Efficiency of a good but not linear nominal unification algorithm

Weixi Ma[1], Jeremy G. Siek[2], David Thrane Christiansen[3], and Daniel P. Friedman[4]

1 Indiana University, mvc@iu.edu
2 Indiana University, jsiek@indiana.edu
3 Galois, Inc., dtc@galois.com
4 Indiana University, dfried@indiana.edu

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 α-equivalent. We preserve names while using special representations of de Bruijn numbers. [The following sentence is not clear enough. –Jeremy] Operations on name handling, i.e., deciding the α-equivalence of two names and inferring a name that α-equals to a given one, are in logarithmic time. To reduce an arbitrary unification problem to such name handlings, we preprocess the unification terms The following phrase needs to be reworded. "the" indicates exactly one. But they had many ideas. Which one are you referring to? Be specific. with the idea of Martelli-Montanari.

Keywords and phrases α-conversion; Binding operations; Efficiency; Unification

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction and background

The rules that identify terms, such as α, β, and η in the λ-calculus [Church, 1941], are critical to building programming languages and formal systems. As users of logic programming languages and theorem provers, we desire such rules to be out-of-the-box in the tool-kit. Two theories have aimed to provide this convenience: higher-order pattern unification of Miller [1989] and nominal unification of Urban et al. [2004]. Higher-order pattern unification, which handles a fragment of $\beta\eta$-rules, is the foundation of Isabelle [Paulson, 1986], λProlog [Nadathur and Miller, 1988], and Twelf [Pfenning and Schürmann, 1999]. Nominal unification, which focuses on the α-rule, has inspired extensions of logic programming languages, like αProlog [Cheney and Urban, 2004] and αKanren [Byrd and Friedman, 2007], as well as theorem provers, like nominal Isabelle [Urban and Tasson, 2005] and αLeanTAP [Near et al., 2008]. Although these two theories can be reduced to eachother [Cheney, 2005, Levy and Villaret, 2012], implementing higher-order pattern unification is more complicated because it has to deal with application and capture-avoiding substitution. On the other hand, implementation of nominal unification, which essentially unifies first-order terms, is more straightforward and easier to formalize. Beyond unification, techniques from the nominal approach, such as swapping and freshness environments, have impacted the areas as diverse as rewriting [Fernández et al., 2004, Fernández and Gabbay, 2005, 2007, Aoto and Kikuchi, 2016], equational theories [Ayala-Rincón et al., 2016], and reasoning about bindings in abstract syntax [Pitts and Gabbay, 2000, Gabbay and Pitts, 2002].

Concerning time complexity, Qian [1996] proved that higher-order pattern unification is decidable in linear time. On the other hand, 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. Also, ideas like applying swappings lazily and composing swappings eagerly and sharing subterms have been explored. Calvès [2010] describes quadratic algorithms that extend Paterson-Wegman and Martelli-Montanari's algorithms with name 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, that is, to decide the α-equivalence of two names, we need to linearly traverse a list whose length grows with respect to the number of binders. One might try to replace these lists with a more efficient lookup structure, such as a hash table, 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 which enables the use of persistent hash tables, in particular, a Hash Array Mapped Trie (HAMT) [Bagwell, 2001]. HAMTs provide efficient lookup and they use sharing to avoid the linear-time costs that would normally be associated with duplicating a hash table.

We organize this paper as follows. In section 2, we show 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, 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 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 α-equivalence. A few programming languages [Norell, 2007] 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 variable-theories [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 its convenience when implementing α-equivalence, programs written with de Bruijn numbers are notoriously obfuscated for humans to read and understand. What's worse, as pointed out by Berghofer and Urban [Berghofer and Urban, 2007], translating pencil-and-paper style proofs to versions using de Bruijn numbers is surprisingly involved: such translation may alter the structures of proofs. Consequently, reproducing proofs with explicit names from 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 the α-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 α-equivalence, one frequently needs the mapping *from numbers to names* to decide what name to assign to a unification variable.

We propose to represent de Bruijn numbers by *static closures*, hereafter referred to as *closures*. Closures preserve the mappings of both directions: names to numbers and numbers to names.

▶ **Definition 1.** A *closure* is an ordered pair $\langle t; \Phi \rangle$ of a term t, defined in figure 1, and a scope Φ, where the scope is an ordered list of names for the binders in the enclosing context. Hereafter, we refer to a name as an *atom*.

