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, Bloomington, Indiana, U.S.A.
mvc@iu.edu
[2] Indiana University, Bloomington, Indiana, U.S.A.
jsiek@indiana.edu
[3] Galois, Inc., Portland, Oregon, U.S.A.
dtc@galois.com
[4] Indiana University, Bloomington, Indiana, U.S.A.
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 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 α, β, and η in the λ-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], λProlog [Nadathur et al., 1988], and Twelf [Pfenning and Schürmann, 1999], handles a fragment of the $\beta\eta$-rules. Nominal unification, the unification modulo the α-rule, has inspired extensions of logic programming languages such as αProlog [Cheney and Urban, 2004] and αKanren [Byrd and Friedman, 2007], as well as theorem provers such as Nominal Isabelle [Urban and Tasson, 2005] and αLeanTAP [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 β-reduction and capture-avoiding substitution. An implementation of nominal unification, in which unification does not involve explicit β-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. Calvès [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 α-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,[1] 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 α-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.

Figure 1: Terms

$$
\begin{array}{llll}
t, l, r & ::= & a & \text{name} \\
 & | & \lambda a.t & \text{abstraction} \\
 & | & (l\, r) & \text{combination}
\end{array}
$$

Figure 2: Free and bound

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

$$\frac{(\texttt{name}{\to}\texttt{idx}\ \Phi\ a) = i \qquad (\texttt{idx}{\to}\texttt{name}\ \Phi\ i) = a}{\Phi \vdash \texttt{Bd}\ a\ i}\ [\textsc{Bound}]$$

Figure 3: $\approx$-rules

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

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

Despite the convenience when implementing α-equivalence, programs written with de Bruijn numbers are notoriously difficult for humans

[1] https://github.com/mvcccccc/UNIF2018

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 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 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 Φ, 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 Φ.*

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 the name in the scope. In this case, the position of a in the scope $(b\, a)$ is 1, its de Bruijn number. The de Bruijn number of b would have been 0 if the closure had been $\langle b; (b\, a) \rangle$ We define three operations on scopes: `ext` extends the scope by `cons`ing a name to the front of the scope; `idx`→`name` yields the name of a given index; `name`→`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 4: Unification terms and problems

X			vars
a, b			names
xs	$::=$	ϵ	list of vars
	$\mid$	X, xs	
t, l, r	$::=$	a	
	$\mid$	$\lambda a.t$	abstractions
	$\mid$	$(l\, r)$	combinations
	$\mid$	X	
e_ν	$::=$	$\langle a; \Phi \rangle = \langle a; \Phi \rangle$	ν-equation
	$\mid$	$\langle a; \Phi \rangle = \langle X; \Phi \rangle$	
p_ν	$::=$	ϵ	ν-problems
	$\mid$	e_ν, p_ν	
e_δ	$::=$	$\langle X; \Phi \rangle = \langle X; \Phi \rangle$	δ-equation
p_δ	$::=$	ϵ	δ-problems
	$\mid$	e_δ, p_δ	

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, 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 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, $(a\, a) \vdash \texttt{Bd}\, a\, 1$, does **not** hold. Figure 3 defines the rules that decide α-equivalent of two names w.r.t. their scopes, written as $\langle a; \Phi \rangle \approx \langle a; \Phi \rangle$.

3 Unification

In Figure 4, we introduce unification variables, abbreviated as *var*s. 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: σ and δ.

A *substitution*, σ, is a partial finite function from unification 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 the simplified problems, we restrict t to a name.

A *closure equation* is a pair of two closures that are α-equivalent. Δ stands for 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 $(\langle t; \Phi\rangle\ \langle t'; \Phi'\rangle)$. δ is a special form of Δ: for each equation in δ, 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_ν and refer to a V-V equation as an e_δ. Given two lists of these closure equations, p_ν and p_δ, we first run the ν-machine, defined in Figure 5, on p_ν to generate a substitution. The δ-machine, defined in Figure 6, then computes the final unifier on three inputs: the substitution resulting 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.

