# Higher Order Unification

Fabian Huch

Technische Universität München

June 2020

**Abstract**

Unification is an important problem in many areas of automated reasoning, mainly automated theorem proving. However, the problem is not decidable in higher-order logics. Still, practical algorithms exist that work in many cases. In this paper, we explain the semi-decision unification algorithm by Huet. We put special emphasis on the intuitive understanding how the algorithm works.

## 1 Introduction

Unification is the process of finding an assignment to free variables in terms $t$ and $t'$, such that they are equal under the assignment. While this is a computable problem for formulas of first-order logic, in higher-order logics, it is known to be undecidable [2]. Nevertheless, it is a important problem in the field of automated reasoning, as unification can be used, for instance, to search for proofs in automated theorem provers. These systems in particular typically use higher order logics, since they require users to state and reason about theorems formally — this is much easier to do in a calculus that allows quantifying over any term, rather than being restricted to first order variables.

To solve the problem, Huet presented a semi-decision procedure in [1], which we will explore in this paper. The original procedure does not assume the axiom of functional extensionality for $\eta$-conversion, to give a more general algorithm. Though typically, logics set up for automatization (for example, Isabelle/HOL) admit $\eta$-conversion. Hence we assume it as well in the main part of the paper, since it makes it easier to understand the key ideas.

The paper is structured as follows: In Section 2, we define the $\lambda$-calculus that we use throughout the paper, and explain basic concepts, symbols and notation; Next, in Section 3, we explain the algorithm in detail and prove its correctness. Lastly, we discuss the algorithm in a calculus without $\eta$-conversion in Section 4.

## 2 Preliminaries

The logic that we use in this paper is a simply typed lambda calculus, similar to that of Church [3]: Types $\tau, \tau', \ldots \in T$ are inductively defined over a set of

elementary types $T_0$ (whose elements are denoted by $\alpha, \beta, \gamma, \dots$) and a single type constructor $\to$:

$$\tau, \tau' \in T \Longrightarrow \tau \to \tau' \in T$$

The type constructor $\to$ is right-associative, so any type $\tau_1 \to (\cdots \to (\tau_n \to \alpha) \cdots)$ can be written as $\tau_1 \to \cdots \to \tau_n \to \alpha$. The *arity* of a type, i.e the number of arguments that need to be applied before an elementary type is returned, is then $n$.

Terms (denoted $t$, $u$, $\dots$) are defined as follows:

$$\begin{array}{rll} t := & C_\tau & \text{(constant)} \\ \mid & x_\tau & \text{(variable of type } \tau\text{)} \\ \mid & \lambda x_\tau . t & \text{(abstraction by a variable } x_\tau\text{)} \\ \mid & t\ u & \text{(application)} \end{array}$$

Atoms @ of this definition consist of constants (denoted by uppercase letters), and variables. Two atoms with same name, but different type, are considered different entities — though we will name distinct entities differently in this paper. A variable $x_\tau$ that occurs somewhere in $t$ is said to be *bound* in the context of $\lambda x_\tau . t$; if no such abstraction binding exists, it is called *free*. Should multiple abstractions for the same variable name be nested in a term, then only the innermost is a binder for occurrences of the variable. We only consider *well-typed* terms, i.e. terms for which the type can be derived from the following rules for the typing relation $t :: \tau$ (assumptions on top of the line, and conclusions below — notation inspired by [4, Ch. 2]):

$$\frac{}{@_\tau :: \tau} \qquad \frac{t :: \tau \to \tau' \qquad u :: \tau}{t\ u :: \tau'} \qquad \frac{t :: \tau}{(\lambda x_{\tau'} . t) :: \tau' \to \tau}$$

We further denote $\mathcal{T}(t) = \tau \iff t :: \tau$, and with $\mathcal{F}(t)$ the free variables in $t$.

