Efficiency of a good but not linear nominal unification algorithm

Weixi Ma¹, Jeremy G. Siek², David Thrane Christiansen³, and Daniel P. Friedman⁴

¹ Indiana University, Bloomington, Indiana, U.S.A.  
² Indiana University, Bloomington, Indiana, U.S.A.  
³ Galois, Inc., Portland, Oregon, U.S.A.  
⁴ Indiana University, Bloomington, Indiana, U.S.A.

Abstract

We present a nominal unification algorithm that runs in $O(n \times \log(n) \times G(n))$ time, where $G$ is the functional inverse of Ackermann’s function. Nominal unification generates a set of variable assignments if there exists one, that makes terms involving binding operations $\alpha$-equivalent. We preserve names while using special representations of de Bruijn numbers to enable efficient name management. We use Martelli-Montanari style multi-equation reduction to generate these name management problems from arbitrary unification terms.

1 Introduction and background

Equational theories over terms, such as the $\alpha$, $\beta$, and $\eta$ in the $\lambda$-calculus [Church, 1941], are a critical component of programming languages and formal systems. As users of logic programming languages and theorem provers, we desire such rules to be available out of the box. Two theories provide this convenience: Miller’s higher-order pattern unification [Miller, 1989] and Urban et al.’s nominal unification [Urban et al., 2004]. Higher-order pattern unification, the foundation of Isabelle [Paulson, 1986], $\lambda$Prolog [Nadathur et al., 1988], and Twelf [Pfenning and Schürmann, 1999], handles a fragment of the $\beta\eta$-rules. Nominal unification, the unification modulo the $\alpha$-rule, has inspired extensions of logic programming languages such as $\alpha$Prolog [Cheney and Urban, 2004] and $\alpha$Kanren [Byrd and Friedman, 2007], as well as theorem provers such as Nominal Isabelle [Urban and Tasson, 2005] and $\alpha$LeanTAP [Near et al., 2008]. Although these two theories can be reduced to one another [Cheney, 2005, Levy and Villaret, 2012], implementing higher-order pattern unification is more complicated because it has to deal with $\beta$-reduction and capture-avoiding substitution. An implementation of nominal unification, in which unification does not involve explicit $\beta$-reduction, is more straightforward and easier to formalize.

Concerning time complexity, Qian [1996] has proven that higher-order pattern unification is decidable in linear time. Still, it has been an open problem whether there exists a nominal unification algorithm that can do better than $O(n^2)$. Levy and Villaret [2012] give a quadratic-time reduction from nominal unification to higher-order pattern unification. Meanwhile, algorithmic advances by Paterson and Wegman [1978] and Martelli and Montanari [1982] for unification have inspired many improvements to the efficiency of nominal unification. Ideas like applying
swappings lazily and composing swappings eagerly and sharing subterms have also been explored. Calvè [2010] describes quadratic algorithms that extend the Paterson-Wegman and Martelli-Montanari’s algorithms with name (atom) handling; Levy and Villaret [2010] describe a quadratic algorithm that reduces unification problems to a sequence of freshness and equality constraints and then solves the constraints.

The inefficiency of these nominal unification algorithms comes from the swapping actions. To decide the $\alpha$-equivalence of two names, we need to linearly traverse a list whose length is the number of binders. One might try to replace these lists with some structures of better lookup efficiency, such as hashtables, but then composing two swappings would take linear time, and that operation is also rather frequent. Here, we present an algorithm that does not use swappings but instead represents names with de Bruijn numbers. De Bruijn numbers enable the use of persistent hashtables, in particular, Bagwell’s Hash Array Mapped Trie (HAMT). HAMTs provide efficient lookup and they use sharing to avoid the linear-time costs that would normally be associated with duplicating a hashtable [Bagwell, 2001].

We organize this paper as follows. In section 2, we provide an alternative representation of de Bruijn numbers that is suitable for unification. In section 3, we describe the abstract machines for name management and unification. In section 4, we discuss the time complexity of this algorithm. The proofs of our claims are in progress and are available at the authors’ Github:\footnote{https://github.com/mvcccccc/UNIF2018} formalized in Agda.

