}\theta \qquad \text{(KX-narrow)}$$

$$\{\langle X\ t\ t'\ \overline{t_n}, s'\rangle\} \uplus\ \mathcal{S} \qquad \Longrightarrow \{\langle t\ \overline{t_n}, s'\rangle\}\theta\ \uplus\ \mathcal{S}\theta \qquad \text{(K-narrow)}$$

$$\{\langle X\ t\ \overline{t_n}, s'\rangle\} \uplus\ \mathcal{S} \qquad \Longrightarrow \{\langle X'\ t\ (X''\ t)\ \overline{t_n}, s'\rangle\}\theta\ \uplus\ \mathcal{S}\theta \qquad \text{(SXX'-narrow)}$$

$$\{\langle X\ t\ t'\ \overline{t_n}, s'\rangle\} \uplus\ \mathcal{S} \qquad \Longrightarrow \{\langle X'\ t'\ (t\ t')\ \overline{t_n}, s'\rangle\}\theta\ \uplus\ \mathcal{S}\theta \qquad \text{(SX-narrow)}$$

$$\{\langle X\ t\ t'\ t''\ \overline{t_n}, s'\rangle\} \uplus\ \mathcal{S} \qquad \Longrightarrow \{\langle t\ t''\ (t'\ t'')\ \overline{t_n}, s'\rangle\}\theta\ \uplus\ \mathcal{S}\theta \qquad \text{(S-narrow)}$$

These four transformation rules are collectively known as the HUT-transformations. They are used alongside syntactic transformations DECOMP, ELIMINATE and TYPEUNIFY. For a system $\mathcal{S}$, its *derived* system is the system of type pairs formed by replacing each term in $\mathcal{S}$ with its type.

5. DECOMP:

$$\{\langle f\ \overline{t_n}, f\ \overline{s_n}\rangle\} \uplus \mathcal{S}\ \Longrightarrow\ \{\langle t_1, s_1\rangle \ldots \langle t_n, s_n\rangle\} \uplus \mathcal{S}$$

6. ELIMINATE:

$$\{\langle X, t\rangle\} \uplus \mathcal{S}\ \Longrightarrow\ \{\langle X, t\rangle\}\ \uplus\ \mathcal{S}\theta$$

where $\theta = \{X \to t\}$ and X does not occur in t.

7. TYPEUNIFY

$$\mathcal{S}\ \Longrightarrow\ \mathcal{S}\theta$$

Where θ is type unifier of the derived system of $\mathcal{S}$.

For any system $\mathcal{S}$, an exhaustive application of WEAKREDUCE, DECOMP and ADDARG results in a simple system $\mathcal{S}'$ with the same unifiers as $\mathcal{S}$ as proved by Dougherty. In [11] the following non-deterministic algorithm for enumerating C-unifiers is called U and is proven to be sound and complete.

1. Reduce the system to a simple system then apply some HUT-transformation out of an unsolved pair.
2. If at any point the system is syntactically unifiable by a pure substitution then optionally return a most general unifier of the system.

We gloss over what is meant by a 'pure' substitution as it is irrelevant to the discussion. Unfortunately U contains infinite computation paths in many cases where Huet's classical algorithm does terminate. Worse, even when restricted to pre-unification, the algorithm produces redundant unifiers.

Lemma 1. *If, for a system of unification pairs Σ, there exists a computation path of U that includes a* HEADNARROW *step, then there exists an infinite computation path of U on $\mathcal{S}$.*

Proof. For the application of a head-narrow step, there must exist a simple system $\mathcal{S}_1$ such that $\mathcal{S} \Longrightarrow^* \mathcal{S}_1$ by a series of U-steps and $\mathcal{S}_1$ includes a pair p of the form $\langle X_{\alpha\to\tau}\, t_\alpha\, \overline{t_n}, t'\rangle$. By the definition of simple system, we have that $head(t')$ is not a combinator. The following HEADNARROW step can then be applied:

$$\mathcal{S}_1 = p \uplus \mathcal{S}_2 \Longrightarrow_{SXX'-narrow} \{\langle X'_{\alpha\to\gamma\to\tau}\, t_\alpha (X''_{\alpha\to\gamma}\, t_\alpha)\, \overline{t_n}, t'\rangle\} \uplus \mathcal{S}_2\theta = \mathcal{S}_3$$

Where γ is a fresh *type* variable. The algorithm proceeds to reduce $\mathcal{S}_3$ to a simple system. As $X'\, t\, (X''\, t)\, \overline{t_n}$ and t' do not have identical rigid heads and neither has a combinator head, these terms are left unchanged by this phase of the algorithm. Thus, the result is a system $\{\langle X'\, t\, (X''\, t)\, \overline{t_n}, t'\rangle\} \uplus \mathcal{S}_4$. It is clear that the pair $\langle X'\, t\, (X''\, t)\, \overline{t_n}, t'\rangle$ is eligible for an identical HEADNARROW step to p, but the narrowable term has one more argument for the head symbol than that of p. This process obviously cannot terminate proving the lemma.