Lemma 3.1. *For all finite inputs, the ν-machine and the δ-machine terminate; for all finite inputs, the ν-machine and the δ-machine succeed with the most general unifier if and only if one exists.*

Proof. By structural induction on the transitions of the machines. $\square$

Figure 5: ν-machine

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

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

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

$$\frac{\langle a_1; \Phi_1\rangle \approx \langle a_2; \Phi_2\rangle \quad \{X_2/a_2\} \cup \sigma_0 \vdash p \Rightarrow_\nu \sigma_1}{\sigma_0 \vdash \langle a_1; \Phi_1\rangle = \langle X_2; \Phi_2\rangle,\ p \Rightarrow_\nu \sigma_1}\ [\text{N-V}]$$

Figure 6: δ-machine and the pull operation

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

$$\frac{}{\sigma; \delta \vdash \epsilon \Rightarrow_\delta \sigma; \delta}\ [\textsc{Empty-Xs}]$$

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

$$\frac{\sigma_0; xs_0 \vdash \delta_0(X) \Rightarrow_{\text{pull}} \sigma'_0; xs_1 \quad \sigma'_0; \delta_0 \setminus \delta_0(X) \vdash xs_1 \Rightarrow_\delta \sigma_1; \delta_1}{\sigma_0; \delta_0 \vdash X, xs_0 \Rightarrow_\delta \sigma_1; \delta_1}\ [\textsc{Pull}]$$

$$\frac{}{\sigma; xs \vdash \epsilon \Rightarrow_{\text{pull}} \sigma; xs}\ [\textsc{Empty}]$$

$$\frac{\langle a_1; \Phi_1\rangle \approx \langle a_2; \Phi_2\rangle \quad \sigma_0(X_1) = a_1 \quad \sigma_0(X_2) = a_2 \quad \sigma_0; xs_0 \vdash p \Rightarrow_{\text{pull}} \sigma_1; xs_1}{\sigma_0; xs_0 \vdash \langle X_1; \Phi_1\rangle = \langle X_2; \Phi_2\rangle, p \Rightarrow_{\text{pull}} \sigma_1; xs_1}\ [\text{N-N}]$$

$$\frac{\langle a_1; \Phi_1\rangle \approx \langle a_2; \Phi_2\rangle \quad \sigma_0(X_1) = a_1 \quad X_2 \notin dom(\sigma_0) \quad \{X_2/a_2\} \cup \sigma_0; (X_2, xs_0) \vdash p \Rightarrow_{\text{pull}} \sigma_1; xs_1}{\sigma_0; xs_0 \vdash \langle X_1; \Phi_1\rangle = \langle X_2; \Phi_2\rangle, p \Rightarrow_{\text{pull}} \sigma_1; xs_1}\ [\text{N-V}]$$

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 $(a\,X)$ and $(Y\,(b\,c))$, the common part is the term $(Y\,X)$, and the frontier is the substitution $\{Y/a,\ X/(b\,c)\}$. 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 ρ-machine, defined in Figure 8, reduces an arbitrary nominal unification problem to p_ν, p_δ, and a substitution where the codomain is unrestricted. Each $\Rightarrow_s$ 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 ρ-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, `new-name` and `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.

Conjecture 3.1. *Given a unification problem, we run the ρ-machine, the ν-machine, and the δ-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 δ-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, `ext`, `idx`$\rightarrow$`name`, and `name`$\rightarrow$`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 ρ-machine. Also, we implement δ 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 ν-machine and the δ-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 ρ-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 ν and δ machines becomes $O(n \times log(n) \times G(n))$.

Figure 7: Multi-equations

e	$::=$	$(\langle t; \Phi\rangle\ \langle t; \Phi\rangle)$	multi-equation
	$\mid$	$\langle t; \Phi\rangle, e$	
U	$::=$	ϵ	list of multi-equations
	$\mid$	e, U	