**Abbreviations.** For brevity, we will omit type annotations if the type is not interesting. We abbreviate nested abstractions and applications:

$$\begin{array}{rcl} \lambda x_1 . (\lambda x_2 . (\dots \lambda x_n . t)) & \text{by} & \lambda x_1\, x_2\, \dots\, x_n . t \\ (\cdots ((t_1\ t_2)\ t_3)\ \cdots)\ t_n & \text{by} & t_1\ (t_2,\ t_3,\ \dots,\ t_n) \end{array}$$

Lastly, application binds stronger than abstraction, so $\lambda x . t_1\ t_2$ means $\lambda x . (t_1\ t_2)$.

**$\lambda$-conversions.** Equality between two terms is taken modulo *$\alpha$-conversion*, i.e. conflict-free renaming of bound variables, and denoted $=_\alpha$.

Terms in the calculus are evaluated using *$\beta$-reduction*: A term of the form $(\lambda x . t)\ u$ (called *$\beta$-redex*) can be reduced by replacing all free occurrences of $x$ in $t$ by $u$, if no free variable in $u$ would be captured by a binder in $t$. To fulfill this side-condition, $\alpha$-conversion might need to be applied first. We write $\Rightarrow_\beta$ for $\beta$-reduction.

Additionally, *$\eta$-expansion and reduction*, i.e. expanding a term $t :: \tau' \to \tau$ to $\lambda x_{\tau'} . t\ x_{\tau'}$ (and, vice versa, collapsing it) might be admissible, depending on whether the axiom of functional extensionality is assumed.

**Substitution.** A substitution $\sigma$ is a type-preserving mapping from (finitely many) free variables to terms:

$$\sigma := \{x_1 \mapsto t_1,\ \dots,\ x_n \mapsto t_n\} \qquad x_i \neq x_j \text{ for } i \neq j$$

Applying $\sigma$ to a term $t$ — written $\sigma t$ — can be formally defined as a mapping from term to term ($\beta$-reducing the result):

$$\sigma t = (\lambda x_1 \ldots x_n.\, t)\ (t_1,\ \ldots,\ t_n)$$

Notably, substitutions can *not* refer to variables bound in an outer scope. For example, applying $\{y \mapsto x\}$ to the term $\lambda x.\, y$ produces:

$$(\lambda y\, x.\, y)\ x =_\alpha (\lambda y\, z.\, y)\ x \Rightarrow_\beta \lambda z.\, x$$

Multiple substitutions can be composed; the evaluation order is right to left, since the rightmost substitution would be applied first to term.

$$\sigma\rho = \{x \mapsto t \mid t = \sigma(\rho x) \wedge t \neq x\}$$

Moreover, substitutions are often compared over a set of variables $V$:

$$\sigma \underset{V}{=} \rho \iff \forall x \in V.\, \sigma x = \rho x$$

**Unifier.** A unifier for two terms $t$ and $t'$ (in other words, the *disagreement pair* $\langle t,\, t' \rangle$) is a substitution $\sigma$ such that $\sigma t =_\alpha \sigma t'$. For a set of disagreement pairs (a *disagreement set*), a unifier is a substitution that unifies all pairs pointwise. We denote the set of unifiers as $\mathcal{U}(...)$, input being either a disagreement pair or a disagreement set.

Finally, $[n]$ denotes $\{1, \ldots, n\}$.

# 3 Basic Algorithm

Intuitively, the basic idea of the unification algorithm is to compare the terms that should be unified in a normal form that has some *innermost* atom. By comparing these atoms, a most general (but finite) set of substitutions can be derived, which reduce the complexity of the unifier for the remaining terms (if one exists).

The structure of the algorithm can be described as a *matching tree* that contains at each node a set of 'disagreeing' pairs of formulas (that still need to be unified), and at each outgoing edge the next substitution to apply. The matching tree might be infinite, but is finite at each level, and if a unifier exists, then a success node will be located on a finite level. Hence, to find it, one only needs to traverse the tree in a suitable fashion (for instance, breadth-first search).

## 3.1 Normal Form

To effectively compare two terms, we need to have them in $\beta\eta$-normal form. A term is $\beta$-normal if it contains no $\beta$-redexes, i.e. there is no $(\lambda x.\, t)\ u$ that could be $\beta$-reduced; then, it can be written as

$$\lambda x_1 \ldots x_n.\, @\ (t_1,\ \ldots,\ t_p)$$

We call @ the *head*, $\{x_1, \ldots, x_n\}$ the *binder*, and $\lambda x_1 \ldots x_n.\, @$ the *heading* of the term. The terms $t_1, \ldots, t_p$ are its *arguments*, which must be $\beta$-normal again. A term is called *rigid* if its head is a bound variable or constant; otherwise it is called *flexible*. The intuition is that a rigid heading always stays the same.