Lemma 2. *Even if U is restricted, such that no transformation steps are carried out on flex-flex pairs, U can still produce redundant unifiers.*

Proof. Consider the simple system $\{\langle Xa, a\rangle\}$. Then by a single application of I-narrow the unifier $\{X \to \boldsymbol{I}\}$ can be produced. Alternatively the derivation path $\{\langle Xa, a\rangle\}$ $\Longrightarrow_{SXX'-narrow}$ $\{\langle X'a(X''a), a\rangle\} \Longrightarrow_{K-narrow} \Longrightarrow$ $\{\langle a, a\rangle\}$ can be followed leading to the unifier $\{X \to \boldsymbol{SK}X''\}$. $\boldsymbol{SK}X'' =_c \boldsymbol{I}$ and is thus a redundant unifier.

Lemmas 1 and 2 show that Dougherty's algorithm, whilst interesting from a theoretical aspect, is not suitable for a practical implementation. In as yet unpublished work [4], Bentkamp et al. present a modification of the given-clause algorithm that deals with possibly infinite streams of unifiers. But even such a method would be unable to handle Dougherty's algorithm as almost all unification problems are likely to be non-terminating quickly leading to memory issues on difficult problems. Instead, we propose a modification to the algorithm that eliminates these unpleasant properties at the cost of completeness.

4 Restricted Combinatory Unification

The pair of problems identified with Dougherty's algorithm in the previous section are both linked to the SXX'-narrow transform. As this step introduces type variables, typing cannot be used to restrict its application. In our modification of Dougherty's algorithm, we remove this head-narrow step. As this step was the only one to introduce type variables, polymorphism can now be eliminated. The three polymorphic combinator axioms used in the HEADNARROW step now become an infinite set of monomorphic axioms. To this set two new axioms schemas are added related to the $\boldsymbol{C}$ and $\boldsymbol{B}$ combinators. These schemas are $\boldsymbol{B}X\,Y\,Z = X(Y\,Z)$ and $\boldsymbol{C}X\,Y\,Z = X\,Z\,Y$. The $\boldsymbol{C}$ and $\boldsymbol{B}$ combinators are redundant, in the sense that they can be defined in terms of $\boldsymbol{S}$, $\boldsymbol{K}$ and $\boldsymbol{I}$, yet their usage often makes combinatory terms smaller.

Below, the modifications to Dougherty's algorithm are presented. The resulting algorithm is referred to as Restricted Combinatory Unification or RCU. In the calculation of the unifier θ in the steps below, no type unification is required. Further Dougherty's syntactic transformation step TYPEUNIFY is no longer required resulting in DECOMP and ELIMINATE being the only syntactic transforms needed.

Definition 3. *The set of all variables contained in a system $\mathcal{S}$, denoted $vars(\mathcal{S})$, is divided into two disjoint subsets R and B. Members of R are referred to as red variables and members of B as blue variables. We define $R = \{X \in vars(\mathcal{S}) \mid X$ introduced by CX-narrow transform$\}$ and $B = vars(\mathcal{S}) - R$.*

1. The following WEAKREDUCE rules are added in addition to the three in Dougherty's algorithm.

$$\{\langle \boldsymbol{B}\ t\ t'\ t''\ \overline{t_n}, s'\rangle\} \uplus\ \mathcal{S} \qquad \Longrightarrow \{\langle t\ (t't'')\ \overline{t_n}, s'\rangle\}\ \uplus\ \mathcal{S} \qquad \text{(B-reduce)}$$

$$\{\langle \boldsymbol{C}\ t\ t'\ t''\ \overline{t_n}, s'\rangle\} \uplus\ \mathcal{S} \qquad \Longrightarrow \{\langle t\ t''\ t'\ \overline{t_n}, s'\rangle\}\ \uplus\ \mathcal{S} \qquad \text{(C-reduce)}$$

2. The SXX'-narrow step is removed from Dougherty's rules and the following HEADNARROW steps are added. In all cases, θ is the syntactic (first-order) unifier of a non-variable prefix subterm and the left-hand side of a suitably renamed combinator axiom.

$$\{\langle X\ t\ t'\ \overline{t_n}, s'\rangle\} \uplus\ \mathcal{S} \qquad \Longrightarrow \{\langle X'\ (t\ t')\ \overline{t_n}, s'\rangle\}\theta\ \uplus\ \mathcal{S}\theta \qquad \text{(BX-narrow)}$$

$$\{\langle X\ t\ t'\ t''\ \overline{t_n}, s'\rangle\} \uplus\ \mathcal{S} \qquad \Longrightarrow \{\langle t\ (t'\ t'')\ \overline{t_n}, s'\rangle\}\theta\ \uplus\ \mathcal{S}\theta \qquad \text{(B-narrow)}$$

$$\{\langle X\ t\ t'\ \overline{t_n}, s'\rangle\} \uplus\ \mathcal{S} \qquad \Longrightarrow \{\langle X'\ t'\ t\ \overline{t_n}, s'\rangle\}\theta\ \uplus\ \mathcal{S}\theta \qquad \text{(CX-narrow)}$$

