Nominal Unification with Atom-Variables

Manfred Schmidt-Schauß

Goethe-University Frankfurt am Main, Germany

David Sabel

Goethe-University Frankfurt am Main, Germany

Yunus D. K. Kutz

Goethe-University Frankfurt am Main, Germany

Abstract

The problem of nominal unification where variables are allowed for atoms, and computing a complete set of unifiers is considered. The complexity is shown to be NP-complete, while for special cases there are polynomial time algorithms. The main result is a novel algorithm to compute a complete set of unifiers which performs lazy guessing of equality or disequality of atom-variables, runs in NP time, and the collecting variant has more chances to keep the complete set of unifiers small. Applications of this algorithm are in reasoning about program transformations in higher order functional languages. We also present a variant of the unification algorithm that delays guessing and checking solvability, and produces a single most general unifier.

Key words: program transformations, nominal unification, functional languages, atom-variables, correctness

1. Introduction

Motivation, Applications and Goals. The initial motivation to introduce and investigate nominal reasoning and unification was the observation that formalizations of proofs in

⋆ The first author is supported by the Deutsche Forschungsgemeinschaft (DFG) under grant under grant SCHM 986/11-1. The second author is supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SA 2908/3-1.

Email addresses: `schauss@ki.informatik.uni-frankfurt.de` (Manfred Schmidt-Schauß), `sabel@ki.informatik.uni-frankfurt.de` (David Sabel), `kutz@ki.informatik.uni-frankfurt.de` (Yunus D. K. Kutz).

higher-order deduction systems like Isabelle (Nipkow et al., 2002; Urban and Kaliszyk, 2012) would profit from a more machine-oriented formalization of reasoning about α-equivalence and for arguments in automated proofs that employ renaming of bound variables by fresh names. This leads to the development and application of nominal theories and techniques (a recent tutorial is (Pitts, 2016), a broad overview is (Pitts, 2013)) and also of nominal unification (Urban et al., 2003; Calvès and Fernández, 2008; Levy and Villaret, 2010).

An often used assumption in the technique of nominal reasoning is that variable names (called atoms) in higher-order binders are concrete, and not abstract, and that modifying the terms is done by applying atom permutations to terms and replace atoms, irrespective of their binding status (see (Urban et al., 2003)).

In the application field of verifying program transformations in higher order languages using syntactical reasoning and the operational semantics (for instance, defined by a small step reduction relation), a foundational reasoning task is to compute overlaps between left hand sides of small-step-reduction rules and transformation rules employing a unification algorithm. This task is similar to the computation of critical pairs in term rewrite systems (variants for nominal syntax were recently investigated by Ayala-Rincón et al. (2016)). A difference is that instead of all overlaps only those overlaps have to be taken into account which match the reduction strategy given by the operational semantics.

An algorithm for computing overlaps for a class of higher-order program calculi is described in (Schmidt-Schauß and Sabel, 2016): It is an algorithm for unification of higher-order expressions with meta-variables, where (among others) expression-variables and variable-variables are used. As a consequence, in that approach, there are no concrete atoms, but "abstract" atoms. For this application it is indispensable to compute complete sets of unifiers. As correctness criterion, syntactical equality is used instead of alpha-equivalence. The treatment of alpha-equivalence and binding constraints uses so-called non-capture constraints. Further reasoning on correctness and executing reductions and/or transformations requires to also use and reason about alpha-renamings, which is, however, not supported by the unification method in (Schmidt-Schauß and Sabel, 2016).

In this paper we focus on a combination of the approaches and investigate nominal unification with the feature of abstract atoms (called *atom-variables*).

We focus on unification of nominal terms with atom-variables, since it is a basic reasoning task required by several reasoning algorithms. Hence, the goal of the paper is to construct a unification algorithm for higher-order meta-expressions which comprise atom-variables. In contrast to (Schmidt-Schauß and Sabel, 2016), we leave the task of combining the methods also with extra constructs like context variables, letrec-constructs and binding chains (which is required for sophisticated reasoning in call-by-need lambda calculi) for further research.

We illustrate the differences between nominal unification with concrete atoms and on the other hand with atom-variables. As a quite small example consider the equation $\lambda x.\lambda y.x \doteq \lambda x.\lambda y.y$. If x, y are atoms, then the equation is ground on both sides (there are no unification variables) and there is no solution: nominal unification only checks alpha-equivalence which results in the unsolvable equation $x \doteq y$. But, if x, y represent unification variables which can be instantiated by atoms, then there exists a solution which instantiates x, y by the same atom a. Furthermore, nominal unification with atom-variables generalizes usual nominal unification, if so called freshness constraints are allowed in the input, since they can enforce disequality of instantiations of atom-variables.

For instance, the freshness constraint $x\#y$ ensures that atom-variables x and y must not be instantiated by the same atom. Thus, the equation $\lambda x.\lambda y.x \doteq \lambda x.\lambda y.y$ together with freshness constraint $x\#y$ represents usual nominal unification and thus has no solution.

As a motivating example for the usefulness of atom-variables in applications, we consider the following instance of a call-by-value beta reduction rule with 3-ary lambda expressions where the arguments are variables, which may be enforced by the syntax. As a small-step reduction rule one would usually write

$$(\lambda(x_1, x_2, x_3).e)\ (y_1, y_2, y_3) \to e[y_1/x_1, y_2/x_2, y_3/x_3]$$

where y_1, y_2, y_3 are variables and implicitly the binders x_1, x_2, x_3 are meant to be different. To (syntactically) reason on such a rule, it has to be represented by using meta-syntax. However, with usual nominal syntax (with atoms but without atom-variables), it is *insufficient* to represent the rule as

$$(\lambda(a_1, a_2, a_3).S)\ (b_1, b_2, b_3) \to S[b_1/a_1, b_2/a_2, b_3/a_3],$$

where S is an expression-variable and a_i, b_i are atoms (and $[\cdot/\cdot, \cdot/\cdot, \cdot/\cdot]$ is a 6-ary function symbol), since the semantics of nominal syntax enforces $a_1, a_2, a_3, b_1, b_2, b_3$ to be pairwise different atoms, however, the variables y_1, y_2, y_3 in the above description of the rule can also be equal. Thus, to represent the rule with nominal syntax, one has to add 4 more rules which cover the cases that two or three of the variables y_1, y_2, y_3 are equal:

$$\begin{aligned}
(\lambda(a_1, a_2, a_3).S)\ (b_1, b_1, b_2) &\to S[b_1/a_1, b_1/a_2, b_2/a_3]\\
(\lambda(a_1, a_2, a_3).S)\ (b_1, b_2, b_1) &\to S[b_1/a_1, b_2/a_2, b_1/a_3]\\
(\lambda(a_1, a_2, a_3).S)\ (b_2, b_1, b_1) &\to S[b_2/a_1, b_1/a_2, b_1/a_3]\\
(\lambda(a_1, a_2, a_3).S)\ (b_1, b_1, b_1) &\to S[b_1/a_1, b_1/a_2, b_1/a_3]
\end{aligned}$$

However, using nominal syntax with atom-variables the rule can be represented by one rule

$$(\lambda(A_1, A_2, A_3).S)\ (B_1, B_2, B_3) \to S[B_1/A_1, B_2/A_2, B_3/A_3]$$

where A_i, B_i are atom-variables and S is an expression-variable. Without further constraints, the rule allows A_1, A_2, A_3 being instantiated with equal atoms, but disequality of A_1, A_2, A_3 can be ensured by adding freshness constraints $A_1\#A_2, A_2\#A_3$, and $A_1\#A_3$.

As a third example, consider the following transformation rule from a call-by-need lambda calculus with sharing (for instance, in (Schmidt-Schauß et al., 2008) such a transformation rule appears, called (cpcxnoa)) which allows to copy so-called constructor applications $(c\ x_1\ \ldots\ x_n)$ provided the arguments are variables (since otherwise copying arbitrary expressions would contradict the sharing philosophy):

$$\texttt{let}\ y = (c\ x_1 \ldots x_n), z = C[y]\ \texttt{in}\ e \to \texttt{let}\ y = (c\ x_1 \ldots x_n), z = C[(c\ x_1 \ldots x_n)]\ \texttt{in}\ e$$

Again, in this rule the variables $x_1, \ldots, x_n$ may be different or equal, and thus in meta-notation they can easily be represented by atom-variables, while using atoms requires to split the rule into several cases.

Peculiarities and Results. Adding atom-variables to the nominal syntax and reasoning algorithms is not straightforward, since nominal unification explicitly deals with renamings by using explicit permutations on atoms. Replacing atoms by atom-variables complicates this approach, since several nice properties, for instance, having compact representations of permutations and also several simplification rules on permutations, no longer apply to permutations with atom-variables. Hence, our algorithm for nominal unification with atom-variables requires – compared with nominal unification with atoms – specific precautions on the treatment of permutations (see Examples 4.1 and 4.2 below and Remark 3.10 in (Urban et al., 2003), discussed in Example 2.12). Thus, we will substantially extend the machinery from nominal unification (Urban et al., 2003), who assume that atoms are distinct.

Throughout the paper, we will work with different languages for expressions which all are built from function symbols and lambda-binders. The following table summarizes the used languages, which we call NL_a, NL_{aS} and NL_{AS} [1]:

NL_a	ground nominal expressions (with ground atoms in binders), and no permutations.
NL_{aS}	nominal expressions with expression-variables, ground atoms in binders, and ground atom permutations
NL_{AS}	nominal expressions with expression-variables, atom-variables in binders, and atom-variable permutations

It will not be necessary to treat a language which contains both, atoms and atom-variables, since disequality of atom-variables can be expressed by freshness constraints (similar as in (Lakin, 2011)).

The following *main results* are obtained in this paper: A sound and complete algorithm AVNomUnify for nominal unification with atom-variables and freshness constraints is constructed, together with some subalgorithms. As a decision algorithm it runs in NP time, and as a collecting algorithm, it outputs an at most exponential number of unifiers where every unifier is represented in polynomial space (see Theorem 5.6). The algorithm is designed such that it has a good chance to compute a small, complete set of unifiers. Polynomiality of our algorithm is achieved by using the Martelli-Montanari-style (Martelli and Montanari, 1982) of unification algorithms. Proposition 6.1 shows that a variant of our unification algorithm produces a single most general unifier provided the input is solvable, however, it delays deciding the satisfiability to solving the freshness constraints. As a novel observation, we show that nominal unification with atom-variables, but without any freshness constraint in the input, can be decided in polynomial time (see Theorem 3.6).

Related Approaches There are extensions and restrictions of nominal unification. A quite general nominal constraint solving is considered in (Cheney, 2004b) where besides atom-variables (as in our setting) also permutation variables are permitted in the problems. However, there is no complete and in-NP algorithm for computing unifiers. Also the overall goal in (Cheney, 2004b) is to solve a different variant of nominal unification –

[1] Our (generic) naming of languages is different from the one in the literature.

called *equivariant unification* (Cheney, 2004a, 2010)), which requires that the instantiated equations of a nominal unification problem still hold under application of atom permutations.

A further related approach are *nominal constraints of non-permutative nominal abstract syntax* (NPNAS) analyzed by Lakin (2011). Those constraints are nominal constraints in a language with atom-variables, but without permutations, and thus the used syntax is more restricted than ours. Lakin (2011) shows that the expressivity is the same as Cheney's name-name equivariant unification (Cheney, 2004a, 2010)) (where unification equations equate name terms instead of full terms). He also constructs an algorithm that proceeds by exploiting the deeper structure of nominal expressions using so-called narrowing (a form of lazy instantiation), and shows that it is a decision algorithm. However, there is no proof of an upper bound on the used space or the running time, and it does not compute a complete set of unifiers or solutions. Since our syntax can represent NPNAS-constraints, our algorithms are directly applicable to solve NPNAS-constraints (without types). For the other direction – i.e. solving nominal unification problems with atom-variables using NPNAS-constraints there is no obvious polynomial time encoding, while the first phase algorithm of Cheney (2004a, 2010) provides an exponential time encoding: non-deterministic guessing of equalities/disequalities of atom-variables with subsequent removal of permutations leads to a non-deterministic polynomial time encoding.

Structure of the paper. Section 2 contains the preliminary definitions and notations, and the definitions of nominal syntax. After briefly recalling the problem and the result for usual nominal unification, the nominal unification problem with atom-variables is introduced. In Section 3 simple but rather inefficient algorithms for the general case of variable atom nominal unification are described and several complexity bounds also for special cases. The second part starts with Section 4 that describes the improved algorithm AVNomUnify, which performs non-deterministic (dis-)equality choices as lazy as possible accompanied by an extended set of unification rules. Section 5 contains proofs of soundness and completeness and complexity of the algorithm AVNomUnify. In Section 6 we sketch an algorithm that produces a single most general unifier. We conclude in Section 7.