Figure 8: ρ-machine

$$\boxed{\begin{array}{l} p_\nu; p_\delta; \sigma \vdash U \Rightarrow_\rho p_\nu; p_\delta; \sigma \\ p_\nu; p_\delta; \sigma \vdash e \Rightarrow_s p_\nu; p_\delta; \sigma \end{array}}$$

$$\frac{}{p_0; \delta_0; \sigma_0 \vdash \epsilon \Rightarrow_\rho p_0; \delta_0; \sigma_0}\ [\textsc{Empty}]$$

$$\frac{\begin{array}{c} p_0; \delta_0; \sigma_0 \vdash e \Rightarrow_s p_0'; \delta_0'; \sigma_0' \\ p_0'; \delta_0'; \sigma_0' \vdash U \Rightarrow_\rho p_1; \delta_1; \sigma_1 \end{array}}{p_0; \delta_0; \sigma_0 \vdash (e, U) \Rightarrow_\rho p_1; \delta_1; \sigma_1}\ [\textsc{Step}]$$

$$\frac{p_1 = (\langle a_1; \Phi_1\rangle\ \langle a_2; \Phi_2\rangle) \cup p_0}{p_0; \delta_0; \sigma_0 \vdash (\langle a_1; \Phi_1\rangle\ \langle a_2; \Phi_2\rangle) \Rightarrow_s p_1; \delta_0; \sigma_0}\ [\text{N-N}]$$

$$\frac{p_1 = (\langle a_1; \Phi_1\rangle\ \langle X_2; \Phi_2\rangle) \cup p_0}{p_0; \delta_0; \sigma_0 \vdash (\langle a_1; \Phi_1\rangle\ \langle X_2; \Phi_2\rangle) \Rightarrow_s p_1; \delta_0; \sigma_0}\ [\text{N-V}]$$

$$\frac{\delta_1 = (\langle X_1; \Phi_1\rangle\ \langle X_2; \Phi_2\rangle) \cup \delta_0}{p_0; \delta_0; \sigma_0 \vdash (\langle X_1; \Phi_1\rangle\ \langle X_2; \Phi_2\rangle) \Rightarrow_s p_0; \delta_1; \sigma_0}\ [\text{V-V}]$$

$$\frac{\begin{array}{c} p_0; \delta_0; \sigma_0 \vdash (\langle l_1; \Phi_1\rangle\ \langle l_2; \Phi_2\rangle) \Rightarrow_s p_0'; \delta_0'; \sigma_0' \\ p_0'; \delta_0'; \sigma_0' \vdash (\langle r_1; \Phi_1\rangle\ \langle r_2; \Phi_2\rangle) \Rightarrow_s p_1; \delta_1; \sigma_1 \end{array}}{p_0; \delta_0; \sigma_0 \vdash (\langle (l_1\, r_1); \Phi_1\rangle\ \langle (l_2\, r_2); \Phi_2\rangle) \Rightarrow_s p_1; \delta_1; \sigma_1}\ [\text{C-C}]$$

$$\frac{\begin{array}{c} \Phi_1' = (\texttt{ext}\, \Phi_1\, a_1) \quad \Phi_2' = (\texttt{ext}\, \Phi_2\, a_2) \\ p_0; \delta_0; \sigma_0 \vdash (\langle t_1; \Phi_1'\rangle\ \langle t_2; \Phi_2'\rangle) \Rightarrow_s p_1; \delta_1; \sigma_1 \end{array}}{p_0; \delta_0; \sigma_0 \vdash (\langle \lambda\, a_1.t_1; \Phi_1\rangle\ \langle \lambda\, a_2.t_2; \Phi_2\rangle) \Rightarrow_s p_1; \delta_1; \sigma_1}\ [\text{A-A}]$$

$$\frac{\begin{array}{l} X_l = (\texttt{new-var}) \quad X_r = (\texttt{new-var}) \\ p_0; \delta_0; \{X_1/(X_l, X_r)\} \cup \sigma_0 \vdash (\langle X_l; \Phi_1\rangle\ \langle l_2; \Phi_2\rangle) \Rightarrow_s p_0'; \delta_0'; \sigma_0' \\ p_0'; \delta_0'; \sigma_0' \vdash (\langle X_r; \Phi_1\rangle\ \langle r_2; \Phi_2\rangle) \Rightarrow_s p_1; \delta_1; \sigma_1 \end{array}}{p_0; \delta_0; \sigma_0 \vdash (\langle X_1; \Phi_1\rangle\ \langle (l_2\, r_2); \Phi_2\rangle) \Rightarrow_s p_1; \delta_1; \sigma_1}\ [\text{V-C}]$$