## 2 De Bruijn numbers should coexist with names

De Bruijn numbers are a technique for representing syntax with binding structure [de Bruijn, 1972]. A de Bruijn number is a natural number that indicates the distance from a name’s occurrence to its corresponding binder. When all names in an expression are replaced with their corresponding de Bruijn numbers, a direct structural equality check is sufficient to decide $\alpha$-equivalence. Many programming languages use de Bruijn numbers in their internal representations for machine manipulation during operations such as type checking. The idea of using names for free variables and numbers for bound variables, known as the locally nameless approach [Charguéraud, 2012], is employed for formalizing programming language metatheory [Aydemir et al., 2006, 2008]. Also, de Bruijn numbers, combined with explicit substitution, have been introduced in higher-order unification [Dowek et al., 2000] to improve the efficiency of unification.

Despite the convenience when implementing $\alpha$-equivalence, programs written with de Bruijn numbers are notoriously difficult for humans
to read and understand. What’s worse, as pointed out by Berghofer and Urban [2007], translating pencil-and-paper style proofs to versions using de Bruijn numbers is surprisingly involved: such a translation may alter the structure of proofs. Thus, recovering proofs with explicit names from proofs that use de Bruijn numbers is difficult or even impossible. Thus, for the sake of both readers and writers of proofs, it is worth providing an interface with names.

If our concern is simply deciding $\alpha$-equivalence between expressions, an easy way to use de Bruijn numbers while preserving names is to traverse the expressions, annotate each name with its de Bruijn number, then read back the expressions without numbers. This approach, however, does not work for unification, because it only contains the mapping from names to numbers. In unification modulo $\alpha$-equivalence, one frequently needs the mapping from numbers to names to decide what name to assign to a unification variable.

We represent de Bruijn numbers by static closures. Such closures preserve the mappings in both directions: names to numbers and numbers to names.

**Definition 2.1.** A closure is an ordered pair $\langle t; \Phi \rangle$ of a term $t$, defined in Figure 1, and a scope $\Phi$, where the scope is an ordered list of names for the binders in the enclosing context. The name of the innermost binder is written first in $\Phi$.

If the term of a closure is a name and the name appears in the scope of the closure, the closure itself represents a de Bruijn number. Consider the term $\lambda a.\lambda b.a$. The de Bruijn number of the name $a$ is 1 and the closure-representation of this number is $\langle a; (b a) \rangle$. We can retrieve the number-representation by finding the position of the first appearance of a given name. We define three operations on scopes: $\text{ext}$ extends the scope by consing a name to the front of the scope; $\text{idx} \rightarrow \text{name}$ yields the name of a given index; $\text{name} \rightarrow \text{idx}$ yields the location of the first appearance of a given name. When repeated names appear, the first one in a scope shadows the others.

Figure 2 defines the free and bound relations “constructively,” with de Bruijn numbers serving as evidence that variables are well-scoped. When a name, $a$, does not appear in the scope, $\Phi$, we say, “$a$ is free with respect to $\Phi$,” written as $\Phi \vdash \text{Fr} a$; when $a$’s first appearance in $\Phi$ is the position $i$, we say, “$a$ is bound at $i$ with respect to $\Phi$,” written as $\Phi \vdash \text{Bd} a i$. The $\text{BOUND}$ rule needs two premises to be algorithmic for either a name or index input, and prevent incorrect results caused by shadowing. For example, given the index 1 and the scope $(a a)$, the relation, $(a a) \vdash \text{Bd} a 1$, does not hold. Figure 3 defines the rules that decide $\alpha$-equivalent of two names w.r.t. their scopes, written as $\langle a; \Phi \rangle \approx \langle a; \Phi \rangle$.  

![Figure 4: Unification terms and problems](image-url)
3 Unification

In Figure 4, we introduce unification variables, abbreviated as \( \text{vars} \). Now, let's consider a simplified unification problem: a variable can only be instantiated by a name, that is, finding the unifier of two terms that share the same structure but differ in names and variables. A unifier consists of two parts: \( \sigma \) and \( \delta \).

A substitution, \( \sigma \), is a partial finite function from unification variables, \( X_i \), to terms, \( t_i \). For readability, we write \( \sigma \) as a set, \( \{X_1/t_1, \ldots, X_j/t_j\} \) and we write \( \{X/t\} \cup \sigma \) for extending \( \sigma \) with \( X/t \). For the simplified problems, we restrict \( t \) to a name.