2. An Algorithm for Nominal Unification with Variables

In this section we recall usual nominal unification with atoms and its results, introduce the generalized languages with atom-variables, introduce the required notation and define the variable-atom nominal unification problem.

2.1. Nominal Terms

In this section we briefly recall nominal expressions (sometimes also called nominal terms). Let $\mathcal{F}$ be a set of function symbols $f \in \mathcal{F}$, s.t. each f has a fixed arity $ar(f) \geq 0$. Let $\mathcal{A}t$ be *the set of atoms* ranged over by a, b. The ground language NL_a consists of atoms, lambda-expressions which bind atoms, and function symbols.

Definition 2.1 (The nominal language NL_a)**.** The syntax of the language NL_a is defined by the following grammar:

$$e \in NL_a ::= a \mid (f\ e_1 \dots e_{ar(f)}) \mid \lambda a.e$$

With $At(e)$ we denote the set of atoms contained in e.

As a convention, we assume throughout this paper, that differently named atoms are different. For convenience, we also use tuples in the language and view them as applications $(f\ e_1 \dots e_{ar(f)})$ not naming the function symbol.

For atoms a, b, a *swapping* $(a\ b)$ represents the function $\{a \mapsto b, b \mapsto a\} \cup \{c \mapsto c \mid c \notin \{a, b\}\}$. In particular, $(a\ b)$ is identical to $(b\ a)$. A list of swappings π is called a *permutation*, i.e. a bijective mapping that maps atoms to atoms. We write $\emptyset$ for the empty permutation (containing no swappings), and we write $\pi_1 \cdot \pi_2$ for concatenation (composition, resp.) of permutations π_1 and π_2. In abuse of notation we also write $(a\ b) \cdot \pi$ or $\pi \cdot (a\ b)$ to prepend or append a swapping $(a\ b)$ to a permutation. Additionally, we write $\pi{\cdot}e$ (and $(a\ b){\cdot}e$, resp.) for the application of a permutation (a swapping, resp.) to a term e. Similarly, we apply the permutations and swappings to atoms. The domain of permutations is defined as $dom(\pi) = \{a \in At \mid \pi(a) \neq a\}$.

For ground expressions an application $\pi{\cdot}a$ can be evaluated by the following rules:

$$\begin{array}{rclrcl}
\emptyset \cdot a & := & a & \pi \cdot (a\ b){\cdot}c & := & \pi{\cdot}c, \\
\pi \cdot (a\ b){\cdot}a & := & \pi{\cdot}b & \pi{\cdot}\lambda a.e & := & \lambda \pi{\cdot}a.\pi{\cdot}e \\
\pi \cdot (a\ b){\cdot}b & := & \pi{\cdot}a & \pi{\cdot}(f\ e_1 \dots e_{ar(f)}) & := & (f\ \pi{\cdot}e_1 \dots \pi{\cdot}e_{ar(f)})
\end{array}$$

Compositions $\pi_1 \circ \pi_2$ and inverses π^{-1} can be immediately computed as follows: compositions are the concatenation of the two lists of swappings, and an inverse is a reverse of the list.

For an atom a and a term e the construct $a\#e$ is called a *freshness constraint.* A freshness constraint holds iff atom a does not occur free in e. For ground expressions the following rules decide whether a freshness constraint holds:

$$\frac{}{a\#b} \qquad \frac{}{a\#\lambda a.e} \qquad \frac{a\#e}{a\#\lambda b.e} \qquad \frac{\forall i : a\#e_i}{a\#(f\ e_1 \dots e_{ar(f)})}$$

Thus, in NL_a, all expressions $\pi{\cdot}e$ can be transformed into expressions without any permutations. In NL_a, also all freshness constraints can immediately be evaluated.

Definition 2.2 (Syntactic α-equivalence $\sim$ in NL_a)**.** The equivalence $\sim$ on expressions $e \in NL_a$ is inductively defined by the following rules:

$$\frac{}{a \sim a} \qquad \frac{\forall i : e_i \sim e'_i}{(f\ e_1 \dots e_{ar(f)}) \sim (f\ e'_1 \dots e'_{ar(f)})} \qquad \frac{e \sim e'}{\lambda a.e \sim \lambda a.e'} \qquad \frac{a\#e' \wedge e \sim (a\ b){\cdot}e'}{\lambda a.e \sim \lambda b.e'}$$

Note that the relation $\sim$ is identical to the equivalence relation generated by α-equivalence by renaming binders, which can be proved in a simple way by arguing on the (binding-)structure of expressions (using de Bruijn-indices), and hence $\sim$ is an equivalence relation on NL_a. It is also a congruence on NL_a, i.e., for a context C, we have $e_1 \sim e_2$ implies $C[e_1] \sim C[e_2]$.

Remark 2.3. A permutation π represented as a list of swappings in NL_a can be represented using at most $|At(\pi)| - 1$ swappings, where $At(\pi)$ is the set of ground atoms occurring in π. Hence the number of swappings in a permutation in NL_a can be bounded linearly in the number of atoms. This can be seen as follows: The behavior of a permutation as a function can be split into (disjoint) cycles. Consider a single cycle $\{a_1 \mapsto a_2, a_2 \mapsto a_3, \ldots, a_{n-1} \mapsto a_n, a_n \mapsto a_1\}$ of π. This can be encoded by the following swapping list: $(a_1\ a_2)(a_2\ a_3)\ldots(a_{n-1}\ a_n)$. The only nontrivial check is the image of a_n, which is a_1 by sequential application of the swappings. The length of the list is at most $|At(\pi)| - 1$.

We write $\pi \equiv \pi'$, if the permutations π, π' are identical as functions.

2.2. Nominal Unification

In this section we recall the nominal unification problem and results on solving it. For unification we extend the syntax of expressions by variables representing expressions. Let $\mathcal{S}$ be *the set of variables standing for expressions*, expression-variables, for short, where elements of $\mathcal{S}$ are ranged over by S, T. Nominal-unification expressions extend ground expressions by variables S, T for expressions and by swappings and permutations. In the syntax below we also overload the symbol $\cdot$ by using it for (syntactic) application of a permutation (or a swapping, resp.) to an expression-variable.

Definition 2.4 (The nominal language NL_{aS})**.** The language NL_{aS} is defined by the following grammar:

$$\begin{aligned} e \in NL_{aS} &::= a \mid S \mid \pi{\cdot}S \mid (f\ e_1 \ldots e_{ar(f)}) \mid \lambda a.e \\ \pi &::= \emptyset \mid (a\ a') \cdot \pi \end{aligned}$$

The inclusion of syntactic swappings and permutations is necessary, since terms like $\pi{\cdot}S$ cannot be further evaluated. Such terms are called *suspensions.* It is not necessary to distinguish between $(\pi_1 \cdot \pi_2) \cdot S$ and $\pi_1 \cdot (\pi_2 \cdot S)$, since semantically they represent the same expressions. Let $ExVar(e)$ be the set of expression-variables contained in e.

A *substitution* σ is a finite mapping of expression-variables S to expressions $e \in NL_{aS}$. In the following we write $e\sigma$ or $\sigma(e)$ for applying a substitution σ to an expression e. For substitutions σ_1, σ_2, we denote with $\sigma_1 \circ \sigma_2$ the composition of σ_1, σ_2, s.t. $e(\sigma_1 \circ \sigma_2) = \sigma_2(\sigma_1(e))$. These notations will also be used for other substitutions in this paper.

We recall the definition of a nominal unification problem and define solutions and unifiers by using the ground term semantics:

Definition 2.5. A *nominal unification problem* is a pair (Γ, ∇) where Γ is a finite set of equations $e \doteq e'$ with $e, e' \in NL_{aS}$ and ∇ is a finite set of freshness constraints $a\#e$ where a is an atom and $e \in NL_{aS}$.

A substitution is *ground* for (Γ, ∇) if it maps all variables $S_1, \ldots, S_n$ in (Γ, ∇) to expressions $e \in NL_a$. A ground substitution σ is a *solution* of a nominal unification problem (Γ, ∇) iff $e\sigma \sim e'\sigma$ for all $(e \doteq e') \in \Gamma$ and $a\#e\sigma$ is valid for all $(a\#e) \in \nabla$.

A nominal unification problem (Γ, ∇) is *solvable* if, and only if there is a solution for (Γ, ∇). A set of freshness constraints ∇ is *solvable* if, and only if $(\emptyset, \nabla)$ is solvable.

For a nominal unification problem (Γ, ∇), a *unifier* is a pair (σ, ∇') where σ is a substitution and ∇' is a set of freshness constraints, s.t. ∇' is solvable and for all ground substitutions γ: $(\forall a\#e \in \nabla' : a\#e\sigma\gamma$ is valid $) \implies (\sigma \circ \gamma)$ is a solution for (Γ, ∇)

For a nominal unification problem (Γ, ∇), a set M of unifiers is *complete*, iff for every solution ρ of (Γ, ∇), there is a unifier $(\sigma, \nabla') \in M$ such that there is a ground substitution γ with $S\sigma\gamma \sim \rho(S)$ for all variables S occurring in (Γ, ∇). A unifier (σ, ∇') is a *most general unifier* of (Γ, ∇), if $\{(\sigma, \nabla')\}$ is a complete set of unifiers for (Γ, ∇).

Theorem 2.6 (Urban et al. (2003); Calvès and Fernández (2008); Levy and Villaret (2008, 2010))**.** *The nominal unification problem in NL_{aS} is solvable in quadratic time. Moreover, for a solvable nominal unification problem* (Γ, ∇)*, there exists a most general unifier, which can be computed in polynomial time.*

2.3. Introducing Atom-Variables

We now introduce the extension of nominal terms and nominal unification by atom-variables. The idea is to allow variables at the position of atoms which semantically can be instantiated by atoms. However, it is not necessary to deal with a language that has both atoms and atom-variables, since it is sufficient to replace atoms by atom-variables and to add freshness constraints which ensure that two different variables always mean different atoms. The semantics of these meta-expressions however, are subsets of the set of all NL_a-expressions.

We adapt the previously introduced notions to also cover atom-variables. Thus, let $\mathcal{A}$ be the *set of atom-variables* ranged over by A, B, and let atom-variable swappings $(A\ B)$ be swappings of atom-variables (with the semantics that $(A\ B)$ represents all swappings of atoms which can be derived by instantiating the atom-variables A and B by (concrete) atoms).

We use atom-variable permutations π which are a list of atom-variable swappings. We also can compute the composition $\pi_1 \circ \pi_2$ and inverses π^{-1} for atom-variable permutations by list concatenation and list reversal, respectively. The domain of atom-variable permutations is defined as $dom(\pi) = \{A \mid \pi(A) \neq A\}$.

Definition 2.7 (The nominal language NL_{AS} with atom-variables)**.** The grammar of the language NL_{AS} is

$$\begin{aligned} e &::= A \mid S \mid \pi{\cdot}A \mid \pi{\cdot}S \mid (f\ e_1 \ldots e_{ar(f)}) \mid \lambda\pi{\cdot}A.e \\ \pi &::= \emptyset \mid (A\ A') \cdot \pi \end{aligned}$$

With $AtVar(e)$ we denote the set of atom-variables contained in e, and as before with $ExVar(e)$ we denote the set of expression-variables contained in e. With $tops(e)$ we denote the top-symbol of expression e, i.e. $tops(A) := A$, $tops(S) := S$, $tops(\pi{\cdot}A) := A$, $tops(\pi{\cdot}S) := S$, $tops(f\ e_1 \ldots\ e_{ar(f)}) = f$, and $tops(\lambda A.e) = \lambda$.

Note that it is not necessary to allow permutations π on every expression, since they can always be shifted into the expression, using the following operations to shift permutations π into the expressions and derive an NL_{AS}-expression:

$$\begin{aligned} &\pi{\cdot}(f\ e_1 \ldots e_{ar(f)}) &&\to (f\ \pi{\cdot}e_1\ \ldots\ \pi{\cdot}e_{ar(f)}), \\ &\pi{\cdot}(\lambda A.e) &&\to \lambda(\pi{\cdot}A).(\pi{\cdot}e),\ \text{and} \\ &\emptyset{\cdot}e &&\to e \end{aligned}$$

In NL_{AS}, we call the constructs $\pi \cdot A$ and $\pi \cdot S$ *suspensions.* Note that in general it is not possible to evaluate the application of an atom-variable swapping to an atom-variable, for instance $(A\ B) \cdot C$, since the result depends on whether $A = C$ and/or $B = C$. A thorough treatment will be given in Section 4.