Where X is not a red variable

$$\{\langle X\ t\ t'\ t''\ \overline{t_n}, s'\rangle\} \uplus\ \mathcal{S} \qquad \Longrightarrow \{\langle t\ t''\ t'\ \overline{t_n}, s'\rangle\ \}\theta\ \uplus\ \mathcal{S}\theta \qquad \text{(C-narrow)}$$

The reason for the restriction on the CX-narrow step is to prevent infinite computation paths such as $\langle Xab, s\rangle \Longrightarrow \langle X'ba, s\rangle \Longrightarrow \langle X''ab, s\rangle \Longrightarrow \ldots$. The convention that no transformations are carried out on solved or trivial pairs is adopted.

Restricting Dougherty's algorithm would be of little interest if the restricted version suffered the same problems as the original. This is not the case with RCU.

Theorem 1. *Every sequence of RCU transformations terminates*

Proof. A proof sketch is provided here (see Appendix A for full proof). Dougherty proves that the set of transforms WEAKREDUCE, DECOMP and ADDARG are terminating. We reduce the proof of termination of RCU to this proof by showing that the remaining transformation can only appear finitely many times on a computation path.

- Each of transforms ELIMINATE and SPLIT, reduce the number of unsolved variables in a system by 1, whilst all other transforms maintain or reduce this measure.

- For HEADNARROW, consider the measure $(\sum_{v \in vars(\mathcal{S})} size(type(v))\ ,\ \#\mathcal{S})$ where $\#\mathcal{S}$ is the number of blue variables in a system $\mathcal{S}$. HEADNARROW reduces this measure whilst WEAKREDUCE, DECOMP and ADDARG maintain or reduce it.

An obvious corollary of Theorem 1 is that RCU is not a complete HOU algorithm. It does not, in general, find the csu. It would be of interest to compare RCU with other restricted forms of higher-order unification such as pattern unification [22]. However, it is not readily comparable with pattern unification and similar restrictions, as these tend to be restrictions on the *input terms* whilst RCU is a restriction on the *transformations*. As such, it most closely resembles depth-bound version of Huet's algorithm. We have not studied the complexity of RCU, but in Section 6, empirical evidence is presented that suggests its performance is reasonable.

5 Imperfect Filtering

In first-order provers, unification is generally carried out via term-indexes. Let R be a relationship on terms. A term-indexing data-structure stores terms in a manner that facilitates the rapid retrieval of all terms l such that $R(l, t)$ for some *query term* t. In the context of theorem-proving, the relationship R could be "is unifiable with", "is a generalisation of" etc. Indexing structures for first-order theorem proving have been intensively studied and efficient indexing structures such as substitution trees, fingerprint indexes and perfect discrimination trees have been developed.

These structures are either *perfect* or *imperfect* depending on whether they return all and only those terms that match the query, or they return some sub/superset of the same. Substitution trees stores substitutions in their nodes. Variables of the form $*_i$ are used to denote substitutions in the tree. The equation $*_i = t$ represents the substitution of the variable $*_i$ by the term t. A path from the root to a leaf represents a term formed by the composition of substitutions on the path (view Figure 1). Substitution trees act as perfect filters for first-order unification. This is achieved by traversing the tree left-right depth-first. In the root, the query term is unified with $*_0$. Each time we move down to a node $*_i = t$ the current unifier is extended with the unifier of $(*_i, t)$ in what is known as *incremental unification*. On backtracking, the unifier is reset to its previous value.

With respect to higher-order unification, some work has been carried out on developing indexing data structures [20] [24], but has gained little acceptance. In [28] the author suggests that the reason for this is the complexity or undecidability of many of the operations required to build and maintain higher-order indexing structures. For example, higher-order unification, anti-unification and matching are all either undecidable or have large complexities. This is a daunting obstacle to developing perfect higher-order indexing structures. However, it does not preclude the development of imperfect filters.

Substitution trees as imperfect filters we have modified substitution trees to act as imperfect filters for higher-order unification. The insights behind this modification are three-fold:

1. If two terms disagree on a function symbol that is not below an applied variable or combinator, then the terms have no higher-order unifier

2. Two terms that are first-order can only have a first-order unifier
3. For two terms t_1 and t_2, if all the higher order subterms of t_1 are at or under a variable position of t_2 and vice versa than the terms only have a first order unifier (with the caveat that both are linear).

We introduce two useful Lemmas that will allow us to explain our approach.

Lemma 3. *If $t|_p = s$ for first-order position p, then $(t\theta|_p) = s\theta$ for all substitutions θ.*

Proof. Proof is by induction on the length of p. If $p = \epsilon$ then $t\theta|_\epsilon = t\theta = s\theta$. In the inductive case, $t\theta|_p = t\theta|_{p'.i} \stackrel{IH}{=} (\zeta\ s'_1 \dots s'_{i-1}, s, s'_{i+1} \dots s'_n)\theta|_i = (\zeta\ s'_1\theta \dots s'_{i-1}\theta, s\theta, s'_{i+1}\theta \dots s'_n\theta)|_i = s\theta$.