A closure equation is a pair of two closures that are \( \alpha \)-equivalent. \( \Delta \) stands for a set of closure equations. We write \( \Delta \) as \( \{(t_1; \Phi_1) \langle t'_1; \Phi'_1 \rangle, \ldots, (t_i; \Phi_i) \langle t'_i; \Phi'_i \rangle\} \) and we write \( \{(t; \Phi) \langle t'; \Phi' \rangle\} \cup \Delta \) for extending \( \Delta \) with \( (t; \Phi) \langle t'; \Phi' \rangle \). \( \delta \) is a special form of \( \Delta \): for each equation in \( \delta \), the terms on both sides are variables. Given a variable \( X \), \( \delta(X) \) yields the list of closure equations where \( X \) appears at least once.

The simplified problem is about solving three kinds of problems: unifying a closure equation that has a name term on one side and a var term on the other side, abbreviated to N-V, and similarly N-N and V-V. We refer to an N-N or N-V equation as an \( e_n \) and refer to a V-V equation as an \( e_v \). Given two lists of these closure equations, \( p_\nu \) and \( p_\delta \), we first run the \( \nu \)-machine, defined in Figure 5, on \( p_\nu \) to generate a substitution. The \( \delta \)-machine, defined in Figure 6, then computes the final unifier on three inputs: the substitution resulting from the \( \nu \)-machine, \( \delta \), and a list of known variables, initialized by the domain of the substitution. If no transitions apply, the machine fails and the unification problem has no unifier.

Lemma 3.1. For all finite inputs, the \( \nu \)-machine and the \( \delta \)-machine terminate: for all finite inputs, the \( \nu \)-machine and the \( \delta \)-machine succeed with the most general unifier if and only if one exists.

Proof. By structural induction on the transitions of the machines.

\[
\sigma \vdash p_\nu \Rightarrow_\nu \sigma
\]

\[
\sigma_0 \vdash \epsilon \Rightarrow_\nu \sigma_0 \quad \text{[EMPTY]}
\]

\[
\langle a_1; \Phi_1 \rangle \approx \langle a_2; \Phi_2 \rangle
\]

\[
\sigma_0 \vdash p \Rightarrow_\nu \sigma_1
\]

\[
\sigma_0 \vdash \langle a_1; \Phi_1 \rangle = \langle a_2; \Phi_2 \rangle, p \Rightarrow_\nu \sigma_1 \quad \text{[N-N]}
\]

\[
\sigma_0 \vdash \langle X_2/a_2 \rangle \cup \sigma_0 \vdash \sigma \Rightarrow_\nu \sigma_1
\]

\[
\sigma_0 \vdash \langle X_2; \Phi_2 \rangle, p \Rightarrow_\nu \sigma_1 \quad \text{[N-V]}
\]

\[
\sigma; \delta; \epsilon \vdash \Rightarrow_\delta \sigma; \delta; \epsilon \quad \text{[EMPTY-D]}
\]

\[
\sigma; \epsilon \vdash \sigma; \epsilon \quad \text{[EMPTY-XS]}
\]

\[
\sigma; \epsilon \vdash \sigma; \epsilon \quad \text{[EMPTY-D]}
\]

\[
\sigma_0; x_0 \vdash \delta_0(X) \Rightarrow \text{pull} \sigma_0'; x_1
\]

\[
\sigma_0'; \delta_0 \vdash \delta_0(X) \Rightarrow \sigma_0; \delta_1
\]

\[
\sigma_0; \delta_0 \vdash \delta_0 \Rightarrow \sigma_1; \delta_1 \quad \text{[PULL]}
\]

\[
\sigma; \epsilon \vdash \Rightarrow \text{pull} \sigma; \epsilon
\]

\[
\langle a_1; \Phi_1 \rangle \approx \langle a_2; \Phi_2 \rangle
\]

\[
\sigma_0(X_1) = a_1, \quad \sigma_0(X_2) = a_2
\]

\[
\sigma_0; x_0 \vdash \Rightarrow \text{pull} \sigma_1; x_1
\]

\[
\sigma_0; x_0 \vdash \langle X_1; \Phi_1 \rangle = \langle X_2; \Phi_2 \rangle, p \Rightarrow \text{pull} \sigma_1; x_1
\]

\[
\langle a_1; \Phi_1 \rangle \approx \langle a_2; \Phi_2 \rangle
\]

\[
\sigma_0(X_1) = a_1, \quad X_2 \not\in \text{dom}(\sigma_0)
\]

\[
\{X_2/a_2\} \cup \sigma_0; \langle X_2; x_0 \rangle \vdash p \Rightarrow \text{pull} \sigma_1; x_1
\]

\[
\sigma_0; x_0 \vdash \langle X_1; \Phi_1 \rangle = \langle X_2; \Phi_2 \rangle, p \Rightarrow \text{pull} \sigma_1; x_1
\]
Now the question is how to generalize the previous algorithm, that is, given two arbitrary terms, where a variable may be instantiated by any term besides names, can we re-shape the two terms to create a proper input to the two machines?