A freshness constraint for the language NL_{AS} is of the form $A\#e$ where e is an NL_{AS}-expression and A is an atom-variable.

In NL_{AS} we have two kinds of substitutions. On the one hand we have substitutions which map expression-variables to expressions in NL_{AS} and atom-variables to atom-variables or suspensions of atom-variables. On the other hand we will use substitutions (called *ground substitutions*), which are also translations from NL_{AS} into NL_a: They map expression-variables to expressions in NL_a and atom-variables to atoms. We will use both kinds of substitutions where the kind is always clear from the context. Furthermore, both kinds of substitutions can be extended in a natural way also to expressions, where variables A are replaced at every position, also at binder position: for example let $\sigma = \{A \mapsto B; S \mapsto \lambda A'.A'\}$; then $(\lambda A.\lambda A.(A, B, S))\sigma = (\lambda B.\lambda B.(B, B, \lambda A'.A'))$.

Definition 2.8. Let Γ be a set of equations of the form $e_1 \doteq e_2$ with NL_{AS}-expressions e_1, e_2, and let ∇ be a set of freshness constraints over NL_{AS}. Then (Γ, ∇) is a *variable-atom nominal unification problem* (*VANUP*).

We define solutions and unifiers for *VANUP*s w.r.t. the ground term semantics, i.e. w.r.t. NL_a-expressions:

Definition 2.9 (Solution of a *VANUP*)**.** A ground substitution ρ is a *solution* of (Γ, ∇), iff for all equations $e_1 \doteq e_2$ in Γ: $e_1\rho \sim e_2\rho$, and for all freshness constraints $A\#e \in \nabla$ the freshness constraint $A\rho\#e\rho$ is valid. A variable-atom nominal unification problem (Γ, ∇) is *solvable* if, and only if a solution for (Γ, ∇) exists. A set of freshness constraints ∇ is *solvable* if, and only if there is a solution for $(\emptyset, \nabla)$.

Definition 2.10 (Unifier of a *VANUP*)**.** Let (Γ, ∇) be a variable-atom nominal unification problem. A pair (σ, ∇') is a *unifier* of (Γ, ∇) iff

∇' is solvable and for all ground substitutions γ:

$$(\forall A\#e \in \nabla' : A\sigma\gamma\#e\sigma\gamma \text{ is valid }) \implies \sigma \circ \gamma \text{ is a solution for } (\Gamma, \nabla)$$

For a variable-atom nominal unification problem (Γ, ∇), a set M of unifiers is *complete*, iff for every solution ρ of (Γ, ∇), there is a unifier $(\sigma, \nabla') \in M$, such that there is a ground substitution γ with $\sigma\gamma(A) = \rho(A)$ and $\sigma\gamma(S) \sim \rho(S)$ for all atom-variables A and expression-variables S occurring in (Γ, ∇).

A unifier (σ, ∇') is a most general unifier of (Γ, ∇), if $\{(\sigma, \nabla')\}$ is a complete set of unifiers for (Γ, ∇).

Note that a *VANUP* may have more than one unifier.

Example 2.11. The equation $\lambda A.A \doteq \lambda B.S$ has a solution $\{A \mapsto a, B \mapsto b, S \mapsto b\}$. It has a unifier $\{S \mapsto B\}$ with empty set ∇, which is indeed most general.

The equation $\lambda A.(A, S) \doteq \lambda B.(A\ B) \cdot (S, B)$ has a solution $\{A \mapsto a, B \mapsto a, S \mapsto a\}$. A unifier is: $\{B \mapsto A, S \mapsto A\}$ with empty set ∇. Note that there is no nominal unifier of the problem that corresponds to treating A, B as different atoms: $\lambda a.(a, S) \doteq \lambda b.(a\ b) \cdot (S, b)$ is not nominal unifiable, since $a\#(a\ b) \cdot (S, b)$ is not valid.

Example 2.12. Consider the example $(A\ B){\cdot}C \doteq C$ (see Remark 3.10 in (Urban et al., 2003)). The set $\{(\{A\#C, B\#C\}, Id), (\emptyset, \{A \mapsto C, B \mapsto C\})\}$ is complete and contains two unifiers, which are incomparable. In the literature, this example is used as justifying the conjecture that sometimes a set of at least two unifiers is necessary for completeness instead of a (single) most general unifier. However, according to our definitions and since we allow rather general freshness constraints, the pair $(\{C\#\lambda(A\ B){\cdot}C.C\}, Id)$ is a most general unifier. The idea is exploited in Section 6 to argue that a single most general unifier is sufficient, where, however, the solvability test has to be shifted to the freshness constraints.

Remark 2.13. Note that a definition of α-equivalence on the language NL_{AS} (or also on NL_{AS}-expressions without expression-variables) is meaningless, since such a notion is not stable under substitutions: for example $\lambda A.\lambda B.(A, B)$ would be α-equivalent to $\lambda C.\lambda D.(C, D)$ under an appropriate definition, but with $\sigma = \{A \mapsto B\}$, we have $(\lambda A.\lambda B.(A, B))\sigma = \lambda B.\lambda B.(B, B)$ which is not α-equivalent to $\lambda C.\lambda D.(C, D)$.

We introduce some notation for freshness constraints:

Definition 2.14. Let ∇ be a set of atom-variable freshness constraints. In the following we write $\nabla \vdash A \neq B$, if $A\#B \in \nabla$ or $B\#A \in \nabla$. More general, for two sets M_1, M_2 of atom-variables, we write $\nabla \vdash M_1\#M_2$, if for all pairs $A_1 \in M_1, A_2 \in M_2 : \nabla \vdash A_1 \neq A_2$. We also write $\nabla \vdash \#M$, if for all pairs of variables $A \neq B$ in M, we have $\nabla \vdash A \neq B$.

3. Complexity of Atom-Variable Nominal Unification

In this section we show that the variable-atom nominal unification problem is NP-complete, and that it is polynomial in two special cases. The NP-completeness result follow from the results in Cheney (2004b, 2010), where NP-hardness also follows from Lakin (2011). However, since our expression syntax (and also several notations) is different, and for the sake of completeness, and since the NP-hardness encoding is different, we include full proofs. We will explain the simple brute-force non-deterministic guessing algorithm, which translates atom-variable nominal unification into usual nominal unification.

Theorem 3.1. *Solvability of atom-variable freshness constraints is NP-hard.*

Proof. We show that the MONOTONE ONE-IN-THREE-3-SAT-problem (Schaefer, 1978) is reducible to satisfiability of atom-variable freshness. Let $\mathcal{C} := \{\{p_{i,1}, p_{i,2}, p_{i,3}\} \mid i = 1, \dots n\}$ be an instance of the MONOTONE ONE-IN-THREE-3-SAT-problem. Here $p_{i,j}$ are the propositional variables, $\{p_{i,1}, p_{i,2}, p_{i,3}\}$ are clauses, and the question is whether there is a valuation ρ that assigns `true` to exactly one literal in every clause, and `false` to the others. This problem is known to be NP-complete.

Let *True* be an atom-variable, and let $A_{i,j}$ be atom-variables (perhaps not different). The following freshness constraints encode that the three variables in single clauses are interpreted as pairwise different atoms (∇_1) and that at least one variable of each clause is equal to the atom-variable *True* (∇_2):

$$\begin{aligned}
\nabla_1 &:= \{A_{i,1}\#A_{i,2}, A_{i,2}\#A_{i,3}, A_{i,3}\#A_{i,1} \mid i = 1, \ldots n\} \\
\nabla_2 &:= \{\mathit{True}\#\lambda A_{i,1}.\lambda A_{i,2}.\lambda A_{i,3}.\mathit{True} \mid i = 1, \ldots, n\} \\
\nabla &:= \nabla_1 \cup \nabla_2
\end{aligned}$$

Obviously, this is a polynomial encoding. If there is a solution ρ of the freshness constraints ∇, then this also solves the MONOTONE ONE-IN-THREE-3-SAT-problem by constructing the following model M: $M(p_{i,j}) =$ `true` iff $\rho(A_{i,j}) = \rho(\mathit{True})$ and $M(p_{i,j}) =$ `false`, otherwise. Furthermore, if $\mathcal{C}$ is solvable, then also the constraint ∇ is solvable: For a model M of $\mathcal{C}$, construct the solution ρ for ∇ with $\rho(\mathit{True}) = b$, and if $M(p_{i,j}) =$ `true` then $\rho(A_{i,j}) = b$ else $\rho(A_{i,j}) := a_{i,j}$, where b, and all $a_{i,j}$ are pairwise different atoms. □

Theorem 3.2. *The variable-atom nominal unification problem, where ∇ consists only of freshness constraints of the form $A\#B$, is NP-hard.*

Proof. We show that the atom-variable freshness constraints $(\nabla_1 \cup \nabla_2)$ in the proof of Theorem 3.1 are solvable if, and only if the *VANUP* $(\Gamma, \nabla_1 \cup \nabla_3)$ is solvable where

$$\begin{aligned}
\Gamma &:= \{\lambda \mathit{True}.S_i \doteq \lambda B.\lambda A_{i,1}.\lambda A_{i,2}.\lambda A_{i,3}.\mathit{True} \mid i = 1, \ldots n\} \\
\nabla_3 &:= \{\mathit{True}\#B\}
\end{aligned}$$

where ∇_1, ∇_2 are defined as in the proof of Theorem 3.1, $S_1, \ldots, S_n$ are expression-variables, and B is an atom-variable which is different from *True* and all $A_{i,j}$.

First, let ρ_0 be a solution of $(\nabla_1 \cup \nabla_2)$. We construct a solution ρ_1 of $(\Gamma, \nabla_1 \cup \nabla_3)$: $\rho_1(A_{i,j}) := \rho_0(A_{i,j})$ for all i, j, $\rho_1(\mathit{True}) := \rho_0(\mathit{True})$, $\rho_1(B) := b$ where b is a fresh atom that is different from $\rho_0(\mathit{True})$ and all atoms $\rho_0(A_{i,j})$, and let $\rho_1(S_i) := \lambda\rho_1(A_{i,1}).\lambda\rho_1(A_{i,2}).\rho_1(A_{i,3}).\rho_1(\mathit{True})$. By applying ρ_1 to $(\Gamma, \nabla_1 \cup \nabla_3)$ one can verify that ρ_1 is a solution.

Now let σ_0 be a solution of $(\Gamma, \nabla_1 \cup \nabla_3)$: We show that σ_0 is a solution of ∇_2: Since σ_0 is a solution of Γ, we have $\sigma_0(\lambda \mathit{True}.S_i) \sim \sigma_0(\lambda B.\lambda A_{i,1}.\lambda A_{i,2}.\lambda A_{i,3}.\mathit{True})$ and thus $\lambda\sigma_0(\mathit{True}).\sigma_0(S_i) \sim \lambda\sigma_0(B).\lambda\sigma_0(A_{i,1}).\lambda\sigma_0(A_{i,2}).\lambda\sigma_0(A_{i,3}).\sigma_0(\mathit{True}))$. Since σ_0 is a solution of ∇_3 we also have $\sigma_0(\mathit{True}) \neq \sigma_0(B)$ and thus by the definition of alpha-equivalence $\sim$ the constraint $\sigma_0(\mathit{True})\#\lambda\sigma_0(A_{i,1}).\lambda\sigma_0(A_{i,2}).\lambda\sigma_0(A_{i,3}).\sigma_0(\mathit{True}))$ must hold. Since $(\Gamma, \nabla_1 \cup \nabla_3)$ satisfies the conditions of the claim, the theorem holds. □

Nominal unification is an instance of variable atom nominal unification, if atom-variables cannot be made equal by a solution (such a remark can also be found in (Lakin, 2011)):

Theorem 3.3. *The variable-atom nominal unification problem (Γ, ∇), where ∇ consists of all possible $A\#B$-freshness constraints, for all $A \neq B$ occurring in Γ, is solvable in polynomial time. Moreover, if (Γ, ∇) is solvable then there is a most general unifier, which can be computed in polynomial time.*

Proof. Let (Γ, ∇) be the *VANUP*. We can encode the problem into the nominal unification problem Γ' such that Γ' is Γ where every atom-variable A_i occurring in Γ is replaced by a fresh atom a_i. Due to Theorem 2.6, in order to prove the first part, it suffices to show that (Γ, ∇) is solvable iff Γ' is solvable.