Lemma 4. *Let p be a first-order position in terms t_1 and t_2. Then t_1 and t_2 have no higher-order unifiers if:*

1. *Both $head(t_1|_p)$ and $head(t_2|_p)$ are first order and $head(t_1|_p) \neq head(t_2|_p)$*
2. *$t_1|_p = X$, $X \in vars(t_2|_p)$ and $X \neq t_2|_p$ or vice versa.*

Proof. Assume that θ is a unifier of t_1 and t_2. Let $t_1|_p$ be s_1 and $t_2|_p$ be s_2. By Lemma 3, $(t_1|_p)\theta = s_1\theta$ and $(t_2|_p)\theta = s_2\theta$. Therefore, we must have that $s_1\theta = s_2\theta$. In case (1), $s_1 = f\ \overline{s'_n}$, $s_2 = g\ \overline{t'_m}$ and $f \neq g$. However, $s_1\theta = (f\ \overline{s'_n})\theta = f\ (\overline{s'_n\theta}) \neq g\ (\overline{t'_m\theta}) = (g\ \overline{t'_m})\theta$ and thus θ cannot be a unifier. Case (2) is similar.

Lemma 4 indicates how substitution trees can be used as imperfect filters. Prior to inserting a term into the substitution tree, all higher-order subterms are replaced with special sort-correct constants not appearing in the input. Call the constant $\#$. When performing incremental unification, $\#$ unifies with all terms. If a CLASH or OCCURSCHECK (terminology from [1]) is encountered, it must be at a first-order position and the search is backtracked.

Consider searching the tree in Figure 1 for unifiers of $g(f\ X)b$. The original substitution is $\sigma_0 = \{*_0 \to g(f\ X)b\}$. In the root, the substitution is extended to unify $*_0$ with $g\ *_1\ *_2$ resulting in $\sigma_1 = \sigma_0 \cup \{*_1 \to f\ X, *_2 \to b\}$. In the left child, $*_1$ is unified with $\#$. This succeeds without adding anything to the unifier, so $\sigma_3 = \sigma_2$. Finally, in the left-most leaf, $*_2$ is unified with b, again succeeding with the empty substitution. The search then backtracks and attempts to enter the second-to-left leaf. This requires unifying $*_2$ which is bound to b, with c and fails due to CLASH. The terms $g(f\ X)b$ and $g(\boldsymbol{I}\ a)c$ can have no unifiers as symbols b and c which are not below a variable or combinator disagree.

The set of terms eventually returned by the query is $\{g(X\ a)b, g(f\ a)b, g(f(Z\ d))b\}$. RCU can now be run term-to-term between the search term and all terms in this set. However, as mentioned above, in at least two cases, higher-order and first-order unification coincide and therefore there is no need to run RCU. Note that in the second case, the linearity constraint is necessary. Consider unification of terms $f\ X\ X$ and $f(\boldsymbol{K}\ a)(Y\ c\ a)$ which leads to the higher-order unification problem $\boldsymbol{K}\ a \stackrel{?}{=} Y\ c\ a$. This insight is formalised in the following lemmas.

Lemma 5. *Let t_1 and t_2 be first-order terms. Then, for any computation path $\{\langle t_1, t_2\rangle\} \Longrightarrow^* \mathcal{S}'$, we have:*

1. *All terms in $\mathcal{S}'$ are first-order.*
2. *In the computation path, only* DECOMP *and* ELIMINATE *are used.*

Proof. Proof by induction on the length of the $\Longrightarrow^*$ path. The base case is trivial. In the inductive case, after p steps the original system is transformed into $\{\langle s_1, s_2\rangle \ldots \langle s_{n-1}, s_n\rangle\}$ where each s_i is first-order by the induction hypothesis. In the $p+1$ step, an arbitrary pair $\langle s_i, s_{i+1}\rangle$ is transformed. If either s_i or s_{i+1} is a variable than the transformation is ELIMINATE and the resulting system again contains only first-order terms. Otherwise s_i and s_{i+1} are of the form $\zeta\,\overline{t_m}$ and $\zeta\,\overline{r_m}$ and the only applicable step is DECOMP. After performing DECOMP, the resulting system is $\{\langle s_1, s_2\rangle \ldots \overline{\langle t_m, r_m\rangle} \ldots \langle s_{n-1}, s_n\rangle\}$ and again all terms are first-order.

Lemma 6. *Call a pair $\langle t_1, t_2\rangle$* nearly first-order *if both terms are linear and if $t_1|_p$ is higher-order than $t_2|_{p'}$ is a variable for some $p' \leq p$ and vice versa. For any computation path $\{p\} \Longrightarrow^* \mathcal{S}'$, where p is nearly first-order, we have:*

1. *All pairs in $\mathcal{S}'$ are nearly first-order.*
2. *In the computation path, only* DECOMP *and* ELIMINATE *are used.*

Proof. Proof is similar to the proof of the previous lemma.