When the term of a closure is an atom, the closure itself represents a de Bruijn number. For example, consider the term $\lambda x.\lambda y.x$. The de Bruijn number of the atom x is 1 and the closure-representation of this number is $\langle x; (y\,x) \rangle$. We can retrieve the number-representation by finding the position of the first appearance of the atom in the scope. In this case, the position of x in the list $(y\,x)$ is 1, which is its de Bruijn number. Similarly, the de Bruijn number of y is 0.

A scope, as a list, supports three operations: `ext-scope`, which extends the scope by adding an atom to the front of the list; `idx→atom`, which returns the atom of a given index starting from the front of the list; and `atom→idx`, which returns the location of the first appearance of a given atom counting from the front of the list. As we are building the list in reversed order, if repeated atoms appear, the first appearance in a list shadows the others.

Figure 1 Terms

$$\begin{array}{llll} t, l, r & ::= & a & \text{atom} \\ & \mid & \lambda a.t & \text{abstractions} \\ & \mid & (l\,r) & \text{applications} \end{array}$$

Figure 2 Free and bound

$$\frac{a \notin \Phi}{\Phi \vdash \texttt{Fr}\ a}\ \textsc{Free}$$

$$\frac{(\texttt{name→idx}\ \Phi\ a) = i \quad (\texttt{idx→name}\ \Phi\ i) = a}{\Phi \vdash \texttt{Bd}\ a\ i}\ \textsc{Bound}$$

Figure 3 $\approx$-rules

$$\frac{\Phi_1 \vdash \texttt{Fr}\ a_1 \quad \Phi_2 \vdash \texttt{Fr}\ a_2 \quad a_1 = a_2}{\langle a_1; \Phi_1 \rangle \approx \langle a_2; \Phi_2 \rangle}\ \textsc{Same-Free}$$

$$\frac{\Phi_1 \vdash \texttt{Bd}\ a_1\ i_1 \quad \Phi_2 \vdash \texttt{Bd}\ a_2\ i_2 \quad i_1 = i_2}{\langle a_1; \Phi_1 \rangle \approx \langle a_2; \Phi_2 \rangle}\ \textsc{Same-Bound}$$

Now in figure 2, we can talk about free and bound variables "constructively," with de Bruijn numbers serving as evidence that variables are well-scoped. When an atom, a, does not appear in the scope, Φ, we say, "a is free with respect to Φ," written as $\Phi \vdash \texttt{Fr}\,a$; when a's first appearance in Φ is the position i, we say, "a is bound at i with respect to Φ," written as $\Phi \vdash \texttt{Bd}\,a\,i$. The BOUND rule has two premises to be algorithmic in both directions, that is, given an atom we can find its index and given an index we can find its atom, if no shadowing happen. Figure 3 defines the rules to decide whether two atoms are α-equivalent w.r.t theirs scopes, written as $\langle a_1; \Phi_1 \rangle \approx \langle a_2; \Phi_2 \rangle$.

3 Unification

In figure 4, we introduce unification variables, shortened as var. First, 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 atoms and variables. A unifier consists of two parts: σ and δ.

Figure 4 Unification terms and problems

X, Y			variables
xs, ys	$::=$	ϵ	list of variables
	$\mid$	X, xs	
t, l, r	$::=$	a	atoms
	$\mid$	$\lambda a.t$	abstractions
	$\mid$	$(l\, r)$	applications
	$\mid$	X	
e_ν	$::=$	$(a, \Phi) = (a, \Phi)$	ν-equations
	$\mid$	$(X, \Phi) = (a, \Phi)$	
p_ν	$::=$	ϵ	ν-problems
	$\mid$	e_ν, p_ν	
e_δ	$::=$	$(X, \Phi) = (X, \Phi)$	δ-equations
p_δ	$::=$	ϵ	δ-problems
	$\mid$	e_δ, p_δ	

▶ **Definition 2.** A substitution σ is a partial finite function from variables, X_i, to terms, t_i. For readability, we write σ as a set, $\{X_1/t_1, ..., X_j/t_j\}$ and we write $\{X/t\} \cup \sigma$ for extending σ with X/t. For now, we assume the co-domain of a σ only includes atoms.

▶ **Definition 3.** A closure equation is a pair of two closures that are α-equivalent. Δ is a set of closure-equations. We write Δ as $\{(\langle t_1; \Phi_1\rangle\ \langle t_1'; \Phi_1'\rangle), ..., (\langle t_i; \Phi_1\rangle\ \langle t_i'; \Phi_1'\rangle)\}$ and we write $\{(\langle t; \Phi\rangle\ \langle t'; \Phi'\rangle)\} \cup \Delta$ for extending Δ with $(\langle t; \Phi\rangle\ \langle t'; \Phi'\rangle)$. We write δ as a special form of Δ: for each equation in δ, the term of both sides are variables. Given a variable X, we write $\delta(X)$ as retrieving the list of closure-equations that X is on one side of.