**Lemma 1.** *A rigid term does not change its heading under any substitution, modulo $\alpha$-conversion.*

*Proof.* From the definition of substitution application. □

Generally, not all $\lambda$-terms have a $\beta$-normal form — consider $(\lambda x.\, x\, x)\,(\lambda x.\, x\, x)$, which does not change after a $\beta$-reduction. However, such a term can also not be assigned a type, which leads us to the following the theorem:

**Theorem 1.** *Any well-typed term can be converted to $\beta$-normal form by a finite sequence of $\beta$-reductions.*

*Proof Sketch.* Let $t$ be a term that contains $\beta$-redexes. Without loss of generality, we choose a subterm $t' = (\lambda x'_\tau.\, t_1)\, u$ such that $u$ is $\beta$-normal. Measuring the complexity by the *type of the redex body* $\lambda x'_\tau.\, t_1$, we can see that it decreases by the reduction: The redex that we reduce has type $\tau' \to \tau$, where $\tau$ denotes the type of $t_1$. New redexes might appear where the bound variable that was reduced is the body of an application, i.e. in subterms $x'_\tau\ t''$. It follows that the body type of newly introduced redexes must be $\tau'$, which is obviously less complex than $\tau' \to \tau$. Thus the overall complexity – measured by count of redexes for each type complexity – decreases in each $\beta$-substitution. It reaches zero for a $\beta$-normal term. □

This sketch only shows that there is at least one terminating sequence of $\beta$-reductions, which is not very practical; in fact, there is a stronger result stating that any sequence of $\beta$-reductions will terminate and yield the same unique normal form. However, the proof for this is a lot more complex, and can be found in [5, Ch. 2].

Next, the conversion from $\beta$- to $\beta\eta$-normal form is simple. If a term

$$\lambda x_{1\tau_1}\ \ldots\ x_{n\tau_n}.\, @\,(t_1,\ \ldots,\ t_p)$$

has type $\tau_1 \to \cdots \to \tau_n \to \tau_{n+1} \to \cdots \to \tau_m \to \alpha$ $(m > n)$, perform $\eta$-expansion $m - n$ times. This way, a normal-form term is obtained whose type is only dependent on its binder types and an elementary return type. In this section, all terms are assumed to be $\beta\eta$-normalized.

## 3.2 Matching Tree

To unify two terms $t$ and $t'$, the algorithm constructs a matching tree that stores the remaining disagreement set as nodes, and substitutions as edges. Fig. 1 shows schematically how the tree is constructed. We give an overview about the construction in this section; the `simplify` and `match` operations are then explained in much greater detail in the following two sections.

The procedure starts with the singleton set containing the disagreement pair $\langle t,\, t' \rangle$. This set is first *simplified*, which means that all pairs $\langle t_i,\, t'_i \rangle$ where both terms are rigid are broken down. `simplify` returns a node $N_1$, which is a set of flexible/flexible and (at least one) flexible/rigid disagreement pairs; if all remaining pairs would be flexible/flexible, then finding a unifier is trivial, and it returns a success node $N_\sigma$ with one unifier instead. Similarly, if a pair turns out to be non-unifiable, `simplify` returns a failure node $N_F$.