Lemmas 5 and 6 show that in some cases only DECOMP and ELIMINATE are applicable. But these two transforms form a sound and complete unification algorithm for syntactic first-order unification [1] and thus, in these cases higher-order unification collapses to first order unification. Our results are presented in terms of RCU, but are in reality general.

We make use of the above as follows. Let $\sigma = \cup_{i=0}^{i=n}\sigma_i$ be the unifier formed during traversal into a leaf. If σ does not bind any variable to a term containing a $\#$, and during the computation of σ, no term was unified with $\#$, then both query and index terms are first-order. The single mgu of the two is σ. On the other hand, if $\sigma = \sigma' \cup \{X \rightarrow t[\#]|_p\}$, we have the second case where both terms are linear and higher-order subterms are beneath variable position. Unification between the terms could still fail on OCCURSCHECK if $\#$ represents a term containing X. In this case, first-order unification is rerun *term-to-term* between the 'dehashed' query and index terms. For example, when traversing into the right-most leaf in Figure 1, we succeed with unifier $\sigma = \{*_1 \rightarrow f\,X, \ldots, X \rightarrow \#\}$. First-order unification is then rerun between terms $g(f\,X)b$ and $g(f(Z\,d))b$ and succeeds with unifier $\{X \rightarrow Z\,d\}$.

An alternative solution would have been to replace different higher-order subterms with different special constants and maintain a lookup table of special constants to terms. On binding a variable to a term, all special constants are replaced by the terms they represent. This solution has the advantage of early failure on OCCURSCHECK at the expense of extra memory.

Note that if the input problem is first-order, then all terms in the index and all query terms will be first-order. Thus, in this case, unification will always be first-order unification, and the addition of RCU to Vampire is *graceful* in the sense of [3].

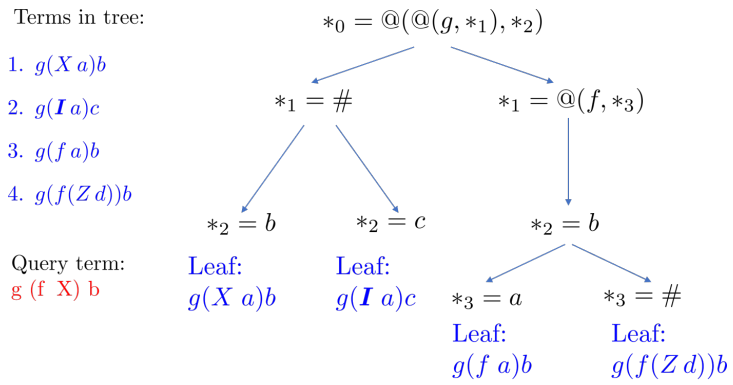

Fig. 1: Example of substitution tree being used as a filter for higher-order unification. The query will return terms (1),(3) and (4).Higher-order unification only needs to be run on the pair $\langle g(f\ X)b,\ g(X\ a)b\rangle$

6 Experimental Results

As with most successful first-order provers, Vampire is a portfolio prover. In finding a proof, it runs a set of *strategies* known as a *schedule*. Each strategy is a predefined set of proof search parameters. Normally, if a problem is solvable, it is solvable within a short space of time using a particular strategy. We created two custom higher-order schedules that include various options linked to higher-order proof search. These options are described in Table 1. Further details can be found in [5].

To evaluate the effect of combinatory unification, two experiments were run. In the first experiment, we ran Vampire twice using one of its higher-order schedules, once with combinatory unification forced on and once with it forced off, over the higher-order portion of the TPTP library. The time limit was 500s. The results of this experiment can be found in Table 2. Forcing combinatory unification on leads to 90 new proofs, demonstrating its value.

In the second experiment, the CASC-2018 versions of leading higher-order theorem provers, Leo-III version 1.3 and Satallax version 3.3 [6] were run on the monomorphic higher-order section of the TPTP library alongside Vampire HOL. Attention was restricted to problems that are not known to be satisfiable as both Leo-III and Vampire are incomplete. This leaves 2727 problems which are either theorems, unsatisfiable or unknown. A (wall-clock) time limit of 520s per problem was used. The results of this experiment can be found in Table 3. Despite the fact that Leo-III and Satallax have been tuned to the higher-order portion of the TPTP library, Vampire is reasonably close to them. It is conjectured that with more time to fettle the higher-order schedules, Vampire's figures would be even closer.