The simplified problem is about solving three kinds of closure-equations: atom-atom, atom-var, and var-var. [talk about the equations and problems defined in Figure 4] We refer to a atom-atom or atom-var problem as a *problem*$_\nu$ and refer to a var-var problem as a *problem*$_\delta$. Given a set of problem$_\nu$ and a set of problem$_\delta$, we first run the ν-machine, defined in figure 5, on the set of problem$_\nu$ to generate a substitution.

The δ-machine in Figure 6 computes the final unifier on three inputs: the substitution resulted from the ν-machine, δ, 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.

Figure 5 ν-machine

$$\boxed{\sigma \vdash p_\nu \Rightarrow_\nu \sigma}$$

$$\frac{}{\sigma_0 \vdash \epsilon \Rightarrow_\nu \sigma_0}\ \textsc{Empty}$$

$$\frac{\sigma_0 \vdash p \Rightarrow_\nu \sigma_1 \quad \langle a_1; \Phi_1\rangle \approx \langle a_2; \Phi_2\rangle}{\sigma_0 \vdash \langle a_1; \Phi_1\rangle = \langle a_2; \Phi_2\rangle,\, p \Rightarrow_\nu \sigma_1}\ \textsc{Name-Name}$$

$$\frac{\begin{array}{c}\{X_2/a_2\} \cup \sigma_0 \vdash p \Rightarrow_\nu \sigma_1 \\ \langle a_1; \Phi_1\rangle \approx \langle a_2; \Phi_2\rangle\end{array}}{\sigma_0 \vdash \langle a_1; \Phi_1\rangle = \langle X_2; \Phi_2\rangle,\, p \Rightarrow_\nu \sigma_1}\ \textsc{Name-Meta}$$

▶ **Lemma 4.** *For all finite input, the ν-machine and the δ-machine terminates; for all input, the ν-machine and the δ-machine succeeds with the mgu if and only if there exists one.*

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

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 atoms, can we re-shape the two terms to create a proper input to the two machines?

Finding the common structure, obviously, is merely a first-order unification problem. The ρ-machine, defined in figure 7, adapts the idea of Martelli-Montanari and reduces an arbitrary nominal unification problem to a set of problem$_\nu$, a set of problem$_\delta$, and a substitution. Here we need to extend the definition of substitution: it is now a partial finite function from variables to terms. Also, in the META-APP and META-ABS rules, we need to create new atoms and new variables.

Figure 6 δ-machine and the pull operation

$$\boxed{\begin{array}{l}\sigma; p_\delta \vdash xs \Rightarrow_\delta \sigma; p_\delta \\ \sigma; xs \vdash p_\delta \Rightarrow_{\text{pull}} \sigma; xs\end{array}}$$

$$\frac{}{\sigma; \delta \vdash \epsilon \Rightarrow_\delta \sigma; \delta}\ \textsc{Empty-Q}$$

$$\frac{}{\sigma; \epsilon \vdash xs \Rightarrow_\delta \sigma; \epsilon}\ \textsc{Empty-D}$$

$$\frac{\begin{array}{l}\sigma_0'; \delta_0 \setminus \delta_0(X) \vdash xs_1 \Rightarrow_\delta \sigma_1; \delta_1 \\ \sigma_0; xs_0 \vdash \delta_0(X) \Rightarrow_{\text{pull}} \sigma_0'; xs_1\end{array}}{\sigma_0; \delta_0 \vdash X, xs_0 \Rightarrow_\delta \sigma_1; \delta_1}\ \textsc{Step}$$

$$\frac{}{\sigma; q \vdash \emptyset \Rightarrow_{\text{pull}} \sigma; q}\ \textsc{Empty}$$

$$\frac{\begin{array}{l}\sigma_0; xs_0 \vdash p \Rightarrow_{\text{pull}} \sigma_1; xs_1 \\ \langle a_1; \Phi_1\rangle \approx \langle a_2; \Phi_2\rangle \quad \sigma_0(Y_1) = a_1 \quad \sigma_0(Y_2) = a_2\end{array}}{\sigma_0; xs_0 \vdash \langle Y_1; \Phi_1\rangle = \langle Y_2; \Phi_2\rangle, p \Rightarrow_{\text{pull}} \sigma_1; xs_1}\ \textsc{Name-Name}$$