Here we use the idea of Martelli and Montanari [1982]: finding the shared shape of two terms by computing the common parts and frontiers over a multi-equation. They define the common part of two terms to be a term obtained by superimposing, and the frontier to be the substitution that captures the differences between each term and the common part. For example, given distinct names \( a, b, \) and \( c \), distinct vars \( X \) and \( Y \), and two terms \( (aX) \) and \( (Y(bc)) \), the common part is the term \( (XY) \), and the frontier is the substitution \( \{ Y/a, X/(bc) \} \).

A multi-equation, defined in Figure 7, groups many closures to be unified, where the variable closures are on the left-hand side, and the non-variable closures are on the right-hand side.

The \( \rho \)-machine, defined in Figure 8, reduces an arbitrary nominal unification problem to \( p_v, p_b \), and a substitution where the codomain is unrestricted. Each \( \Rightarrow \) transition computes the common part and the frontier of a multi-equation. For readability, the sketch only shows the rules for multi-equations with two closures. A multi-equation with more than two closures is handled by simultaneously applying the rule to all closures. Unlike the Martelli-Montanari algorithm, the \( \rho \)-machine finds the maximum common part instead of the minimum. Thus, in the V-C, V-A, and V-A’ rules, we need two operators, \textbf{new-name} and \textbf{new-var}, to create new names and new variables for the shapes that fit with combinations and abstractions. The ordering of multi-equation is the same with Martelli-Montanari: for each multi-equation, we count the appearances of its left-hand side variables in other multi-equations of \( U \) and select the multi-equation associated with the smallest counter each time.

\textbf{Conjecture 3.1.} Given a unification problem, we run the \( \rho \)-machine, the \( \nu \)-machine, and the \( \delta \)-machine in sequence. The algorithm terminates; if the algorithm fails, i.e., no transitions apply, the problem has no solution; if the algorithm terminates, then the result of the \( \delta \)-machine is the mgu.

4 A note on time complexity

To improve time efficiency, we represent a scope with a counter and two persistent hashtables. One hashtable maps from names to numbers, the other maps from numbers to names, and the counter is used to track the de Bruijn number. When we extend a scope with a name, we extend the two hashtables with the corresponding maps and add one to the counter. A persistent hashtable, in practice, has constant time for update and lookup, although the worst case scenario could be \( O(\log(n)) \). Thus, \textbf{ext}, \textbf{idx→name}, and \textbf{name→idx} are all logarithmic time. In addition, using persistent structures avoids copying the entire data-structure when branching, in particular, during the C-C rule of the \( \rho \)-machine. Also, we implement \( \delta \) with a hashtable that maps from a variable to the list that contains its closure equations, i.e., the equation \( \langle X_1; \Phi_1 \rangle \approx \langle X_2; \Phi_2 \rangle \) exists in both \( X_1 \)'s entry and \( Y_2 \)'s entry in the hashtable. Now the \( \nu \)-machine and the \( \delta \)-machine are both worst case \( O(n \times \log(n)) \), where \( n \) is the sum of name and variable occurrences. The algorithm of Martelli-Montanari is \( O(n \times G(n)) \), when representing sets with UNION-FIND [Tarjan, 1975], where \( n \) is the number of variable occurrences in the original terms. The \( \rho \)-machine is similar except that two new factors are involved: the update operation of HAMT and the generation of names and variables. We consider the former one to have \( O(\log(n)) \) complexity, and we implement name and variable creation with state monads [Moggi, 1991] to have constant time. Thus reducing an arbitrary unification problem to the input of the \( \nu \) and \( \delta \) machines becomes \( O(n \times \log(n) \times G(n)) \).
\[
\begin{align*}
e & ::= (\langle t; \Phi \rangle \langle t; \Phi \rangle) \quad \text{multi-equation} \\
& \quad \langle t; \Phi \rangle, e \\
U & ::= e \quad \text{list of multi-equations} \\
& \quad e, U
\end{align*}
\]