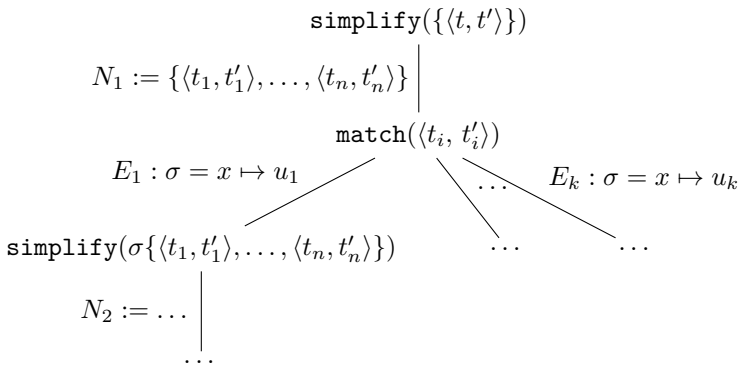


Figure 1: Process of creating the matching tree

Next, an arbitrary flexible/rigid pair from the simplified set is *matched*, i.e., from the structure of the rigid term, all possible substitutions for the head of the flexible term are derived. Each substitution creates an edge $E_i$, which points to the next node, i.e. the simplified result of applying the substitution to all terms in the previous node. Recursion for a branch ends once `simplify` returns a failure node or no more substitutions can be found; the algorithm stops once a success node is found or once the tree is exhaustively searched. However, if no unifier exists, the search might not terminate. Fig. 2 shows an exemplary matching tree (which only stores nodes $N_i$ and edges $E_i$). The `simplify` and `match` algorithms (as well as the lemmas required for the correctness proof) are presented in the following sections; finally, we prove correctness of the algorithm.

### 3.3 Simplifying Disagreement Sets

For the simplification of a disagreement set $S$, we first eliminate all rigid/rigid pairs $\langle t,\, t'\rangle \in S$. Suppose those terms in normal form:

$$t = \lambda x_1\ \ldots\ x_n.\ @\ (t_1,\ \ldots,\ t_p)$$
$$t' = \lambda x_1\ \ldots\ x_m.\ @'\ (u_1,\ \ldots,\ u_q)$$

Where $@$ and $@'$ are constants or bound variables. Should the headings be different (under $=_\alpha$), then the terms cannot be unified, so return $N_F$, and we have:

**Lemma 2.** $\texttt{simplify}(S) = N_F \implies \mathcal{U}(S) = \emptyset$

*Proof.* No substitution or $\beta$-reduction can change the heading by Lemma 1, so $\sigma t \neq \sigma t'$ for any $\sigma$ ($\langle t,\, t'\rangle \in S$). [1] □

If the headings are equal, then $n = m$ and $p = q$ (since types must also be equal). As a result, the unification only depends on the arguments. Hence $\langle t,\, t'\rangle$ is replaced by $\{\langle \lambda x_1\ \ldots\ x_n.\, t_i,\ \lambda x_1\ \ldots\ x_n.\, u_i\rangle \mid i \in [p]\}$ in the return node $N_i$. Essentially, this leads us to the following property:

[1] And since by construction of the procedure, the corresponding node is returned only in this case. This argument is the same for all the proofs in this section, and is omitted for brevity.

**Lemma 3.** $\texttt{simplify}(S) = N_i \implies \mathcal{U}(S) = \mathcal{U}(N_i)$

*Proof.* $\sigma \in \mathcal{U}(\{\langle \lambda x_1 \ldots x_n.\, t_i,\, \lambda x_1 \ldots x_n.\, u_i \rangle \mid i \in [p]\}) \iff \sigma \in \mathcal{U}(\langle t,\, t' \rangle)$ by Lemma 1 and since the headings of $t$ and $t'$ are equal. □

It is obvious that the procedure terminates, as the newly introduced terms have at least one less atom than the eliminated pair. Once all rigid/rigid pairs are eliminated in the disagreement set $S'$, all rigid/flexible pairs are swapped to form $S''$ (this does not affect the unifiers, as they unify the set point-wise). Finally, if at least one flexible/rigid pair exists, the $N_i := S''$ is returned as new node, and we have:

**Lemma 4.** *If* $\texttt{simplify}(S) = N_i$*, there must be* $\langle t,\, t' \rangle \in N_i$ *such that* $t$ *is rigid and* $t'$ *is flexible.*

*Proof.* Construction above. □

Otherwise, only flexible/flexible pairs are left. They do not impose too much structure, so we can directly construct a unifier (and return it in a success node $N_\sigma$): For each $y \in \mathcal{F}(S'')$ where $y :: (\tau_1 \to \cdots \to \tau_k \to \alpha)$, suppose $y \mapsto \lambda z_1 \ldots z_k.\, h_\alpha \in \sigma$, where $h_\alpha$ is an fresh free variable (unique for type $\alpha$):