$$\frac{\begin{array}{l}\{Y_2/a_2\} \cup \sigma_0; (Y_2, xs_0) \vdash p \Rightarrow_{\text{pull}} \sigma_1; xs_1 \\ \langle a_1; \Phi_1\rangle \approx \langle a_2; \Phi_2\rangle \quad \sigma_0(Y_1) = a_1 \quad Y_2 \notin dom(\sigma_0)\end{array}}{\sigma_0; xs_0 \vdash \langle Y_1; \Phi_1\rangle = \langle Y_2; \Phi_2\rangle, p \Rightarrow_{\text{pull}} \sigma_1; xs_1}\ \textsc{Na}$$

Figure 7 ρ-machine

$$\boxed{\begin{array}{l}(problem^*_\nu, problem^*_\delta, \sigma) \vdash multieqn^* \Rightarrow_\rho (problem^*_\nu, problem^*_\delta, \sigma)\\ (problem^*_\nu, problem^*_\delta, \sigma) \vdash multieqn \Rightarrow_{\text{step}} (problem^*_\nu, problem^*_\delta, \sigma)\end{array}}$$

$$\frac{}{(p_0, \delta_0, \sigma_0) \vdash \emptyset \Rightarrow_\rho (p_0, \delta_0, \sigma_0)}\ \text{EMPTY}$$

$$\frac{\begin{array}{c}(p'_0, \delta'_0, \sigma'_0) \vdash U^* \Rightarrow_\rho (p_1, \delta_1, \sigma_1)\\ (p_0, \delta_0, \sigma_0) \vdash U \Rightarrow_{\text{step}} (p'_0, \delta'_0, \sigma'_0)\end{array}}{(p_0, \delta_0, \sigma_0) \vdash (U, U^*) \Rightarrow_\rho (p_1, \delta_1, \sigma_1)}\ \text{STEP}$$

$$\frac{p_1 = (\langle a_1; \Phi_1\rangle\, \langle a_2; \Phi_2\rangle) \cup p_0}{(p_0, \delta_0, \sigma_0) \vdash (\langle a_1; \Phi_1\rangle\, \langle a_2; \Phi_2\rangle) \Rightarrow_{\text{step}} (p_1, \delta_0, \sigma_0)}\ \text{NAME-NAME}$$

$$\frac{p_1 = (\langle a_1; \Phi_1\rangle\, \langle X_2; \Phi_2\rangle) \cup p_0}{(p_0, \delta_0, \sigma_0) \vdash (\langle a_1; \Phi_1\rangle\, \langle X_2; \Phi_2\rangle) \Rightarrow_{\text{step}} (p_1, \delta_0, \sigma_0)}\ \text{NAME-META}$$

$$\frac{\delta_1 = (\langle X_1; \Phi_1\rangle\, \langle X_2; \Phi_2\rangle) \cup \delta_0}{(p_0, \delta_0, \sigma_0) \vdash (\langle a_1; \Phi_1\rangle\, \langle X_2; \Phi_2\rangle) \Rightarrow_{\text{step}} (p_0, \delta_1, \sigma_0)}\ \text{META-META}$$

$$\frac{\begin{array}{c}(p_0, \delta_0, \sigma_0) \vdash (\langle l_1; \Phi_1\rangle\, \langle l_2; \Phi_2\rangle) \Rightarrow_{\text{step}} (p'_0, \delta'_0, \sigma'_0)\\ (p'_0, \delta'_0, \sigma'_0) \vdash (\langle r_1; \Phi_1\rangle\, \langle r_2; \Phi_2\rangle) \Rightarrow_{\text{step}} (p_1, \delta_1, \sigma_1)\end{array}}{(p_0, \delta_0, \sigma_0) \vdash (\langle (l_1\, r_1); \Phi_1\rangle\, \langle (l_2\, r_2); \Phi_2\rangle) \Rightarrow_{\text{step}} (p_1, \delta_1, \sigma_1)}\ \text{APP-APP}$$

$$\frac{\begin{array}{l}(p_0, \delta_0, \sigma_0) \vdash (\langle t_1; \Phi'_1\rangle\, \langle t_2; \Phi'_2\rangle) \Rightarrow_{\text{step}} (p_1, \delta_1, \sigma_1)\\ \Phi'_1 = (\texttt{ext-scope}\, \Phi_1\, a_1) \quad \Phi'_2 = (\texttt{ext-scope}\, \Phi_2\, a_2)\end{array}}{(p_0, \delta_0, \sigma_0) \vdash (\langle \lambda\, a_1.t_1; \Phi_1\rangle\, \langle \lambda\, a_2.t_2; \Phi_2\rangle) \Rightarrow_{\text{step}} (p_1, \delta_1, \sigma_1)}\ \text{ABS-ABS}$$