$$\frac{\begin{array}{l} \Phi_1' = (\texttt{ext}\, \Phi_1\, a_1) \quad \Phi_2' = (\texttt{ext}\, \Phi_2\, a_2) \quad \Phi_1 \vdash \texttt{Fr}\ a_1 \quad a_2 = (\texttt{new-name}) \quad X_t = (\texttt{new-var}) \\ p_0; \delta_0; \{X_1/\lambda\, a_1.X_t\} \cup \sigma_0' \vdash (\langle X_t; \Phi_1'\rangle\ \langle t_2; \Phi_2'\rangle) \Rightarrow_s p_1; \delta_1; \sigma_1 \end{array}}{p_0; \delta_0; \sigma_0 \vdash (\langle X_1; \Phi_1\rangle\ \langle \lambda\, a_2.t_2; \Phi_2\rangle) \Rightarrow_s p_1; \delta_1; \sigma_1}\ [\text{V-A}]$$

$$\frac{\begin{array}{l} \Phi_1' = (\texttt{ext}\, \Phi_1\, a_1) \quad \Phi_2' = (\texttt{ext}\, \Phi_2\, a_2) \quad \Phi_1 \vdash \texttt{Bd}\ a_1\ i \quad \Phi_2 \vdash \texttt{Bd}\ a_2\ i \quad X_t = (\texttt{new-var}) \\ p_0; \delta_0; \{X_1/\lambda\, a_1.X_t\} \cup \sigma_0' \vdash (\langle X_t; \Phi_1'\rangle\ \langle t_2; \Phi_2'\rangle) \Rightarrow_s p_1; \delta_1; \sigma_1 \end{array}}{p_0; \delta_0; \sigma_0 \vdash (\langle X_1; \Phi_1\rangle\ \langle \lambda\, a_2.t_2; \Phi_2\rangle) \Rightarrow_s p_1; \delta_1; \sigma_1}\ [\text{V-A}']$$

Acknowledgment

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

References

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 '06, 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 SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 3–15, New York, NY, USA, 2008.

Phil Bagwell. Ideal Hash Trees. *Technical Report EPFL-REPORT-169879, Ecole polytechnique fédérale de Lausanne*, November 2001.

Stefan Berghofer and Christian Urban. A Head-to-Head Comparison of De Bruijn Indices and Names. *ENTCS 174*, (5):53–67, June 2007.

William E. Byrd and Daniel P. Friedman. αKanren A Fresh Name in Nominal Logic Programming. *Université Laval Technical Report*, (DIUL-RT-0701, Scheme Workshop '07, editor Danny Dubé):79 – 90, 2007.

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

Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, October 2012.

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*, LNCS 3132, pages 269–283. Springer, Berlin, Heidelberg, September 2004.

Alonzo Church. *The Calculi of Lambda-conversion*. Princeton University Press, Humphrey Milford Oxford University Press, 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.

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

Jordi Levy and Mateu Villaret. An Efficient Nominal Unification Algorithm. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, RTA '10, pages 209–226, Edinburgh, Scottland, UK, 2010.

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

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

Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In *Extensions of Logic Programming*, LNCS 475, pages 253–281. Springer, Berlin, Heidelberg, December 1989.

Eugenio Moggi. Notions of Computation and Monads. *Inf. Comput. 93*, 93(1):55–92, July 1991.

G. Nadathur, D. Miller, University of Pennsylvania Department of Computer, and Information Science. *An Overview of Lambda Prolog*, volume 116 of *LINC LAB*. University of Pennsylvania, Department of Computer and Information Science, 1988.

Joseph P. Near, William E. Byrd, and Daniel P. Friedman. αleanTAP: A declarative theorem prover for first-order classical logic. In *Logic Programming*, LNCS 5366, pages 238–252. Springer, Berlin, Heidelberg, December 2008.

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

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

Frank Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *CADE-16*, LNCS 1632, pages 202–206. Springer, Berlin, Heidelberg, July 1999.

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

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

Christian Urban and Christine Tasson. Nominal techniques in Isabelle/HOL. In *CADE-20*, LNCS 3632, pages 38–53. Springer, Berlin, Heidelberg, July 2005.

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