**Lemma 5.** $\texttt{simplify}(S) = N_\sigma \implies \sigma \in \mathcal{U}(S)$

*Proof.* Apply $\sigma$ to a flexible term with atom $y$ of type $\tau_1 \to \cdots \to \tau_k \to \alpha$ ($k = p$ due to $\eta$-normal form):

$$\begin{aligned} \sigma[\lambda x_1 \ldots x_n.\, y\,(t_1,\, \ldots,\, t_p)] \;&=\; \lambda x_1 \ldots x_n.\, (\lambda z_1 \ldots z_p.\, h_\alpha)\,(t'_1,\, \ldots,\, t'_p) \\ &\Rightarrow_\beta \lambda x_1 \ldots x_n.\, h_\alpha \end{aligned}$$

For each substitution pair, both terms have the same type, thus they will be reduced to the same term. □

**Examples**

| | |
|---|---|
| $\{\langle y\ A,\, v \rangle, \langle \lambda x.\, C\,(x,\ D),\, \lambda x.\, C\,(x,\ A) \rangle\}$ | (break down $\langle \lambda x.\, C\ \ldots, \lambda x.\, C\ \ldots \rangle$) |
| $\rightsquigarrow \{\langle y\ A,\, v \rangle, \langle \lambda x.\, x,\, \lambda x.\, x \rangle, \langle \lambda x.\, A,\, \lambda x.\, D \rangle\}$ | (eliminate $\langle \lambda x.\, x,\, \lambda x.\, x \rangle$) |
| $\rightsquigarrow \{\langle y\ A,\, v \rangle, \langle \lambda x.\, A,\, \lambda x.\, D \rangle\}$ | ($A \neq D$ in the elimination) |
| $\rightsquigarrow N_F$ | |

| | |
|---|---|
| $\{\langle \lambda x.\, y\,(y\ x),\, \lambda x.\, v \rangle, \langle y\ v,\, y\ D \rangle\}$ | (only flexible/flexible pairs) |
| $\rightsquigarrow N_\sigma$ | ($\sigma = \{y \mapsto \lambda z.\, h_\alpha, v \mapsto h_\alpha\}$) |

## 3.4 Matching Disagreement Pairs

The `match` procedure has one flexible term $t$ and one rigid term $t'$ as arguments (as well as a set of variables $V$ for fresh variable generation):

$$\begin{aligned} t &= \lambda x_1 \ldots x_n.\, y\,(t_1,\, \ldots,\, t_p) \\ t' &= \lambda x_1 \ldots x_n.\, @\,(u_1,\, \ldots,\, u_q) \end{aligned}$$

Conveniently, the number of binders must be equal as both terms have the same type and must be $\eta$-normal. The `match` procedure returns a set of distinct possible substitutions for $y$. To that end, the structure of the rigid term is imposed on the flexible term, while keeping the substitution as general as possible.

If @ is a constant $C$, then one possibility is to *imitate* $C$ by $y$. Alternatively, $C$ could be the result of a *projection* on one of its arguments. On the other hand, if @ is a bound variable, only the projection case is admissible (since a bound variable can't be directly imitated).

**Imitation** To imitate the head $C$, $y$ is replaced by a term such that $C$ is the head of $t$ after $\beta$-reduction; this is achieved by any term with heading $\lambda z_1 \ldots z_p.\, C$.

The arguments should be free variables so they can be arbitrarily instantiated later on; however, they might also depend on a bound variable. Applying all bound variables to a fresh free variable $h_i \notin V$ is a solution that covers both, since any possible term can be derived by subsequent substitution and $\eta$-reduction.

Thus we have as replacement term:

$$\lambda z_1 \ldots z_p.\, C\,(h_1\,(z_1,\,\ldots,\,z_p),\,\ldots,\,h_q\,(z_1,\,\ldots,\,z_p))$$

The type of variable $h_i$ can be derived from $\mathcal{T}(C)$ and $\mathcal{T}(z_j) = \mathcal{T}(u_j)$.