Let us first assume that Γ' is solvable, i.e. there exists a ground substitution ρ, s.t. $(e\rho \sim e'\rho)$ for all $e \doteq e' \in \Gamma'$. Let $\rho_0 = \rho \cup \{A_i \mapsto a_i \mid A_i \text{ occurs in } \Gamma\}$. Then ρ_0 is a solution for (Γ, ∇).

Now assume that (Γ, ∇) is solvable. Then there exists a ground substitution ρ s.t. $e\rho \sim e'\rho$ for all $e \doteq e' \in \Gamma$ and $A\rho \# B\rho$ for all $A\#B \in \nabla$. Clearly ρ must map each atom-variable A to a unique atom a. Let ρ_S be the restriction of ρ to expression-variables and let ρ_A be the restriction of ρ to atom-variables. Then clearly (up to a renaming of atoms) the set Γ' consists exactly of the equations $e\rho_A \doteq e'\rho_A$ (for all $e \doteq e' \in \Gamma$) and ρ_S is a solution of Γ.

The same construction can be used to show the second part of claim: Let (σ_a, ∇_a) be a most general unifier of Γ'. Then let $\mu(a_i) = A_i$ for $i = 1, \ldots, n$ be a mapping and (σ_A, ∇_A) is defined such that $\sigma_A = \sigma_a \circ \mu$, restricted to the (atom- and expression-) variables and $\nabla_A := (\nabla_a)\mu$. The claim is that (σ_A, ∇_A) is a most general unifier of (Γ, ∇). If ρ is a solution of (Γ, ∇), then let $b_i = \rho(A_i)$ for $i = 1, \ldots, n$ be the atoms used in ρ. Since the names of the atoms can be changed using a bijection mapping atoms to atoms, using $\mu'(b_i) = A_i$ for $i = 1, \ldots, n$, the substitution $\rho\mu'$ (restricted to S-variables) is the substitution ∇_A of the unifier. The freshness constraints can be verified in the same way. □

Theorem 3.4. *The variable-atom nominal unification problem is in NP.*
Moreover, there is an exponential time algorithm to compute a complete set of unifiers of polynomial size for every input.

Proof. Assume given a variable-atom nominal unification problem (Γ, ∇). Then as a first step, guess for all pairs A, B of atom-variables A, B occurring in $AtVar(\Gamma, \nabla)$, whether for a ground solution γ: $\gamma(A) = \gamma(B)$ or $\gamma(A) \neq \gamma(B)$. If $\gamma(A) = \gamma(B)$, then replace A by B in Γ and ∇, otherwise, add the freshness constraint $A\#B$. This results in (Γ', ∇'). This guessing and replacement can be done in polynomial time. Now for all atom-variable pairs A, B occurring in Γ', there is a freshness constraint $A\#B$ or $B\#A$ in ∇'. Now Theorem 3.3 shows that solvability of (Γ', ∇') can be decided in polynomial time, and that a most general unifier can be computed in polynomial time.

Scanning all possibilities results in a deterministic exponential-time algorithm for this problem. □

Theorems 3.1 and 3.4 imply:

Theorem 3.5. *The variable-atom nominal unification problem is NP-complete.*

However, if there are no freshness constraints at all, then solvability of VANUPs can be checked efficiently, by instantiating all atom-variables with the same atom, and then checking for solvability.

Theorem 3.6. *Solvability of variable-atom nominal unification problems where $\nabla = \emptyset$ can be decided in polynomial time.*

Proof. Let Γ be a variable atom nominal unification problem. Assume that Γ is solvable by a ground substitution γ that replaces atom-variables by atoms and expression-variables by expressions from NL_a. Then, the following structural equality $\equiv_0$ holds for all equations $s \doteq t$ after instantiation: $s\gamma \equiv_0 t\gamma$, where $\equiv_0$ is recursively defined as follows: $a \equiv_0 b$ for all atoms a, b; $\lambda a.s \equiv_0 \lambda b.t$ iff $s \equiv_0 t$, and $f\ s_1 \ldots s_n \equiv_0 f\ t_1 \ldots t_n$ iff $s_i \equiv_0 t_i$ for all i. This is equivalent to the statement that γ_0 is a solution of Γ, where γ_0 replaces all atom-variables by the same atom.

The latter condition can be checked algorithmically as follows:
First apply a translation ψ to Γ with the following definition, where *lam* is a fresh unary function symbol, and a_0 fresh atom. Let $\psi(A) := a_0$ for all atom-variables A, $\psi(\lambda a.e) := (lam\ (\psi(e))$, and $\psi(f\ e_1 \ldots e_{ar(f)}) := (f\ \psi(e_1) \ldots \psi(e_{ar(f)}))$. The result is a first-order unification problem with variables S, and with the property that Γ is solvable as variable atom nominal unification problem iff $\psi(\Gamma)$ is solvable as a first-order unification problem. This test can be performed in polynomial time, since first-order unification is a polynomial time algorithm. $\square$

Note that the algorithm of Theorem 3.6 is a decision algorithm and that it is not complete in general w.r.t. unifiers.

4. Nominal Unification Algorithm with Lazy Disequality Guessing

The goal of this section is to describe a solution algorithm for *VANUPs* of non-deterministic polynomial time complexity, such that its collecting version has a small number of solutions by using simplifications and by avoiding unnecessary (dis-)equality-guessings for pairs A, B of atom-variables.

A difference to nominal unification is that in variable-atom nominal unification problems, besides suspensions $\pi{\cdot}S$ (as they occur in usual nominal unification (Urban et al., 2003)), there are further expressions that cannot be simplified. For *VANUP*s, an *atom-variable suspension* $\pi{\cdot}A$ can also in general not be simplified. However, if more is known about equality and disequality of the atom-variables in π, then there are chances to further simplify it. Also freshness constraints in NL_{AS} can not in every case be brought into a simpler form; for example $A\#(\lambda B.e)$ cannot be further processed if the (dis-)equality of A, B is unknown. The same holds for $A\#B$, and $A\#(\pi \cdot S)$.

A hazard is the instantiation of A with $\pi{\cdot}B$, which potentially may generate (nested) swappings like $(\pi{\cdot}B\ B')$, which will produce too complex expressions in (Γ, ∇). We will avoid this in the following in (Γ, ∇), but it may occur in the computed unifier.

The main ideas of the algorithm in this section are: (1) find out and store, which atom-variable pairs are already known to be instantiated with different atoms; and (2) indicate the pairs A, B of atom-variables, where a guessing of (dis-)equality of A, B makes progress.
Examples for the justification of a guessing are: (i) equations of the form $\lambda A.e \doteq \lambda B.e'$, (ii) freshness constraints of the form $A\#(\lambda B.e)$, (iii) potential substitution components $A \mapsto \pi{\cdot}B$, where A and B occur in swappings, (iv) suspensions $\pi{\cdot}S$, where π has a large number of swappings, and (v) expressions of the form $\pi{\cdot}A$, where π cannot be further simplified.

4.1. Properties of Atom-Variable Permutations

In this section we analyze properties of permutations on atom-variables, in particular that of the representation. We also analyze properties of permutations as part of a VANUP, as far as these are required in the algorithm.

Let us assume that atom-variable permutations are represented as lists of atom-variable swappings. The following properties hold for those permutations: Two permutations can be composed by concatenating the lists of the respective swappings; the inverse π^{-1} of a permutation π is the reversed list of swappings of π; and $(A\ A)$ and $(A\ B)\cdot(A\ B)$ are the identity.

Note that this remains correct under replacing atom-variables by atom-variables (or by atoms), which may make two different variables equal. However, further operations which are correct on atom permutations may be wrong on atom-variable permutations:

Example 4.1. The permutation $(a\ b)\cdot(c\ d)\cdot(a\ b)$ represents the same function as $(c\ d)$, since atoms with different names are always assumed to be different. Now consider the permutation on atom-variables $\pi = (A\ B)\cdot(C\ D)\cdot(A\ B)$. The permutation cannot be replaced by $(C\ D)$, since this is not valid for every instantiation: For example if $\rho(A) \neq \rho(B)$, $\rho(A) \neq \rho(D)$, $\rho(B) \neq \rho(D)$, but $\rho(B) = \rho(C)$, then $\rho(\pi \cdot A) = \rho(D)$ while $\rho((C\ D)\cdot A) = \rho(A) \neq \rho(D)$.

The following example shows that the interpretation of permutations as functions on atom-variables is not stable under instantiations. Thus, the usual interpretation as functions is misleading. A better intuition is that atom-variable permutations are representations of sets of atom permutations (after a substitution).

Example 4.2. Let $\pi_1 = (A\ B)\cdot(B\ C)$ and let $\pi_2 = (A\ C)\cdot(A\ B)$. Under an interpretation as operating on different atom-variables, both permutations are the mapping $\{A \mapsto B, B \mapsto C, C \mapsto A\}$. However, after the instantiation $A = C$, π_1 is the identity, whereas $\pi_2 = (A\ B)$.

Example 4.3. An example with a possible simplification of a permutation represented with atom-variables is as follows:
The permutation $(A\ B)\cdot(C\ D)\cdot(C\ A)\cdot(C\ A)\cdot(A\ B)$ can be simplified to $(C\ D)$ under the assumption $\nabla \vdash \{A, B\}\#\{C, D\}$ in the following steps:

$$\begin{aligned}
&(A\ B)\cdot(C\ D)\cdot(C\ A)\cdot(C\ A)\cdot(A\ B)\\
&\to (A\ B)\cdot(C\ D)\cdot(A\ B)\\
&\to (A\ B)\cdot(A\ B)\cdot(C\ D)\\
&\to (C\ D)
\end{aligned}$$

For atom-variable permutations in contrast to atom permutations, it is unclear how long lists of swappings can be shortened. So the unification algorithm has to prevent exponential growth of the length of permutations, by reasoning about equivalent but shorter representations and by guessing (dis-)equality of atom-variable pairs A, B.

In the remainder of the paper we allow to write $\pi{\cdot}e$ for arbitrary NL_{AS}-expressions, where "$\cdot$" means the following function to shift permutations downwards, if necessary.

Definition 4.4 (Shifting permutations downwards)**.**
We define the function $\cdot$ by the following equations which shifts permutations π into NL_{AS}-expressions, if the application does not match the syntax of NL_{AS}.

$$\begin{aligned}\pi{\cdot}(f\ e_1 \dots e_{ar(f)}) &:= (f\ \pi{\cdot}e_1\ \dots\ \pi{\cdot}e_{ar(f)})\\ \pi{\cdot}(\lambda A.e) &:= \lambda(\pi{\cdot}A).(\pi{\cdot}e)\end{aligned}$$

Definition 4.5 (Simplification of permutation and application)**.**
We assume now that π is over NL_{AS}. Let there be a set ∇ of atom-variable freshness constraints. Then the following simplifications or modifications of permutations or of permutation applications can be performed. These are written as $g \xrightarrow{\nabla} g'$. We also write $g \xrightarrow{\nabla,*} g'$ for its reflexive-transitive closure.

- $(A\ A) \xrightarrow{\nabla} \emptyset$
- $\pi \xrightarrow{\nabla} \pi'$ if $\nabla \vdash \#AtVar(\pi)$ and where π' is a representation of π as a function on atom-variables where π' has at most $|\#AtVar(\pi)|-1$ swappings, and strictly less swappings than π (see Remark 2.3).
- $(\pi \cdot (A\ B)) \cdot A \xrightarrow{\nabla} \pi \cdot B$
- $(\pi \cdot (C\ D)) \cdot A \xrightarrow{\nabla} \pi \cdot A$ if $\nabla \vdash A \neq C \wedge A \neq D$

We say a set of freshness constraints ∇ is *standardized*, if it only consists of elements $A\#B$ and $A\#S$.