We also tested Vampire against Leo-III and Satallax on a set of 1253 benchmarks generated by Sledgehammer, kindly made available to us by the Matryoshka team. Fol-

Table 1: Higher-order parameters in Vampire

Option Name	Description
equality_to_equiv	A preprocessing option. If set on, equality between boolean terms is converted into equivalence
always_use_proxies	A preprocessing option. If set on, logical constants are always translated to 'proxy' functions
add_func_ext_ax	Adds a set of functional extensionality axioms to the problem based on the sorts present in the input
extended_narrowing	Switches on an inference rule similar to primitive substitution. An attempt is made to 'guess' the logical structure of literals with variable heads.
hol_constant_elimination	Switches on a set of inference rules to handle logical proxies.
hol_short_circuit_eval	An inference rule that allows logical proxies not at the literal level to be replaced by boolean values.
comb_unif	Switches on off combinatory unification as described in this paper
comb_select_val	Defines how to handle look-ahead literal selection when running with combinatory unification. This option can be set to actual in which case combinatory unification is run to ascertain the number of inferences possible were a particular literal to be selected. Other options include setting a penalty for potential inferences that use combinatory unification.
combinator_elimination	This option can be set to axioms in which case combinators are axiomatised. It can be set to inference_rules to enable a set of inferences that rewrite fully applied combinators. Axioms and rewriting can be used together by setting the option to both
add_combinators	The add_combinators heuristically adds combinators to the problem based on the sorts present in the input. A further option controls the number of combinators added.

lowing their naming convention, these benchmarks are referred to as SH-λ. For this test all provers were run with a 300s CPU time limit.

The experiments were performed on StarExec [30] nodes equipped with four Intel Xeon E5-2609 0 CPUs each with 8192 MB of memory and clock speed 2.40 GHz. Our experimental data is publicly available[2]. Amongst the TPTP benchmarks, Vampire solves 31 problems not solved by Leo-III and Satallax[3]. Out of these problems, 3

[2] The csv output files can be found at https://github.com/vprover/vampire_experimental_data/tree/master/CADE-2019-RCU

[3] Some of these problems are marked as being solvable by Satallax on the TPTP website. However, the CASC-2018 version of Satallax that we used in our tests was unable to find a proof

Table 2: Combinatory unification on and off

	Number Solved	Number of Uniques
RCU on	1894	90
RCU off	1893	89

Table 3: Number of problems proved theorem or unsat

	TPTP problems	SH-λ
Vampire HOL	1947	714
Leo-III	2097	668
Satallax	2095	513

(SEV016^5[4], SEV032^5 and SEU684^1) are difficulty rating 1.00 problems, meaning that they are unsolvable by any current theorem prover. Amongst the 31 problems 18 are solved by strategies that utilise combinatory unification showing its value. On the SH-λ benchmarks, Vampire solves 53 problems that Leo-III and Satallax cannot. All 53 as well as 560 other problems are solved by strategies that use combinatoty unification.

Finally, we investigated the efficiency of combinatory unification. It would have been interesting to compare it against first-order unification, but this is not possible in Vampire as first order unification is carried out incrementally whilst RCU is carried out term-to-term. Instead, the amount of time spent on RCU in each run was recorded as well as the number of unifiers produced. The results can be found in Table 4. It can be seen that combinatory unification is not dominating the running time. Further the number of combinatory unifiers produced suggests that whatever the worst case complexity is, in practice the procedure is efficient.

7 Conclusion and Related Work

Pragmatic approaches to higher-order theorem proving include Otter-λ, a prover for lambda logic [2] developed by Michael Beeson. Lambda logic is a relatively weak extension of first-order logic. Also included is Cruanes' prover Zipperposition [8] which he extended to HOL. The prover rewrites using the combinator definitions which resembles our approach, but is less goal directed.

The approach we present here resembles that taken by the Leo-III higher-order prover [29]. Leo-III implements Huet's complete algorithm for higher-order unification, but then imposes a depth bound, thereby losing completeness. It is felt that RCU is more amenable to implementation within a first-order prover than depth-bound Huet's algorithm and therefore our approach is complementary. By retaining the ordering restrictions of the superposition calculus, our approach further resembles that of Leo-III which makes use of the computational path order (CPO) further losing completeness.

[4] Though this problem can be solved by the new version of E, Ehoh developed by Vukmirović et al.

Table 4: Efficiency of RCU

Problem Category	**Average time spent on combinatory unification (as % of total time)**	**Average number of unifiers produced per problem**
SYO	11.71%	63764
SEU	9.5%	42968
SET	9.89%	57339
NUM	9.55%	22779
ALG	11.08%	54723

Related to out work is the Matryoshka team's extension of superposition to lambda-free higher-order logic [3] and, in as yet unpublished work, to full higher-order logic [4]. Their approach aims for completeness and thus has to deal with possible infinite sets of unifiers. Due to the incompleteness of our method, there will certainly be problems provable by a complete calculus that are inaccessible to us. On the other hand, it appears likely that, at least on some problems, the overhead of dealing with infinite streams of possibly redundant unifiers will allow lightweight but incomplete solutions such as RCU to outperform complete methods. However, what promises to be an interesting empirical comparison of the two approaches is future work.