**Projection** To project $y$ onto one of its arguments in a most general way, we first abstract all arguments, then choose one of the binders as head. This gives us $p$ distinct substitutions. Like in the imitation case, the applied arguments are most general.

$$\lambda z_1 \ldots z_p.\, z_i\,(h_1\,(z_1,\,\ldots,\,z_p),\,\ldots,\,h_m\,(z_1,\,\ldots,\,z_p)) \qquad \text{for } i \in [p]$$

The number of arguments $m$ is equal to the arity of $z_i$ (with $\mathcal{T}(z_j) = \mathcal{T}(u_j)$). Moreover, the types of $h_j$ depend on $\mathcal{T}(z_1), \ldots \mathcal{T}(z_p)$. Only the return type of the replacement matches that of @.

Because the replacements are as general as possible and cover all cases, if $\langle t,\, t' \rangle$ was unifiable, then it must be unifiable under one of the substitutions. However, to show termination (if a unifier exists), we need a complexity measure. For terms of form $t = \lambda x_1 \ldots x_n.\, @\,(u_1,\,\ldots,\,u_p)$, and substitutions $\sigma = \{y_1 \mapsto t_1,\,\ldots,\,y_k \mapsto t_k\}$, we define:

$$\pi(t) := p + \sum_{i=1}^{p} \pi(u_i)$$

$$\Theta(\sigma) := k + \sum_{i=1}^{k} \pi(t_k)$$

Now we can formulate the lemma:

**Lemma 6.** $\forall \rho \in \mathcal{U}(t,t').\, \exists \sigma \in \mathtt{match}(t,t',V), \eta.\, \rho \underset{V}{=} \eta\sigma$ *and* $\Theta(\rho) > \Theta(\eta)$

*Proof.* Let $\rho$ be an arbitrary unifier for $t$ and $t'$. Since @ is rigid and can't change heading from substitution by Lemma 1, $\rho$ must assign a term to $y$.

By construction of the procedure, all possible headings for $y$ that can unify $t$ and $t'$ were considered, and `match` must return $\sigma$ such that heading$(\sigma y)$ = heading$(\rho y)$. The number of arguments of $\sigma y$ and $\rho y$ must also be identical, as the heads are of same type and terms must be $\eta$-normal. Hence $\rho y$ is a term of form $\lambda z_1 \ldots z_p.\, @'\, (w_1,\, \ldots,\, w_m)$. Define

$$\eta := \{h_j \mapsto w_j \mid j \in [m]\} \cup (\rho - \{y \mapsto \rho y\})$$

($h_i, \ldots, h_m$ from the `match` construction above). Then $\rho \underset{V}{=} \eta\sigma$ holds, and

$$\Theta(\eta) \leq (m + \sum_{j=1}^{m} \pi(w_j)) + (\Theta(\rho) - (1 + \pi(\rho y)) = \Theta(\rho) - 1$$

thus $\Theta(\rho) > \Theta(\eta)$. □

**Example**

$$\begin{aligned}
&\langle \lambda x_1\, x_2.\, y\, (D,\, \lambda x_3.\, g\, x_3),\, \lambda x_1\, x_2.\, C\, (\lambda x_3.\, x_3\, x_2)\rangle \\
&\quad \rightsquigarrow y \mapsto \lambda z_1\, z_2.\, C\, (h_1\, (z_1,\, z_2),\, h_2\, (z_1,\, z_2)) && \text{(imitation)} \\
&\quad \rightsquigarrow y \mapsto \lambda z_1\, z_2.\, z_1 && \text{(first projection)} \\
&\quad \rightsquigarrow y \mapsto \lambda z_1\, z_2.\, z_2\, (h_1\, (z_1,\, z_2)) && \text{(second projection)}
\end{aligned}$$

## 3.5 Exemplary Matching Tree

Putting it all together, we can now construct a full example for a matching tree. In Fig. 2, we unify the pair $\langle \lambda x.\, y\, (C\, (y\, x)),\, \lambda x.\, C\, x\rangle$. This example covers most of the cases for `simplify` and `match`.