Figure 7: Multi-equations

\[
\begin{align*}
p_0; \delta_0; \sigma_0 \vdash U \Rightarrow_p p_0; \delta_0; \sigma_0 \quad \text{[EMPTY]} \\
p_0; \delta_0; \sigma_0 \vdash e \Rightarrow_s p_0; \delta_0; \sigma_0
\end{align*}
\]

\[
\begin{align*}
p_0; \delta_0; \sigma_0 \vdash (\langle a_1; \Phi_1 \rangle \langle a_2; \Phi_2 \rangle) \Rightarrow_p p_1; \delta_1; \sigma_0 \quad \text{[N-N]} \\
p_0; \delta_0; \sigma_0 \vdash (\langle a_1; \Phi_1 \rangle \langle X_2; \Phi_2 \rangle) \Rightarrow_p p_1; \delta_1; \sigma_0 \quad \text{[N-V]} \\
p_0; \delta_0; \sigma_0 \vdash (\langle X_1; \Phi_1 \rangle \langle X_2; \Phi_2 \rangle) \Rightarrow_s p_1; \delta_1; \sigma_0 \\
p_0; \delta_0; \sigma_0 \vdash (\langle l_1; \Phi_1 \rangle \langle l_2; \Phi_2 \rangle) \Rightarrow_s p_1; \delta_1; \sigma_0 \\
p_0; \delta_0; \sigma_0 \vdash (\langle l_1 r_1; \Phi_1 \rangle \langle l_2 r_2; \Phi_2 \rangle) \Rightarrow_s p_1; \delta_1; \sigma_0
\end{align*}
\]

\[
\begin{align*}
\Phi'_1 = (\text{ext } \Phi_1 a_1) \quad \Phi'_2 = (\text{ext } \Phi_2 a_2) \\
p_0; \delta_0; \sigma_0 \vdash (\langle t_1; \Phi_1 \rangle \langle t_2; \Phi_2 \rangle) \Rightarrow_s p_1; \delta_1; \sigma_1 \\
p_0; \delta_0; \sigma_0 \vdash (\langle \lambda a_1 t_1; \Phi_1 \rangle \langle \lambda a_2 t_2; \Phi_2 \rangle) \Rightarrow_s p_1; \delta_1; \sigma_1
\end{align*}
\]

\[
\begin{align*}
X_t = \text{(new-var)} \\
p_0; \delta_0; \sigma_0 \vdash (\langle X_t \rangle \langle X_t \rangle) \Rightarrow_s p_1; \delta_1; \sigma_1
\end{align*}
\]

\[
\begin{align*}
\Phi'_1 = (\text{ext } \Phi_1 a_1) \quad \Phi'_2 = (\text{ext } \Phi_2 a_2) \\
p_0; \delta_0; \sigma_0 \vdash (\langle \lambda a_1 t_1; \Phi_1 \rangle \langle \lambda a_2 t_2; \Phi_2 \rangle) \Rightarrow_s p_1; \delta_1; \sigma_1
\end{align*}
\]

Figure 8: \(\rho\)-machine

\[
\begin{align*}
\Phi'_1 = (\text{ext } \Phi_1 a_1) \quad \Phi'_2 = (\text{ext } \Phi_2 a_2) \\
p_0; \delta_0; \sigma_0 \vdash \text{Fr } a_1 \quad a_2 = \text{(new-name)} \\
p_0; \delta_0; \sigma_0 \vdash (\langle \lambda a_1 t_1; \Phi_1 \rangle \langle \lambda a_2 t_2; \Phi_2 \rangle) \Rightarrow_s p_1; \delta_1; \sigma_1
\end{align*}
\]

\[
\begin{align*}
\Phi'_1 = (\text{ext } \Phi_1 a_1) \quad \Phi'_2 = (\text{ext } \Phi_2 a_2) \\
p_0; \delta_0; \sigma_0 \vdash \text{Bd } a_1 \quad i \quad \text{Bd } a_2 \quad i \\
p_0; \delta_0; \sigma_0 \vdash (\langle \lambda a_1 t_1; \Phi_1 \rangle \langle \lambda a_2 t_2; \Phi_2 \rangle) \Rightarrow_s p_1; \delta_1; \sigma_1
\end{align*}
\]
Acknowledgment

We thank Christian Urban for the discussion and inspiration and the anonymous reviewers for their comments and suggestions.

References