As far as restrictions to higher-order unification are concerned, our approach adds to a long list of attempts to devise useful restrictions. Foremost amongst these are *pattern unification* [22] and its generalisation by Libal and Miller [19]. Pattern unification has gained popularity, because it is decidable and mgus exist. Bounding the number of 'λ's that can appear in a unifier has been shown to make higher-order unification decidable [26]. Various other restrictions have been shown to either be decidable or undecidable in [18] and [25]. As far as we are aware, we are the first to address restrictions to combinatory higher-order unification algorithms.

On the mildly higher-order Sledgehammer benchmarks, Vampire with RCU outperformed full higher-order solvers. It remains to be investigated what role combinatory unification played in this. It also remains to be seen whether this results holds across larger benchmark sets. Other lines of investigation, include implementing Dougherty's algorithm, but utilising it in a limited form. Consider the following inference which we call HYPEREQRES:

$$\frac{t_1 \neq t_2 \vee t_3 \neq t_4 \ldots \vee t_{n-1} \neq t_n}{[]}\ \text{HYPEREQRES}$$

where Dougherty's algorithm is used to find a unifier θ that simultaneously unifies each pair of terms. As this leads directly to a refutation, the problem of redundant unifiers is circumvented. Likewise, polymorphism can be restricted to the unification algorithm and there is no need to introduce it to the prover's kernel. Finally, it would be interest to evaluate the usage of substitution trees as imperfect filters. On average what percentage of terms in the index are discarded? In practice, how often does first-order unification suffice?

Acknowledgements Thanks to Jasmin Blanchette, Alexander Bentkamp, Simon Cruanes and Petar Vukmirović for many discussions on aspects of this research. A big thanks to the Matryoshka team as a whole for sharing their benchmarks. We would also like to thank Andrei Voronkov, Michael Rawson, Alexander Steen and the maintainers of StarExec. Special thanks to Martin Riener for proof-reading the paper. The first author thanks the family of James Elson for funding his research.

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.
2. Michael Beeson. Lambda logic. In *Automated Reasoning: Second International Joint Conference, IJCAR 2004*, pages 4–8. Springer.
3. Alexander Bentkamp, Jasmin Christian Blanchette, Simon Cruanes, and Uwe Waldmann. Superposition for lambda-free higher-order logic. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning*, pages 28–46, Cham, 2018. Springer International Publishing.
4. Alexander Bentkamp, Jasmin Christian Blanchette, Sophie Tourret, Petar Vukmirović, and Uwe Waldmann. Superposition with lambdas. *submitted for publication*, 2019.
5. Ahmed Bhayat and Giles Reger. Set of support for higher-order reasoning. In *6th Workshop on Practical Aspects of Automated Reasoning (PAAR)*, pages 2–16, 2018.
6. Chad E. Brown. Satallax: An automatic higher-order prover. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, pages 111–117, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
7. Guillaume Burel. Embedding deduction modulo into a prover. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic*, pages 155–169, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
8. Simon Cruanes. Superposition with structural induction. In Clare Dixon and Marcelo Finger, editors, *Frontiers of Combining Systems*, pages 172–188, Cham, 2017. Springer International Publishing.
9. Łukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *Journal of Automated Reasoning*, 61(1):423–453, Jun 2018.
10. Flávio LC de Moura, Mauricio Ayala-Rincón, and Fairouz Kamareddine. Higher-order unification: A structural relation between huet's method and the one based on explicit substitutions. *Journal of Applied Logic*, 6(1):72–108, 2008.
11. Daniel J. Dougherty. Higher-order unification via combinators. *Theoretical Computer Science*, 114(2):273 – 298, 1993.
12. Gilles Dowek. Higher Order Unification via Explicit Substitutions. *Information and Computation*, 157(1-2):183–235, 2000.
13. Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72.
14. Peter Graf. *Substitution tree indexing*, pages 117–131. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.
15. Kryštof Hoder and Andrei Voronkov. Comparing unification algorithms in first-order theorem proving. In *Annual Conference on Artificial Intelligence*, pages 435–443. Springer, 2009.
16. Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science - TCS*, 1:27–57, 06 1975.
17. Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *International Conference on Computer Aided Verification*, pages 1–35. Springer, 2013.

18. Jordi Levy. Decidable and undecidable second-order unification problems. In *International Conference on Rewriting Techniques and Applications*, pages 47–60. Springer, 1998.
19. Tomer Libal and Dale Miller. Functions-as-constructors higher-order unification. In *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.
20. Tomer Libal and Alexander Steen. Towards a Substitution Tree Based Index for Higher-order Resolution Theorem Provers. In *5th Workshop on Practical Aspects of Automated Reasoning*, July 2016.
21. Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40(1):35–60, Jan 2008.
22. Dale Miller. Unification of simply typed lambda-terms as logic programming. 1991.
23. Lawrence C Paulson and Jasmin Christian Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. *IWIL-2010*, 1, 2010.
24. Brigitte Pientka. Higher-order term indexing using substitution trees. *ACM Trans. Comput. Logic*, 11(1):6:1–6:40, November 2009.
25. Christian Prehofer. Decidable higher-order unification problems. In *International Conference on Automated Deduction*, pages 635–649. Springer, 1994.
26. Manfred Schmidt-Schauß and Klaus U Schulz. Decidability of bounded higher-order unification. *Journal of Symbolic Computation*, 40(2):905–954, 2005.
27. Wayne Snyder and Jean Gallier. Higher-order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8(1-2):101–140, 1989.
28. Alexander Steen. *Extensional Paramodulation for Higher-Order Logic and its Effective Implementation Leo-III*. PhD thesis, Freie Universität Berlin, 2018.
29. Alexander Steen and Christoph Benzmüller. The higher-order prover Leo-III. In *International Joint Conference on Automated Reasoning*, pages 108–116. Springer, 2018.
30. Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. Starexec: A cross-community infrastructure for logic solving. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Automated Reasoning*, pages 367–373, Cham, 2014. Springer International Publishing.
31. Ulrich Wertz. *First-order theorem proving modulo equations*. PhD thesis, Max-Planck-Institut für Informatik, 1992.