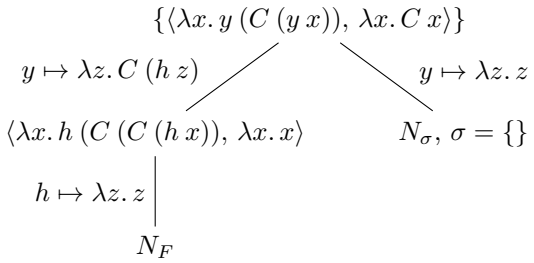


Figure 2: Exemplary matching tree

## 3.6 Proof of Correctness

We now prove the correctness of the algorithm, which means of *soundness* and *completeness*:

**Theorem 2.** *Let* $N_1 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_p} N_\sigma$ *be a path in a matching tree. Then* $\sigma\sigma_p \cdots \sigma_1 \in \mathcal{U}(N_1)$.

*Proof.* By induction on the path.

**Base:** If the path only consist of the single node $N_\sigma$, then $\sigma$ is a unifier for it by Lemma 5.

**Induction:** Have the induction hypothesis hold for $N_{i+1} \xrightarrow{\sigma_{i+1}} \cdots \xrightarrow{\sigma_p} N_\sigma$, i.e. $\xi \in \mathcal{U}(N_{i+1})$ for $\xi := \sigma\sigma_p \cdots \sigma_{i+1}$. Then for $N_i \xrightarrow{\sigma_i} N_{i+1}$, from the construction of the tree, there is a $\langle t, t' \rangle \in N_i$ such that $\sigma_i \in \texttt{match}(t, t', \mathcal{F}(N_i))$ and $\texttt{simplify}(\sigma_i N_i) = N_{i+1}$.

Hence $\xi \in \mathcal{U}(\sigma_i N_i)$ by Lemma 3. From this and because the disagreement pairs in $\sigma_i N_i$ are of form $\langle \sigma_i t_j, \sigma_i t'_j \rangle$, it follows that $\xi\sigma_i \in \mathcal{U}(N_i)$.

Thus $\sigma\sigma_p \cdots \sigma_i \in \mathcal{U}(N_i)$.

□

**Corollary 1.** *If there is a success node in the matching tree for $t$ and $t'$, then the path to the success node defines a unifier.*

Since the matching tree might be infinite, we cannot exhaustively search it. Thus for completeness we show the following:

**Theorem 3.** *Let $N_1$ be a node of a matching tree. If $\mathcal{U}(N_1) \neq \emptyset$, then the matching tree for $N_1$ contains a success node at a finite level.*

*Proof.* Let $V = \mathcal{F}(N_1)$. We inductively construct a path

$$N_1 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_{i-1}} N_i \xrightarrow{\sigma_i} N$$

such that either $N$ is a success node, or an intermediate node $N_{i+1}$ such that for any unifier $\rho \in \mathcal{U}(N_1)$, there exists an $\eta_{i+1} \in \mathcal{U}(N_{i+1})$ such that $\rho \underset{V}{=} \eta_{i+1}\sigma_i \cdots \sigma_1$. The complexity of this unifier must decrease in each step: $\Theta(\eta_i) > \Theta(\eta_{i+1})$.

**Base:** The path starts at $N_1$, which $\rho$ unifies by assumption.

**Induction:** Assume we have a path $N_1 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_{i-1}} N_i$ such that $N_i$ is an intermediate node that can be unified by $\eta_i$.

From Lemma 4, we know that there must be a $\langle t,\, t' \rangle \in N_i$ where $t$ is flexible and $t'$ rigid (which $\eta_i$ must also unify). By Lemma 6, $\exists \sigma \in \texttt{match}(t, t', V), \eta$ such that $\eta_i \underset{V}{=} \eta\sigma$.

We take $\sigma_{i+1} := \sigma$, and thus $N := \texttt{simplify}(\sigma N_i)$ as next node; $\eta_i$, i.e. $\eta\sigma$, unifies $N_i$, and hence $\eta \in \mathcal{U}(\sigma N_i)$. This means that $N$ can't be a failure node by Lemma 2; if $N$ is not a success node, then it must be an intermediate node, and from Lemma 3 we know that $\eta \in \mathcal{U}(N)$.