Lemma 4.6. *The simplification rules for permutations and applications (Definition 4.5) are sound and strictly reduce the size of permutations.*

Remark 4.7. Also the rewrite rule

$$\pi_1 \cdot \pi_2 \to \pi_2 \cdot \pi_1 \text{ if } \nabla \vdash AtVar(\pi_1)\#AtVar(\pi_2)$$

is sound since every solution ρ of ∇ must instantiate π_1 and π_2 s.t. $dom(\pi_1\rho) \cap dom(\pi_2\rho) = \emptyset$ and that $\pi_i\rho$ are permutations. This and perhaps further sound modifications may be advantageously used in implementations to simplify atom-variable permutations. However, since the rule is non-terminating, we do not add it to the relation $\xrightarrow{\nabla}$. □

Lemma 4.6 implies the following proposition:

Proposition 4.8. *The length n of any derivation $\pi_0 \xrightarrow{\nabla} \pi_1 \xrightarrow{\nabla} \dots \xrightarrow{\nabla} \pi_n$ or $\pi_0{\cdot}A \xrightarrow{\nabla} \pi_1{\cdot}A_1 \xrightarrow{\nabla} \dots \xrightarrow{\nabla} \pi_n{\cdot}A_n$ is polynomially bounded in the size of π_0.*

Definition 4.9 (Simplification of Freshness Constraints)**.** The following rewrite rules

$\xrightarrow{\#}$ allow to simplify (atom-variable) freshness constraints ∇:

- $\{A\#(f\ e_1 \dots e_{ar(f)})\} \uplus \nabla \xrightarrow{\#} \{A\#e_1, \dots, A\#e_{ar(f)}\} \uplus \nabla$
- $\{A\#(\lambda A.e)\} \uplus \nabla \xrightarrow{\#} \nabla$
- $\{A\#e\} \uplus \nabla \xrightarrow{\#} \nabla$ if e does not contain atom-variables
- $\{A\#\lambda B.e\} \uplus \nabla \xrightarrow{\#} \{A\#e\} \uplus \nabla$ if $\nabla \vdash A \neq B$
- $\{A\#(A\ B) \cdot \pi{\cdot}e\} \uplus \nabla \xrightarrow{\#} \{B\#\pi{\cdot}e\} \uplus \nabla$
- $\{A\#(C\ D) \cdot \pi{\cdot}e\} \uplus \nabla \xrightarrow{\#} \{A\#\pi{\cdot}e\} \uplus \nabla$ if $\nabla \vdash A\#\{C, D\}$

By inspecting the semantics of freshness constraints in NL_{AS} the following lemma holds:

Lemma 4.10. *The simplification rules for freshness constraints in Definition 4.9 are sound.*

Lemma 4.11. *The number of possible applications of the simplification rules for freshness constraints in Definition 4.9 is bounded polynomially in the size of ∇.*

4.2. Preparatory Steps of the Algorithm

Definition 4.12 (Flattening)**.** As a preparation for the unification algorithm, all expressions in equations are exhaustively flattened as follows: Every expression $(f\ t_1 \dots t_n)$ is replaced by $(f\ S_1 \dots S_n)$ and the equations $S_1 \doteq t_1, \dots, S_n \doteq t_n$ are added, where S_i are fresh expression-variables. Also $\lambda A.e$ is replaced by $\lambda A.S$ with equation $S \doteq e$. The introduced variables are always fresh ones. We denote the resulting set of equations of flattening an equation eq as $\mathit{flat}(eq)$.

Thus, all expressions in equations in the input to the algorithm can be assumed to be flat, which means that they belong to the language generated by the following grammar: $e_{var} ::= A \mid S \mid \pi{\cdot}A \mid \pi{\cdot}S$ and $e_{flat} ::= e_{var} \mid (f\ e_{var,1} \dots e_{var,ar(f)}) \mid \lambda\pi{\cdot}A.e_{var}$. We will see that flatness of equations is maintained during the algorithm.

4.3. Rules of the Algorithm AVNomUnify

The algorithm searches for unifiers, which are substitutions together with freshness constraints for the introduced variables (see Definition 2.10).

AVNomUnify operates on a tuple (Γ, ∇, θ) over NL_{AS}, where Γ is a set of flattened equations $e_1 \doteq e_2$, and where we assume that $\doteq$ is symmetric, ∇ contains freshness constraints, θ represents the already computed substitution as a list of replacements of the form $X \mapsto e$ (i.e. in so-called triangle-form). Initially θ is empty.

More rigorously, for an input *VANUP* (Γ_0, ∇_0) the algorithm AVNomUnify starts with $(\mathit{flat}(\Gamma_0), \nabla_0, \emptyset)$. If unification does not fail, then the algorithm will output a pair $\langle\nabla, \theta\rangle$ which is a unifier of (Γ_0, ∇_0). The *collecting variant* of AVNomUnify, which explores always both alternatives of (GuessEQ), outputs a set of unifiers.

Threshold Let $\mathit{size}(\Gamma, \nabla)$ be the number of all occurrences of symbols S, A, f, λ in (Γ, ∇), where the atomic freshness constraints $A\#B$ are not counted. In order to control the run of the algorithm, a threshold *theq* has to be chosen, which must be a polynomial in N with $\mathit{theq} \geq 4N^2 * (\mathit{Maxarity} + 2)$ where N is the size of the input, and *Maxarity* is the maximum of 2 and the maximal arity of function symbols in the input. The global condition is that AVNomUnify cannot execute a unification rule, if after the execution and exhaustive simplification the size is larger than the threshold, i.e. if $\mathit{size}(\Gamma, \nabla) \geq \mathit{theq}$.

In the notation of the rules, we use $[e/S]$ as substitution that replaces the expression-variable S by expression e, and analogously, we use $[A/B]$ (or $\pi{\cdot}A/B$) if atom-variable B is replaced by atom-variable A (or suspension $\pi{\cdot}A$, respectively). Similarly, we write $[\pi_1/\pi_2]$ for the replacement of permutation π_2 by permutation π_1 and $[\pi_1 \cdot A_1/\pi_2 \cdot A_2]$ for the replacement of atom-variable suspension $\pi_2 \cdot A_2$ by atom-variable suspension $\pi_1 \cdot A_1$. We will use a notation "|" in the consequence part of rule GuessEQ in order to separate alternatives, to denote disjunctive (i.e. don't know) non-determinism. The only non-deterministic rule that requires exploring all alternatives is rule (GuessEQ) below. All other rules can be applied in any order, where it is not necessary to explore alternatives.

Simplifications and Rewriting

$$\text{(Rew1)}\ \frac{(\Gamma, \nabla, \theta)}{(\Gamma, \nabla, \theta)[\pi'/\pi]}\ \text{if}\ \pi \xrightarrow{\nabla} \pi' \qquad \text{(Rew2)}\ \frac{(\Gamma, \nabla, \theta)}{(\Gamma, \nabla, \theta)[\pi'{\cdot}A'/\pi{\cdot}A]}\ \text{if}\ \pi{\cdot}A \xrightarrow{\nabla} \pi'{\cdot}A'$$

$$\text{(Simp)}\ \frac{(\Gamma, \nabla, \theta)}{(\Gamma, \nabla', \theta)}\ \text{if}\ \nabla \xrightarrow{\#} \nabla'\ \text{(see Definition 4.9)}$$

Standard unification and decomposition rules: We assume that rules (SD3), (SD4a), (SD4b), (SD4d), (SD6), and (VarFail) are also applicable if $\pi = \emptyset$. The substitution $\nabla[\pi{\cdot}B/A]$ in rule (SD4a) replaces freshness constraints $A\#e$ in ∇ by $B\#\pi^{-1} \cdot e[\pi{\cdot}B/A]$, and in rule (SD4b) $B\#e$ in ∇ is replaced by $A\#\pi \cdot e[\pi^{-1}{\cdot}A/B]$.

$$\text{(SD1)}\ \frac{(\Gamma \mathbin{\dot\cup} \{e \doteq e\}, \nabla, \theta)}{(\Gamma, \nabla, \theta)} \qquad \text{(SD2)}\ \frac{(\Gamma \mathbin{\dot\cup} \{\pi \cdot S \doteq e\}, \nabla, \theta)}{(\Gamma \mathbin{\dot\cup} \{S \doteq \pi^{-1} \cdot e\}, \nabla, \theta)}\ \text{if}\ e \notin \mathit{ExVar}(\Gamma, \nabla)$$

$$\text{(SD3)}\ \frac{(\Gamma \mathbin{\dot\cup} \{S \doteq \pi{\cdot}X\}, \nabla, \theta)}{(\Gamma[\pi{\cdot}X/S], \nabla[\pi{\cdot}X/S], \theta \cup \{S \mapsto \pi{\cdot}X\})}\ \text{if}\ S \neq X\text{, and where}\ X\ \text{is an atom- or an expression-variable}$$

$$\text{(SD4a)}\ \frac{(\Gamma \mathbin{\dot\cup} \{A \doteq \pi{\cdot}B\}, \nabla, \theta)}{(\Gamma[\pi{\cdot}B/A], \nabla[\pi{\cdot}B/A], \theta \cup \{A \mapsto \pi{\cdot}B\})}\ \text{if}\ A\ \text{and}\ B\ \text{are different atom-variables, and}\ A\ \text{does not occur in}\ \pi\ \text{nor in a permutation in}\ (\Gamma, \nabla)$$

$$\text{(SD4b)}\ \frac{(\Gamma \mathbin{\dot\cup} \{A \doteq \pi{\cdot}B\}, \nabla, \theta)}{(\Gamma[\pi^{-1}{\cdot}A/B], \nabla[\pi^{-1}{\cdot}A/B], \theta \cup \{B \mapsto \pi^{-1}{\cdot}A\})}\ \text{if}\ A\ \text{and}\ B\ \text{are different atom-variables, and if}\ B\ \text{does not occur in}\ \pi\ \text{nor in a permutation in}\ (\Gamma, \nabla)$$

$$\text{(SD4c)}\ \frac{(\Gamma \mathbin{\dot\cup} \{A \doteq B\}, \nabla, \theta)}{(\Gamma[B/A], \nabla[B/A], \theta \cup \{A \mapsto B\})}\ \text{if}\ A\ \text{and}\ B\ \text{are different atom-variables}$$

(SD4d) $\dfrac{(\Gamma \mathbin{\dot\cup} \{A \doteq (A\ B)\cdot\pi\cdot A\}, \nabla, \theta)}{(\Gamma \mathbin{\dot\cup} \{B \doteq \pi\cdot A\}, \nabla, \theta)}$ if A and B are different atom-variables

(SD4e) $\dfrac{(\Gamma \mathbin{\dot\cup} \{A \doteq (B\ C)\cdot\pi\cdot A\}, \nabla, \theta)}{(\Gamma \mathbin{\dot\cup} \{A \doteq \pi\cdot A\}, \nabla, \theta)}$ if A, B, C are different atom-variables and $\nabla \vdash A \neq B, A \neq C$

(SD5) $\dfrac{(\Gamma \mathbin{\dot\cup} \{(f\ e_1 \ldots e_{ar(f)}) \doteq (f\ e'_1 \ldots e'_{ar(f)})\}, \nabla, \theta)}{(\Gamma \mathbin{\dot\cup} \{e_1 \doteq e'_1, \ldots, e_{ar(f)} \doteq e'_{ar(f)}\}, \nabla, \theta)}$

(SD6) $\dfrac{(\Gamma \mathbin{\dot\cup} \{\lambda\pi\cdot A.e_1 \doteq \lambda\pi\cdot A.e_2\}, \nabla, \theta)}{(\Gamma \mathbin{\dot\cup} \{e_1 \doteq e_2\}, \nabla, \theta)}$

(SD7) $\dfrac{(\Gamma \mathbin{\dot\cup} \{\lambda A.e_1 \doteq \lambda B.e_2\}, \nabla, \theta)}{(\Gamma \mathbin{\dot\cup} \{e_1 \doteq (A\ B)\cdot e_2\}, \nabla \cup \{A\#e_2\}, \theta)}$ if $\nabla \vdash A \neq B$

(SD8) $\dfrac{(\Gamma \mathbin{\dot\cup} \{S \doteq \pi \cdot S\}, \nabla, \theta)}{(\Gamma, \nabla \mathbin{\dot\cup} \{A\#\lambda\pi\cdot A.S \mid A \in AtVar(\pi)\}, \theta)}$ if $\pi \neq Id$

Guessing (Dis-)Equality of Atom-Variables:
The following rule is applied only, if no other rules apply.

(GuessEQ) $\dfrac{(\Gamma, \nabla, \theta)}{(\Gamma[A_1/A_2], \nabla[A_1/A_2], \theta \cup \{A_1 \mapsto A_2\}) \ \mid\ (\Gamma, \nabla \mathbin{\dot\cup} \{A_1\#A_2\}, \theta)}$ if A_1, A_2 occur in Γ or ∇

Main Rules:
In the following rules, *compound terms* denote terms e, where $tops(e)$ is not a variable. The rule (MMS) is reminiscent of Martelli-Montanari-style of first-order unification (Martelli and Montanari, 1982). For a discussion and comparison with other approaches see (Calvès, 2013) and (Schmidt-Schauß et al., 2016).

(MMS) $\dfrac{(\Gamma \mathbin{\dot\cup} \{S \doteq e_1, \ldots, S \doteq e_n\}, \nabla, \theta)}{(\Gamma \mathbin{\dot\cup} \{e_1 \doteq e_2, \ldots, e_1 \doteq e_n\}, \nabla, \theta \cup \{S \mapsto e_1\})}$ if e_i are compound terms, and S does not occur in Γ, and not in e_i for all i

(Output) $\dfrac{(\emptyset, \nabla, \theta)}{\langle \nabla, \theta \rangle}$ if (∇, θ) is solvable using the algorithm AVSolNabla (see Def. 4.14), otherwise Fail.