$$\frac{\begin{array}{l}(p_0, \delta_0, \{X_1/(X_l, X_r)\} \cup \sigma'_0) \vdash (\langle X_l; \Phi_1\rangle\, \langle l_2; \Phi_2\rangle) \Rightarrow_{\text{step}} (p'_0, \delta'_0, \sigma'_0)\\ (p'_0, \delta'_0, \sigma'_0) \vdash (\langle X_r; \Phi_1\rangle\, \langle r_2; \Phi_2\rangle) \Rightarrow_{\text{step}} (p_1, \delta_1, \sigma_1)\end{array}}{(p_0, \delta_0, \sigma_0) \vdash (\langle X_1; \Phi_1\rangle\, \langle (l_2\, r_2); \Phi_2\rangle) \Rightarrow_{\text{step}} (p_1, \delta_1, \sigma_1)}\ \text{META-APP}$$

$$\frac{\begin{array}{l}(p_0, \delta_0, X_1/\lambda\, a_1.X_t \cup \sigma'_0) \vdash (\langle X_t; \Phi'_1\rangle\, \langle t_2; \Phi'_2\rangle) \Rightarrow_{\text{step}} (p_1, \delta_1, \sigma_1)\\ \Phi'_1 = (\texttt{ext-scope}\, \Phi_1\, a_1) \quad \Phi'_2 = (\texttt{ext-scope}\, \Phi_2\, a_2)\end{array}}{(p_0, \delta_0, \sigma_0) \vdash (\langle X_1; \Phi_1\rangle\, \langle \lambda\, a_2.t_2; \Phi_2\rangle) \Rightarrow_{\text{step}} (p_1, \delta_1, \sigma_1)}\ \text{META-ABS}$$

4 A note on time complexity

In the previous sections, we represent scopes by lists for simplicity, but lists are inefficient for variable lookup. To have better time complexity, we represent a scope with a counter and two immutable hashtables, also known as hamt [Bagwell, 2001]. 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. An immutable hashtable, in practice, has constant time for update and lookup, thought the worst case scenario could be $O(log(n))$. Thus, `ext-scope`, `idx→atom`, and `atom→idx` are all logarithmic time. In addition, using immutable structures avoids copying the entire data-structure when branching, in particular, during the APP-APP rule of the ρ-machine.

We implement δ with a hashtable that maps from a variable to the list that contains its closure-equations. Doing so doubles the space consumption, i.e, the equation $\langle X; \Phi_1\rangle \approx \langle Y; \Phi_2\rangle$ exists in both X's entry and Y's entry, but improves the time efficiency.

Given the above optimizations, the ν-machine and the δ-machine are both $O(n \times log(n))$ at the worst case, where n is the number of atom occurrences 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 ρ-machine is similar except that two new factors are involved: the update operation of an hamt and the generation of atoms and variables. We consider the former one to have $O(log(n))$ complexity, and we implement atom and variable creation with state monads in constant time. Thus reducing an arbitrary unification problem to the input of the ν and δ machines is $O(n \times log(n) \times G(n))$.

References

Takahito Aoto and Kentaro Kikuchi. Nominal confluence tool. In *Automated Reasoning*, Lecture Notes in Computer Science, pages 173–182. Springer, Cham, June 2016. ISBN 978-3-319-40228-4 978-3-319-40229-1.

Mauricio Ayala-Rincón, Maribel Fernández, and Daniele Nantes-Sobrinho. Nominal narrowing. In Delia Kesner and Brigitte Pientka, editors, *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*, volume 52 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:17, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-010-1.

Brian Aydemir, Aaron Bohannon, and Stephanie Weirich. Nominal Reasoning Techniques in Coq. In *International Workshop on Logical Frameworks and Meta-Languages:Theory and Practice (LFMTP)*, Seattle, WA, USA, August 2006.

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 3–15, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9.

Phil Bagwell. Ideal Hash Trees. November 2001.

Stefan Berghofer and Christian Urban. A Head-to-Head Comparison of de Bruijn Indices and Names. *Electronic Notes in Theoretical Computer Science*, 174(5):53–67, June 2007. ISSN 1571-0661.