By Lemma 6, also $\Theta(\eta_i) > \Theta(\eta)$ holds, thus for $\eta_{i+1} := \eta$, $\Theta(\eta_i) > \Theta(\eta_{i+1})$. Moreover, $\rho \underset{V}{=} \eta_i\sigma_i \cdots \sigma_1 \underset{V}{=} (\eta\sigma)\sigma_i \cdots \sigma_1 = \eta_{i+1}\sigma_{i+1} \cdots \sigma_1$.

□

**Corollary 2.** *If $\exists \sigma.\, \sigma t = \sigma t'$, then the matching tree for $t$ and $t'$ contains a success node at a finite level.*

# 4 Omitting $\eta$-conversion

The axiom of functional extensionality might not be desirable in a calculus. Without it, $\eta$-conversion is not admitted. Still, the algorithms works very similar; the main difference is that more cases need to be accounted for in the `match` procedure.

Consider the $\beta$-normal disagreement pair $\langle t, t' \rangle$ that is the input for `match`. The terms then look as follows:

$$t = \lambda x_1 \ \ldots \ x_n . \, y \, (t_1, \, \ldots, \, t_p)$$
$$t' = \lambda x_1 \ \ldots \ x_n \, x_{n+1} \ \ldots \ x_m . \, @ \, (u_1, \, \ldots, \, u_q)$$

$n$ must be less than or equal to $m$ since any substitution can only affect the binder of a flexible term, i.e., increase it.

**Case** $n = m$ Then the procedure is very similar to the original. However, since $y$ can occur *partially applied* in a subterm of $t$, and $\lambda x . \, C \, x \neq C$ without $\eta$-reduction, additionally all cases where the binder has less than $p$ variables have to be considered. Let the following denote a replacement term with $p$ variables in the binder:

$$\lambda z_1 \ \ldots \ z_p . \, @' \, (h_1 \, (z_1, \, \ldots, \, z_p), \, \ldots, \, h_m \, (z_1, \, \ldots, \, z_p))$$

Then for each $k$ with $0 \leq k \leq min(p, m)$ we add:

$$y \mapsto \lambda z_1 \ \ldots \ z_{p-k} . \, @' \, (h_1 \, (z_1, \, \ldots, \, z_{p-k}), \, \ldots, \, h_{q-k} \, (z_1, \, \ldots, \, z_{p-k}))$$

to the set of possible substitutions, if the term is type-correct with respect to @.

**Case** $n < m$ . In this case, $m - n$ additional binders need to be introduced to unify the headings of $t$ and $t'$. This is straightforward:

$$y \mapsto \lambda z_1 \ \ldots \ z_p \, x_{n+1} \ \ldots \ x_m . \, @' \, (H_1, \, \ldots, \, H_m)$$

where $H_j := h_j \, (z_1, \, \ldots, \, z_p, \, x_{n+1}, \, \ldots, \, x_m)$, for $z_1, \, \ldots, \, z_p$, $m$, and $@'$ as in the original procedure. Moreover, a new case arises for the head $x_{n+k}$: This is not a projection case, since all $p$ arguments are already absorbed by $z_p$. It is rather an imitation, and is thus only applicable if $@ = x_{n+k}$. In the previous section, only constants were considered for imitation, since the whole binder of $t$ was inaccessible from $y$.

The properties of `simplify` and `match` don't change; in the proofs of the lemmas, the additional cases need to be accounted for. Other than that, they stay largely the same. We have noted explicitly where $\eta$-conversion or the $\eta$-normal property was used.

# References


[1] G. Huet. Unification in typed lambda calculus. In *International Symposium on Lambda-Calculus and Computer Science Theory*, pages 192–212. Springer, 1975.

[2] G. P. Huet. The undecidability of unification in third order logic. *Information and control*, 22(3):257–267, 1973.

[3] A. Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

[4] M. Wenzel, S. Berghofer, F. Haftmann, and L. Paulson. *The Isabelle/Isar Implementation*. Technische Universität München, 2019.

[5] R. Loader. *Notes on simply typed lambda calculus*. University of Edinburgh, 1998.