Note that the size of freshness constraints may be increased by rule (MMS). Note also that elements of ∇ may still contain permutations.
The rule (Output) could also tell the successful choices made by AVSolNabla, but we omit it. Our presentation renders the rule as complete.

Definition 4.13 (Failure Rules of AVNomUnify)**.** The failure rules are:

(ClashFailure) $\dfrac{(\Gamma \uplus \{e_1 \doteq e_2\}, \nabla, \theta)}{\mathit{Fail}}$ if e_1, e_2 are not variables nor suspensions, and $\mathit{tops}(e_1) \neq \mathit{tops}(e_2)$

(VarFail) $\dfrac{(\Gamma \uplus \{\pi{\cdot}A \doteq e\}, \nabla, \theta)}{\mathit{Fail}}$ if $\mathit{tops}(e)$ is a function symbol f or λ

(CycleDetection) $\dfrac{(\Gamma \uplus \{S_1 \doteq e_1, \ldots, S_n \doteq e_n\}, \nabla, \theta)}{\mathit{Fail}}$ if all e_i are neither variables nor suspensions and S_{i+1} occurs in e_i for $i = 1, \ldots, n-1$ and S_1 occurs in e_n

(FreshnessFail) $\dfrac{(\Gamma, \nabla \uplus \{A\#A\}, \theta)}{\mathit{Fail}}$

4.4. Solving Freshness Constraints

We describe the algorithm AVSolNabla that solves problems where $\Gamma = \emptyset$, i.e. freshness constraints together with a substitution θ.

Definition 4.14 (Sub-Algorithm AVSolNabla)**.** The input is a pair (∇, θ) of a set of atom-variable freshness constraints ∇ and a substitution θ, as constructed by the algorithm AVNomUnify.
The following operations and rules are used (with empty Γ): (Simp), (Rew1), (Rew2), and (FreshnessFail), together with the rules (GuessEQFC), (NSubst1), (NSubst2), (NSubst3), and (Sat). The rules can only be applied if the size of ∇, after application of a rule and simplifications, is not greater than *theq*.

(GuessEQFC) $\dfrac{(\nabla, \theta)}{(\nabla[A_1/A_2], \theta[A_1/A_2]) \;\mid\; (\nabla \uplus \{A_1\#A_2\}, \theta)}$ if A_1, A_2 occur in ∇ or in θ, but not in the domain of θ

(NSubst1) $\dfrac{(\nabla \uplus \{A\#\pi \cdot S\}, \theta \cup \{S \mapsto e\})}{(\nabla \uplus \{A\#\pi \cdot e\}, \theta \cup \{S \mapsto e\})}$ if S does not occur in θ

(NSubst2) $\dfrac{(\nabla, \theta \uplus \{A \mapsto B\})}{(\nabla[B/A], \theta[B/A])}$

(NSubst3) $\dfrac{(\nabla, \theta \uplus \{V \mapsto e\})}{(\nabla, \theta)}$ if V (an A or S) does not occur in ∇ nor θ

(Sat) $\dfrac{(\nabla, \theta)}{\text{“satisfiable”}}$ if ∇ is standardized, θ is empty, and (FreshnessFail) is not applicable

This algorithms checks solvability of (∇, θ) by (non-deterministically) executing the rules above until either Fail or “satisfiable” is returned.

The algorithm AVSolNabla treats θ (which is in triangle-form) like a compression by a dag, and decompresses it step by step using (NSubst1), (NSubst2) and (NSubst3).

Rule (GuessEQFC) is a restricted version of rule (GuessEQ) s.t. guessing (dis-)equality of atom-variables prevents a destruction of the structure of θ as a substitution.

4.5. Situations for Guessing (Dis-)Equality in AVNomUnify.

In the following situations, when no other rule is applicable, a pair A, B of atom-variables is guessed as equal or not equal, where the pair can be selected according to the following descriptions. We think that a good proposal for the priority of guessing is the sequence below:

Lemma 4.15. *If the algorithm* AVNomUnify *can only proceed with (GuessEQ), and a guess of (dis-)equality is performed, then at least one of the following situations is valid.*

(1) *There is an equation $\lambda A.e_1 \doteq \lambda B.e_2$, where (dis-)equality of A and B is unknown. Then guess (dis-)equality of A, B.*
(2) *There is an equation $\lambda(\pi{\cdot}A).e_1 \doteq e_2$, where $tops(e_2) = \lambda$, and $(\pi{\cdot}A)$ cannot be further simplified. Then guess (dis-)equality of atom-variables in π using* (8)*.*
(3) *There is an equation $A \doteq \pi{\cdot}A$ in Γ, where π cannot be removed. Then guess (dis-)equality of atom-variables in π using* (8)*.*
(4) *There is an equation $S \doteq \pi{\cdot}S$ in Γ, where π cannot be removed. Then guess (dis-)equality of atom-variables in π using* (8)*.*
(5) *In a freshness constraint $A\#\pi' B$, or $A\#\lambda\pi' B.e$, it is not known whether $A = B$ or $A \neq B$. Then guess (dis-)equality of A, B.*
(6) *An application of a rule would generate a permutation π such that the size of the system exceeds the threshold theq; then guess (dis-)equality of one or more pairs of atom-variables in π, such that π can be shortened afterwards.*
(7) *An instantiation of the form $A \mapsto \pi{\cdot}B$ has to be performed, where A as well as B occur in swappings; then guess (dis-)equality of some pair of atom-variables in π.*
(8) *There is a subexpression $\pi \cdot (C\ D){\cdot}A$ of (Γ, ∇), where (dis-)equality of at least one pair A, C and A, D is yet unknown, then we may guess this for the respective pair.*
(9) *A single application of (MMS) would make the size of the current state greater than the threshold theq. Then apply (GuessEQ) in order to reduce the size.*

4.6. Examples

First we look at the example in Remark 3.10 in (Urban et al., 2003), but see also Example 2.12 for another treatment.

Example 4.16. The problem is to solve $(A\ B) \cdot C \doteq C$, where A, B, C are atom-variables, i.e. $(\Gamma, \nabla, \theta) = (\{(A\ B) \cdot C \doteq C\}, \emptyset, \emptyset)$ at the beginning. The only applicable rule is (GuessEQ), say for A, C:

(1) if $A = C$, then the problem degenerates to $(\{(C\ B) \cdot C \doteq C\}, \{A \mapsto C\})$, which is transformed into $(\{B \doteq C\}, \emptyset, \{A \mapsto C\})$, and then into $(\emptyset, \emptyset, \{A \mapsto C, B \mapsto C\})$ which outputs a unifier $(\{A \mapsto C, B \mapsto C\}, \emptyset)$.
(2) The other alternative is to add $A\#C$, resulting in $(\{(A\ B) \cdot C \doteq C\}, \{A\#C\}, \emptyset)$. Then again we have to apply (GuessEQ), say for B,C:
    (a) If $B = C$, then we get the problem $(\{(A\ C) \cdot C \doteq C\}, \{A\#C\}, \{B \mapsto C\})$ which can first be simplified into $(\{A \doteq C\}, \{A\#C\}, \{B \mapsto C\})$ and then results in $(\emptyset, \{C\#C\}, \{A \doteq C, B \mapsto C\})$ which leads to *Fail*.

(b) If $B\#C$, then we get the problem $(\{(A\ B)\cdot C \doteq C\}, \{A\#C, B\#C\}, \emptyset)$ which first results in $(\{C \doteq C\}, \{A\#C, B\#C\}, \emptyset)$ and then in $(\emptyset, \{A\#C, B\#C\}, \emptyset)$ which is trivially solvable and thus the algorithms outputs $(Id, \{A\#C, B\#C\})$.

Thus, combining all non-deterministic runs, the algorithm outputs two unifiers: $(\emptyset, \{A \mapsto C, B \mapsto C\})$ and $(\{A\#C, B\#C\}, Id)$.

Example 4.17. Consider the example $\lambda x.\lambda y.(S, x) = \lambda y.\lambda x.(x, S)$ (see Example 6.2 for improvements). In our encoding this is the equation: $\lambda A.\lambda B.(S, A) \doteq \lambda B.\lambda A.(A, S)$ where we omit the flattening for better readability. The first step is to apply (GuessEQ):
1) Guess $A \neq B$: The steps are:

$$\begin{array}{llll}
 & \Gamma & \nabla & \theta \\
\rightarrow & \{\lambda A.\lambda B.(S, A) \doteq \lambda B.\lambda A.(A, S)\} & \{A\#B\} & \emptyset \\
\rightarrow & \{\lambda B.(S, A) \doteq (A\ B)\cdot\lambda A.(A, S)\} & \{A\#\lambda A.(B, S), A\#B\} & \emptyset \\
\rightarrow & \{\lambda B.(S, A) \doteq \lambda B.(B, (A\ B)\cdot S)\} & \{A\#B\} & \emptyset \\
\rightarrow & \{(S, A) \doteq (B, (A\ B)\cdot S)\} & \{A\#B\} & \emptyset \\
\rightarrow & \{S \doteq B, A \doteq (A\ B)\cdot S\} & \{A\#B\} & \emptyset \\
\rightarrow & \{S \doteq B, (A\ B)\cdot A \doteq S\} & \{A\#B\} & \emptyset \\
\rightarrow & \{S \doteq B\} & \{A\#B\} & \emptyset \\
\rightarrow & \emptyset & \{A\#B\} & \{S \mapsto B\}
\end{array}$$

2) Guess $A = B$:

$$\begin{array}{llll}
 & \Gamma & \nabla & \theta \\
\rightarrow & \{\lambda A.\lambda A.(S, A) \doteq \lambda A.\lambda A.(A, S)\} & \emptyset & \{B \mapsto A\} \\
\rightarrow & \{\lambda A.(S, A) \doteq \lambda A.(A, S)\} & \emptyset & \{B \mapsto A\} \\
\rightarrow & \{(S, A) \doteq (A, S)\} & \emptyset & \{B \mapsto A\} \\
\rightarrow & \{S \doteq A\} & \emptyset & \{B \mapsto A\} \\
\rightarrow & \emptyset & \emptyset & \{B \mapsto A, S \mapsto A\}
\end{array}$$

Example 4.18. We consider the following *VANUP* as input:

$$(\{(A\ B)\cdot(C\ D)\cdot A \doteq (C\ D)\cdot(A\ B)\cdot C, (A\ B)\cdot C \doteq (C\ D)\cdot A\}, \{B\#C\})$$

We first guess whether $A \neq C$ or $A = C$.
1) Guess $A \neq C$: The steps are:

$$\begin{array}{llll}
 & \Gamma & \nabla & \theta \\
\rightarrow & (A\ B)\cdot(C\ D)\cdot A \doteq (C\ D)\cdot(A\ B)\cdot C, (A\ B)\cdot C \doteq (C\ D)\cdot A & \{A\#C, B\#C\} & \emptyset \\
\rightarrow & (A\ B)\cdot(C\ D)\cdot A \doteq (C\ D)\cdot C, C \doteq (C\ D)\cdot A & \{A\#C, B\#C\} & \emptyset \\
\rightarrow & (A\ B)\cdot(C\ D)\cdot A \doteq D, C \doteq (C\ D)\cdot A & \{A\#C, B\#C\} & \emptyset
\end{array}$$

Now we guess whether $A = D$ or $D \neq A$:
1a) Guess $D \neq A$

$$\begin{array}{llll}
 & \Gamma & \nabla & \theta \\
\rightarrow & \{B \doteq D, C \doteq (C\ D)\cdot A\} & \{A\#C, B\#C, D\#A\} & \emptyset \\
\rightarrow & \{C \doteq (C\ D)\cdot A\} & \{A\#C, D\#C, D\#A\} & \{B \mapsto D\} \\
\rightarrow & \{C \doteq A\} & \{A\#C, D\#C, D\#A\} & \{B \mapsto D\} \\
\rightarrow & \emptyset & \{A\#A, D\#A\} & \{B \mapsto D, C \mapsto A\} \\
\rightarrow & \text{Fail} & &
\end{array}$$

1b) Guess $D = A$

$$\begin{array}{llll}
 & \Gamma & \nabla & \theta \\
\rightarrow & \{(A\ B)\cdot(C\ A)\cdot A \doteq A, C \doteq (C\ A)\cdot A\} & \{A\#C, B\#C\} & \{D \mapsto A\} \\
\rightarrow & \{(A\ B)\cdot C \doteq A, C \doteq C\} & \{A\#C, B\#C\} & \{D \mapsto A\} \\
\rightarrow & \{C \doteq (A\ B)\cdot A\} & \{A\#C, B\#C\} & \{D \mapsto A\} \\
\rightarrow & \{C \doteq B\} & \{A\#C, B\#C\} & \{D \mapsto A\} \\
\rightarrow & \emptyset & \{A\#B, B\#B\} & \{D \mapsto A\} \\
\rightarrow & \text{Fail} & &
\end{array}$$

2) Guess $A = C$: The steps are:

$$\begin{array}{llll}
 & \Gamma & \nabla & \theta \\
\rightarrow & \{(A\ B)\cdot(C\ D)\cdot A \doteq (C\ D)\cdot(A\ B)\cdot C, (A\ B)\cdot C \doteq (C\ D)\cdot A\} & \{B\#C\} & \emptyset \\
\rightarrow & \{(A\ B)\cdot(A\ D)\cdot A \doteq (A\ D)\cdot(A\ B)\cdot A, (A\ B)\cdot A \doteq (A\ D)\cdot A\} & \{B\#A\} & \{C \mapsto A\} \\
\rightarrow & \{(A\ B)\cdot D \doteq (A\ D)\cdot B, B \doteq D\} & \{B\#A\} & \{C \mapsto A\} \\
\rightarrow & \{A \doteq A\} & \{B\#A\} & \{C \mapsto A, D \mapsto B\} \\
\rightarrow & \emptyset & \{B\#A\} & \{C \mapsto A, D \mapsto B\}
\end{array}$$

Example 4.19. A similar example can also be solved by a most general unifier in a different way. This examples also shows that a unifier still may contain swappings for atom-variables, where not every dis-equality is known.