William E. Byrd and Daniel P. Friedman. *alphaKanren: A fresh name in nominal logic programming.* Scheme and Functional Programming Workshop, 2007.

Christophe Calvès. *Complexity and Implementation of Nominal Algorithms.* PhD thesis. King's College of London, 2010.

Arthur Charguéraud. The locally nameless representation. *J Autom Reasoning*, 49(3):363–408, October 2012. ISSN 0168-7433, 1573-0670.

James Cheney. Relating nominal and higher-order pattern unification. In *Proceedings of UNIF 2005*, pages 104–119, 2005.

James Cheney and Christian Urban. αProlog: A logic programming language with names, binding and α-equivalence. In *Logic Programming*, Lecture Notes in Computer Science, pages 269–283. Springer, Berlin, Heidelberg, September 2004. ISBN 978-3-540-22671-0 978-3-540-27775-0.

Alonzo Church. *The Calculi of Lambda Conversion.* 1941.

Nicolaas G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, January 1972. ISSN 1385-7258.

Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Higher Order Unification via Explicit Substitutions. *Information and Computation*, 157(1):183–235, February 2000. ISSN 0890-5401.

Maribel Fernández and Murdoch J. Gabbay. Nominal rewriting with name generation: abstraction vs. locality. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, pages 47–58, New York, NY, USA, 2005. ACM. ISBN 978-1-59593-090-3.

Maribel Fernández and Murdoch J. Gabbay. Nominal rewriting. *Information and Computation*, 205(6):917–965, June 2007. ISSN 0890-5401.

Maribel Fernández, Murdoch J. Gabbay, and Ian Mackie. Nominal rewriting systems. In *Proceedings of the 6th ACM SIGPLAN International Conference on Principles and Prac-*

tice of Declarative Programming, PPDP '04, pages 108–119, New York, NY, USA, 2004. ACM. ISBN 978-1-58113-819-1.

Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Form Aspects Comput*, 13(3-5):341–363, July 2002. ISSN 0934-5043, 1433-299X.

Jordi Levy and Mateu Villaret. An efficient nominal unification algorithm, 2010.

Jordi Levy and Mateu Villaret. Nominal unification from a higher-order perspective. *ACM Transactions on Computational Logic*, 13(2):1–31, April 2012. ISSN 15293785. arXiv: 1005.3731.

Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, April 1982. ISSN 0164-0925.

Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In *Extensions of Logic Programming*, Lecture Notes in Computer Science, pages 253–281. Springer, Berlin, Heidelberg, December 1989. ISBN 978-3-540-53590-4 978-3-540-46879-0.

Gopalan Nadathur and Dale Miller. An Overview of Lambda-Prolog. pages 810–827, June 1988.

Joseph P. Near, William E. Byrd, and Daniel P. Friedman. αleanTAP: A declarative theorem prover for first-order classical logic. In *Logic Programming*, Lecture Notes in Computer Science, pages 238–252. Springer, Berlin, Heidelberg, December 2008. ISBN 978-3-540-89981-5 978-3-540-89982-2.

Ulf Norell. *Towards A Practical Programming Language Based on Dependent Type Theory*. 2007.

M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, April 1978. ISSN 0022-0000.

Lawrence C. Paulson. Natural deduction as higher-order resolution. *The Journal of Logic Programming*, 3(3):237–258, October 1986. ISSN 0743-1066.

Frank Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *Automated Deduction — CADE-16*, Lecture Notes in Computer Science, pages 202–206. Springer, Berlin, Heidelberg, July 1999. ISBN 978-3-540-66222-8 978-3-540-48660-2.

Andrew M. Pitts and Murdoch J. Gabbay. A metalanguage for programming with bound names modulo renaming. In *Mathematics of Program Construction*, Lecture Notes in Computer Science, pages 230–255. Springer, Berlin, Heidelberg, July 2000. ISBN 978-3-540-67727-7 978-3-540-45025-2.

Z. Qian. Unification of Higher-order Patterns in Linear Time and Space. *J Logic Computation*, 6(3):315–341, June 1996. ISSN 0955-792X.

Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22 (2):215–225, April 1975. ISSN 0004-5411.

Christian Urban and Christine Tasson. Nominal techniques in Isabelle/HOL. In *Automated Deduction – CADE-20*, Lecture Notes in Computer Science, pages 38–53. Springer, Berlin, Heidelberg, July 2005. ISBN 978-3-540-28005-7 978-3-540-31864-4.

Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1-3):473–497, September 2004. ISSN 03043975.