A Restricted Combinatory Unification is Terminating

We prove the termination of an algorithm identical to RCU, except that reductions can be performed at all positions. That RCU is terminating follows immediately. A straightforward corollary of this is that RCU is incomplete.

Lemma 7. *For any (finite) system $\mathcal{S}$, if there exists an infinite RCU computation path on $\mathcal{S}$, then there exists a system $\mathcal{S}'$ such that $\mathcal{S} \Longrightarrow^* \mathcal{S}'$ and there exists an infinite computation path on $\mathcal{S}'$ that does not include the* ELIMINATE *or* SPLIT *transforms.*

Proof. Both ELIMINATE and SPLIT reduce the number of unsolved variables in $\mathcal{S}$. As $\mathcal{S}$ is finite, it can only contain a finite number of unsolved variables. A case analysis of the other rules shows that either they reduce the number of unsolved variables or leave it unchanged. Thus SPLIT and ELIMINATE can only be carried out a finite number of times. □

Based on Lemma 7, to prove that RCU is terminating, it suffices to prove that RCU without the eliminate and split transforms is terminating. Call the resulting set of transformation rules RCU−. Next a result analogous to 7 is proved with respect to the HEADNARROW transformation.

Definition 4 (Size). *The size of of a type σ is defined inductively as follows:*

- *σ is atomic, the $size(\sigma) = 0$*
- *$\sigma = \alpha \to \beta$, then $size(\sigma) = size(\alpha) + size(\beta) + 1$*

For a term t, its type is denoted by $\tau(t)$

Intuitively, $size(\sigma)$ is the number of '$\to$'s in σ.

Lemma 8. *For any (finite) system $\mathcal{S}$, if there exists an infinite RCU− computation path on $\mathcal{S}$, then there exists a system $\mathcal{S}'$ such that $\mathcal{S} \Longrightarrow^* \mathcal{S}'$ and there exists an infinite computation path on $\mathcal{S}'$ that does not include the head narrow transform.*

Proof. Consider the following measure on systems:

$$(\sum_{v \in vars(\mathcal{S})} size(\tau(v)) \ , \ \#\mathcal{S})$$

Where the pairs are compared lexicographically and $\#\mathcal{S}$ denotes the number of blue variables in $\mathcal{S}$ or equivalently, the cardinality of B. The transformation rules of RCU−, other than head narrow, keep this measure constant or reduce it. HEADNARROW reduces the measure, so there can only be a finite number of applications of head narrow. The former claim is demonstrated for a number of HEADNARROW steps:

1. KX-narrow. For a variable X to be eligible for a KX-narrow step, it must have type $\alpha \to \beta$. It is replaced by a term $\boldsymbol{K}_{\beta \to \alpha \to \beta} X'_{\beta}$. Clearly $size(\tau(X')) < size(\tau(X))$ and so the first item of the measure is reduced.
2. BX-narrow. For a variable X to be eligible for a BX-narrow step, it must have type $(\alpha \to \beta) \to \alpha \to \gamma$. It is replaced by a term $\boldsymbol{B}_{(\beta \to \gamma) \to (\alpha \to \beta) \to \alpha \to \gamma} X'_{\beta \to \gamma}$. Again $size(\tau(X')) < size(\tau(X))$ and so the first item of the measure is reduced.

3. CX-narrow. In this case, the size of the type of the variable being narrowed is not reduced. For a variable X to be eligible for a CX-narrow step, it must have type $\alpha \to \beta \to \gamma$. It is replaced by a term $\boldsymbol{C}_{(\beta\to\alpha\to\gamma)\to\alpha\to\beta\to\gamma} X'_{\beta\to\alpha\to\gamma}$. Here $size(\tau(X')) = size(\tau(X))$. However, each CX-narrow step replaces a blue variable with a red variable and therefore the second item of the measure is reduced.

□

Based on Lemma 8 to prove that RCU− is terminating, it suffices to prove that RCU− without the HEADNARROW transformation is terminating. This is precisely Dougherty's VT transformations which he has proven to be terminating in [11]. Therefore we have:

Theorem 2. *Every sequence of RCU transformations terminates*