The equation is $(B\ C)\cdot(C\ D)\cdot A \doteq (C\ D)\cdot B$. This equation can be transformed into $A \doteq (C\ D)\cdot(B\ C)\cdot(C\ D)\cdot B$, which results in a most general unifier:
$(\{A \mapsto (C\ D)\cdot(B\ C)\cdot(C\ D) \cdot B\}, \emptyset)$.

5. Invariants, Soundness and Completeness

We show that for a threshold $\mathit{theq} \geq 4N^2 * (\mathit{Maxarity} + 2)$, the unification algorithm does not get stuck.

In the following let $\mathcal{S}_0 := (\Gamma_0, \nabla_0)$ be the initial problem and let the initial size be N. We remind the reader that *Maxarity* is the maximum of 2 and the maximal arity of function symbols that are used in the input problem.

Proposition 5.1. *The algorithm* AVNomUnify *can be successfully executed with a threshold theq* $\geq 4N^2 * (\mathit{Maxarity} + 2)$*, such that it does not lose any solution.*

Proof. In order to prove the claim, we assume an intermediate state $\mathcal{S}$ that satisfies the threshold condition, and a ground solution ρ. We show that the brute force guessing leads to a unifier that covers ρ:
First we apply the rule (GuessEQ) until all (dis)-equalities of atom-variables are guessed (according to ρ). This adds at most N^2 freshness constraints of the form $A\#B$, which are not counted by the threshold-measure.

Since now all pairs of atom-variables A, B with $A \neq B$ have a freshness constraint $A\#B$, we can apply simplifications, which do not increase the size, and implies that simplified states that are successors of $\mathcal{S}$ now satisfy the following: (i) permutations have at most $N-1$ swappings, and (ii) there are no suspensions $\pi{\cdot}A$. Furthermore, all equations between two compound expressions can be transformed: using (SD5) for two expressions with top function symbol, or using (SD6) and (SD7) for $\lambda A.e_1 \doteq \lambda B.e_2$, which do not increase the size, since e_1, e_2 are variables (after the guessing). For the freshness constraints, the simplification and rewrite rules will transform them into the form $A\#B$ or $A\#S$, also not increasing the size. Hence there are at most N^2 freshness constraints. Application of the replacement rules shows that all equations of the form $X \doteq Y$ or $X \doteq \pi{\cdot}Y$, where X, Y are atom- or expression-variables, can be removed from Γ.

Let $\mathcal{S}'$ be the state with these properties, and in addition such that the failure rules do not apply. The remaining equations are of the form $\pi{\cdot}S \doteq e$, where e is a compound expression. In order to estimate the size, we have to analyze the effect of (MMS)-applications between $\mathcal{S}_0$ and $\mathcal{S}'$, where we only look for the number $N(f, \lambda)$ of occurrences of function symbols and λs in Γ: If there are n equations, removed by (MMS)-application, then $n-1$ equations between compound expressions are generated, which are then decomposed. Hence an (MMS)-step with subsequent decomposition strictly decreases $N(f, \lambda)$. Thus the number of equations in $\mathcal{S}'$ is smaller than N.
Thus in $\Gamma(\mathcal{S}')$ there are at most N equations, where every equation is of size at most $2 + \mathit{Maxarity} + 2(N-1)(\mathit{Maxarity}+1) \leq 2N(\mathit{Maxarity}+1)$, hence the size of $\Gamma(\mathcal{S}')$ is at most $2N^2 * (\mathit{Maxarity} + 1)$. An (MMS)-step in $\mathcal{S}'$ on n equations may increase the size by $(n-2) * 2N * \mathit{Maxarity}$, i.e. in general by at most $2N^2 * \mathit{Maxarity}$, summing up to a most $4N^2 * (\mathit{Maxarity} + 1)$.

The size of ∇ where we only count $A\#S$ is at most N^2, hence the total size remains below $N^2 + 4N^2 * (\mathit{Maxarity} + 1)$, which is less than $4N^2 * (\mathit{Maxarity} + 2)$.

Hence every threshold *theq* $\geq 4N^2 * (\mathit{Maxarity} + 2)$ permits the algorithm to proceed. Since we prove below that all rules are sound and complete, we have proved the claim. □

We analyze the invariants of the algorithm AVNomUnify and show that it is sound and runs in non-deterministic polynomial time, and that the determinized version outputs a complete set of solutions.

Lemma 5.2. *The algorithm* AVNOMUNIFY *does not instantiate atom-variables in swappings with suspensions, and it keeps the equation system* Γ *flattened.*

Proof. The rules are constrained such that replacements of atom-variables within swappings are only performed with atom-variables. Also only swappings of the form $(A\ B)$ will be generated. The replacements are such that variables are replaced by variables or suspensions, hence flattening is kept. $\square$

Definition 5.3. We say a unification rule or a unification algorithm by transformations is

sound if for every transformation $\mathcal{S} \to \mathcal{S}'$: if ρ is a solution of $\mathcal{S}'$, then ρ is also a solution of $\mathcal{S}$.

complete if for every state $\mathcal{S}$: if ρ is a solution of $\mathcal{S}$, then there is successor state $\mathcal{S}'$, i.e. with $\mathcal{S} \to \mathcal{S}'$, such that ρ is also a solution of $\mathcal{S}'$.

Theorem 5.4. *The algorithm* AVNOMUNIFY *is sound and complete:*

Proof. Soundness is obvious, since either conditions are added, or equality transformations are performed.
For completeness, let us assume that ρ is a solution before a transformation. We show (i) that there is a possibility of execution such that ρ is a solution of the next state, (ii) that there is no failure rule applicable, and (iii) that the rule (Output) will fire eventually.
The rule (GuessEQ) can guess according to whether $A\rho = B\rho$ or not.
For all other rules with the exception of (SD7) and (SD8), it is easy to see that no solution is lost, since these are either equality transformations or decompositions.

We check completeness of rule (SD7): Let ρ be a solution before the transformation. Then $A\rho \neq B\rho$, and $\lambda(A\rho).e_1\rho \sim_\alpha \lambda(B\rho).e_2\rho$. Using the reasoning in NL_{aS}, which is the same as the calculus in Urban et al. (2003), and using the reasoning there, this is equivalent to the condition $A\rho\#e_2\rho$ and $e_1\rho \sim_\alpha (A\rho\ B\rho){\cdot}e_2\rho$. Hence ρ solves $e_1 \doteq (A\ B){\cdot}e_2$ and the added freshness constraint.

We check completeness of rule (SD8): If there is a solution ρ of the state before, i.e. $S\rho \sim \pi\rho{\cdot}S\rho$, then under the assumption that for all atom-variables A, B occurring in π, we have $A\rho \neq B\rho$, then (in the same way as for nominal unification) alpha-equality can only hold, if for all atom-variables A with $A\rho \neq \pi\rho(A\rho)$, $A\rho$ does no occur free in $S\rho$, which is equivalent to $A\rho\#S\rho$. Hence the added freshness constraints are solved by ρ.

We check completeness of the failure rules: If there is a solution, then rule (Clash-Failure) cannot fire, since the preconditions cannot be satisfied. Rule (VarFail) cannot be applicable, since the solution ρ by definition maps atom-variables to atoms. The rule (CycleDetection) can also not be applicable, since ρ is a solution, which contradicts the existence of such a cycle: consider the sizes a_i of the expression e_i in such a cycle in the solution. This would imply a set of inequations $a_1 > a_2, \ldots, a_n > a_1$, which is impossible.

The final argument is that Proposition 5.1 shows that there is no stuck state, even using the threshold, hence finally the rule (Output) will fire, since ρ is also a solution of the final freshness constraint. $\square$

Theorem 5.5. *The algorithm* AVSOLNABLA *decides solvability of* (∇, θ) *in non-deterministic polynomial time.*

Proof. It is easy to check that the rules of AVSolNabla are sound and complete. We explain why (Sat) is also complete: A set ∇ of freshness constraints in standardized form without any freshness constraint $A\#A$ is satisfiable:. A ground substitution as instance can be constructed by substituting all atom-variables with different atoms, and by substituting fresh atoms for the not substituted expression-variables S.

We exploit that θ is constructed as a substitution, and thus without cycles. Note that it can be seen as a dag used for compression. Note also that a naive expansion may lead to an exponential size of ∇.

The size of ∇ can be kept below the threshold *theq*: At the start of AVSolNabla the size is not greater than *theq*. All used rules with the exception of (NSubst1) do not increase the size of (∇, θ). The strategy for (non-deterministically) solving (∇, θ) is to first apply (GuessEQFC) as often as possible, and to delay (NSubst1). Since θ is a substitution, this is possible by starting with substitution components $A \mapsto e$, where the atom-variables occurring in e are not in $dom(\theta)$. Guessing and applying simplifications permits to transform such a component into the form $A \mapsto B$, and then an application of (NSubst2) removes this component. Since θ is a substitution, and since for components $A \mapsto e$, there are no occurrences of expression-variables in e, this can be done until there are no more components $A \mapsto e$ in θ. Now (dis-) equality guessing can and will be done for all remaining pairs of atom-variables. After this step, and simplifications, all compound expressions are of the form $\lambda A.e_1$ or $(f\ e_1 \ldots e_n)$, where e_i are atom-variables, or expression-variables, or suspensions of expression-variables, and moreover, all permutations π in (∇, θ) are normalized: they have at most $N - 1$ swappings. Also, since guessing is exhaustive, after simplification, ∇ is of size at most N^2. A single step (NSubst1) may increase the size of ∇ to at most $N^2 + 2Maxarity * N$, which is less than *theq*. Using immediate simplifications reduces the size of ∇ again to at most N^2.

The first phase using guessing is polynomial. The second phase of applying (NSubst1) multiple times is also polynomial, since the sequence of instantiations starts with the topmost expression-variables S w.r.t. the substitutions θ, which guarantees polynomial time of the rule application phase. Since the number of re-instantiations is at most $N * theq$, the running time of AVSolNabla is polynomial. $\square$

Theorem 5.6. *The algorithm* AVNomUnify *runs in non-deterministic polynomial time.*
It decides solvability of the input in NP time. The collecting variant computes an at most exponential set of unifiers where every unifier is of at most polynomial size.

Proof. We have to check the time for a single run of the algorithm AVNomUnify. We assume that Γ is flattened.

The proof is structured according to table 1. The method of the proof is as follows: We show that the rules with a X in column 1 are executed at most a polynomial number of times. Then we go to the second column and only look for execution sequences without execution of rules that are already dealt with in column 1. Similar for the other columns.

(1) We look for the rules crossed "X" in column 1: There is no rule that increases the number of atom-variables nor the number of expression-variables. Hence the number of applications of rules (SD3), (SD4a), (SD4b), (SD4c), (MMS) is at most

rule name(s)	A, S-number	*sizeWoPi*	*sizePie*	*sizeSh*
Rew1,Rew2		$=$	$=$	$<$
Simp		$\leq$	$=$	$<$
SD1		$<$		
SD2		$=$	$<$	
SD3,SD4a,SD4b,SD4c	X			
SD4d,SD4e		$=$	$=$	$<$
SD5,SD6,SD7,SD8		$<$		
GuessEQ	X			
MMS	X			

Table 1. Proof structure and the effect of rules

N. The number of pairs of atom-variables is not greater than N^2, since the algorithm does not generate fresh atom-variables. Hence (GuessEQ) can only be applied polynomially often.

(2) For column 2 we use the measure $\mathit{sizeWoPi}(\Gamma)$, which is the size of expressions in Γ ignoring the permutations. The threshold condition shows that $\mathit{sizeWoPi}(\Gamma)$ is at most polynomial in N. We only check sequences of applications of rules in {Rew1, Rew2, Simp, SD1, SD2, SD4d, SD4e, SD5, SD6, SD7, SD8}. and see, that the size $\mathit{sizeWoPi}(\Gamma)$ is not increased during these subsequences, and that SD1, SD5, SD6, SD7, SD8 strictly decrease this measure. Hence the number of applications of them is at most polynomial. The remaining rules are {Rew1, Rew2, Simp, SD2, SD4d, SD4e}.

(3) For column 3 the goal is to treat rule SD2. As a measure we use $\mathit{sizePie}(\Gamma)$, the number of equations in Γ of the form $\pi{\cdot}S \doteq e$. Now we inspect only sequences of applications of rules from {Rew1, Rew2, Simp, SD2, SD4d, SD4e}. Measure *sizePie* is strictly decreased by (SD2) and not increased by the other rules, hence the number of applications of SD2 is polynomial.
The remaining rules are {Rew1, Rew2, Simp, SD4d, SD4e}.

(4) For column 4 the goal is to treat the remaining rules. As a measure we use $\mathit{sizeSh}(\Gamma, \nabla)$, which is defined as the sum of *sizeSh* of all expressions e_1, e_2 occurring in equations $e_1 \doteq e_2$: It is defined as $\mathit{sizeSh}(\pi{\cdot}X) := \mathit{size}(\pi{\cdot}X)$ where X may be A, S, and $\mathit{sizeSh}(\pi{\cdot}\lambda e_1.e_2) := \mathit{sizeSh}(\lambda e_1.e_2) + \mathit{size}(\pi) * \mathit{sizeWoPi}(\lambda e_1.e_2)$, and $\mathit{sizeSh}(\pi{\cdot}(f\ e_1 \dots e_n)) := \mathit{sizeSh}(f\ e_1 \dots e_n) + \mathit{size}(\pi) * \mathit{sizeWoPi}(f\ e_1 \dots e_n)$. For freshness constraints in ∇, it is defined as $\mathit{sizeSh}(\pi{\cdot}A\#e) := 1 + \mathit{size}(\pi) * \mathit{sizeWoPi}(e) + \mathit{sizeSh}(e)$, and $\mathit{sizeSh}(A\#e) := 1 + \mathit{sizeSh}(e)$. The threshold condition and the definition of the measure imply that $\mathit{sizeSh}(\Gamma, \nabla)$ is at most polynomial in N.

We only check sequences of applications of rules in {Rew1, Rew2, Simp, SD4d, SD4e}: All rules application strictly decrease this measure, hence the number of applications in a subsequence is polynomial

The structure of the proof now allows to conclude that the overall number of rule applications is polynomial, since a product of a fixed number of polynomials is itself a polynomial. □

Note that satisfiability of freshness constraints with atom-variables is NP-hard by Theorem 3.1, hence the final test in case $\Gamma = \emptyset$, whether there are solutions at all, cannot be done in (deterministic) polynomial time (unless P = NP).

6. Nominal Unification Variant with Most General Unifier

We describe a variant AVNomUnifyMGU of the unification algorithm that uses the rules that transform the equations into a single representation, where the equation part is solved, a substitution is generated, but the solvability of the resulting freshness constraints is NP-complete. This idea is also used in Example 2.12.

Note that nested representations of permutations π are not generated in Algorithm AVNomUnifyMGU.

We assume that all equations are flattened before the algorithm starts. This means that abstractions and function applications in Γ are only of the form $\lambda\pi{\cdot}A.\pi'{\cdot}S$ and $(f\ \pi_1{\cdot}S_1\ \ldots\ \pi_n{\cdot}S_n)$ where equations of the form $S \doteq e$ for fresh S may be added if necessary. The rules are don't-care non-deterministic, and thus it is not necessary to explore alternatives. For complexity issues, we will assume that sharing is implicitly used, in particular for the substitutions, which is in triangle-form.

We use (SD1), (SD2), (SD3), (SD4c), (SD5), (SD6), (SD8), (MMS) in Section 4.3, the Failure rules, and the following rules:

$$\text{(SD4MGU)}\ \frac{(\Gamma \mathbin{\dot\cup} \{\pi_1{\cdot}A \doteq \pi_2{\cdot}B\}, \nabla, \theta)}{(\Gamma, \nabla \cup \{\pi_1{\cdot}A\#\lambda\pi_2{\cdot}B.\pi_1{\cdot}A\}, \theta)}$$

$$\text{(SD7MGU)}\ \frac{(\Gamma \mathbin{\dot\cup} \{\lambda\pi_1{\cdot}A.e_1 \doteq \lambda\pi_2{\cdot}B.e_2\}, \nabla, \theta)}{\begin{array}{l}(\Gamma \mathbin{\dot\cup} \{e_1 \doteq (A'\ B'){\cdot}e_2\},\\ \quad \nabla \cup \{A'\#\lambda\pi_1{\cdot}A.A', B'\#\lambda\pi_2{\cdot}B.B', A'\#\lambda B'.e_2\}, \theta)\end{array}}\ \text{where } A', B' \text{ are fresh}$$

$$\text{(SD7MGUsimp)}\ \frac{(\Gamma \mathbin{\dot\cup} \{\lambda A.e_1 \doteq \lambda B.e_2\}, \nabla, \theta)}{(\Gamma \mathbin{\dot\cup} \{e_1 \doteq (A\ B){\cdot}e_2\},\ \nabla \cup \{A\#\lambda B.e_2\}, \theta)}$$

The trick here is to encode equality of A and e as a freshness constraint $A\#\lambda e.A$. Using the methods in this paper and since the application of the rules above do not lose any solutions, no guessing is necessary, and since we assume that substitutions are not applied, but shared, we obtain:

Proposition 6.1. *For a solvable input constraint, the algorithm* AVNomUnifyMGU *is sound and complete and generates a most general unifier. The algorithm requires polynomial time and generates a representation of polynomial size, provided sharing is used.*

Note that the rules (SD4MGU), (SD7MGU), and (SD7MGUsimp) could also be applied in the algorithm AVNomUnify.

Example 6.2. We reconsider Example 4.17 and its improvement by AVNomUnifyMGU. The equation is $\lambda A.\lambda B.(S,A) \doteq \lambda B.\lambda A.(A,S)$. We apply (SD7MGUsimp) and obtain:
$(\{\lambda B.(S,A) \doteq (B\ A){\cdot}\lambda A.(A,S)\}, \{A\#\lambda B.\lambda A.(A,S)\})$. The next steps are:
$(\{\lambda B.(S,A) \doteq \lambda B.(B,(B\ A){\cdot}S)\}, \{A\#\lambda B.\lambda A.(A,S)\})$
$(\{(S,A) \doteq (B,(B\ A){\cdot}S)\}, \{A\#\lambda B.\lambda A.(A,S)\})$
$(\{S \doteq B, A \doteq (B\ A){\cdot}S)\}, \{A\#\lambda B.\lambda A.(A,S)\})$
$(\{A \doteq (B\ A){\cdot}B)\}, \{A\#\lambda B.\lambda A.(A,B)\}, \{S \mapsto B\})$
$(\emptyset, \{A\#\lambda B.\lambda A.(A,B)\}, \{S \mapsto B\})$
A clever algorithm could detect that $A\#\lambda B.\lambda A.(A,B)$ is always true, and redundant. Hence we get $\{S \mapsto B\}$ as most general unifier, which is like a union of the two unifiers in Example 4.17.

7. Conclusion

Nominal unification is extended to problems where atom-variables are permitted. It is shown that this extended problem class is NP-complete. A collecting algorithm with lazy (dis-)equality guessing and with an extra threshold parameter to avoid size explosion is constructed, which has a realistic chance to output less unifiers than using a simple guessing strategy. We also showed that a (single) most general unifier is sufficient, however, leaving freshness constraints that have to be solved. An implementation of the algorithm is under way in order to experiment with optimizations and appropriate threshold functions. Future research is to extend the algorithm to more expressive languages that can also incorporate more constructs that occur in programming languages.

Acknowledgements

We thank the reviewers for their valuable remarks and their hints to previous works.

References

Ayala-Rincón, M., Fernández, M., Gabbay, M. J., Rocha-Oliveira, A. C., 2016. Checking overlaps of nominal rewriting rules. Electron. Notes Theor. Comput. Sci. 323, 39–56.

Calvès, C., 2013. Unifying nominal unification. In: van Raamsdonk, F. (Ed.), Proceedings of the 24th International Conference on Rewriting Techniques and Applications (RTA 2013). Vol. 21 of LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 143–157.

Calvès, C., Fernández, M., 2008. A polynomial nominal unification algorithm. Theor. Comput. Sci. 403 (2-3), 285–306.

Cheney, J., 2004a. The complexity of equivariant unification. In: Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP 2004). Vol. 3142 of Lecture Notes in Comput. Sci. Springer-Verlag, pp. 332–344.

Cheney, J., 2004b. Nominal logic programming. Ph.D. thesis, Cornell University, Ithaca, NY.

Cheney, J., 2010. Equivariant unification. J. Automat. Reason. 45 (3), 267–300.

Lakin, M. R., 2011. Constraint solving in non-permutative nominal abstract syntax. Log. Methods Comput. Sci. 7 (3).

Levy, J., Villaret, M., 2008. Nominal unification from a higher-order perspective. In: Voronkov, A. (Ed.), Proceedings of the 19th International Conference on Rewriting Techniques and Applications (RTA 2008). Vol. 5117 of Lecture Notes in Comput. Sci. Springer, pp. 246–260.

Levy, J., Villaret, M., 2010. An efficient nominal unification algorithm. In: Lynch, C. (Ed.), Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA 2010). Vol. 6 of Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl, pp. 209–226.

Martelli, A., Montanari, U., 1982. An efficient unification algorithm. ACM Transactions on Programming Languages and Systems 4 (2), 258–282.

Nipkow, T., Paulson, L. C., Wenzel, M., 2002. Isabelle/HOL — A Proof Assistant for Higher-Order Logic. Vol. 2283 of Lecture Notes in Comput. Sci. Springer.

Pitts, A., Feb. 2016. Nominal techniques. ACM SIGLOG News 3 (1), 57–72.

Pitts, A. M., 2013. Nominal Sets: Names and Symmetry in Computer Science. Cambridge University Press, New York, NY, USA.

Schaefer, T. J., 1978. The complexity of satisfiability problems. In: Lipton, R. J., Burkhard, W. A., Savitch, W. J., Friedman, E. P., Aho, A. V. (Eds.), Proceedings of the 10th Annual ACM Symposium on Theory of Computing (STOC 1978). ACM, pp. 216–226.

Schmidt-Schauß, M., Kutsia, T., Levy, J., Villaret, M., 2016. Nominal unification of higher order expressions with recursive let. In: LOPSTR 2016. Vol. 10184 of Lecture Notes in Comput. Sci. Springer, to be published.

Schmidt-Schauß, M., Sabel, D., 2016. Unification of program expressions with recursive bindings. In: Vidal, G. (Ed.), Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming (PPDP 2016). ACM, New York, NY, USA, pp. 160–173.

Schmidt-Schauß, M., Schütz, M., Sabel, D., 2008. Safety of Nöcker's strictness analysis. J. Funct. Programming 18 (04), 503–551.

Urban, C., Kaliszyk, C., 2012. General bindings and alpha-equivalence in nominal Isabelle. Log. Methods Comput. Sci. 8 (2).

Urban, C., Pitts, A. M., Gabbay, M., 2003. Nominal unification. In: Computer Science Logic, 17th International Workshop, CSL 2003, 12th Annual Conference of the EACSL, and 8th Kurt Gödel Colloquium, KGC 2003, Proceedings. Vol. 2803 of Lecture Notes in Comput. Sci. Springer, pp. 513–527.