Nominal Unification Revisited

Christian Urban
TU Munich, Germany
urbanc@in.tum.de

Nominal unification calculates substitutions that make terms involving binders equal modulo alpha-equivalence. Although nominal unification can be seen as equivalent to Miller's higher-order pattern unification, it has properties, such as the use of first-order terms with names (as opposed to alpha-equivalence classes) and that no new names need to be generated during unification, which set it clearly apart from higher-order pattern unification. The purpose of this paper is to simplify a clunky proof from the original paper on nominal unification and to give an overview over some results about nominal unification.

1 Introduction

The well-known first-order unification algorithm by Robinson [18] calculates substitutions for variables that make terms syntactically equal. For example the terms

$$f\ \langle X, X\rangle =^? f\ \langle Z, g\ \langle Y\rangle\rangle$$

can be made syntactically equal with the substitution $[X :=g\ \langle Y\rangle, Z := g\ \langle Y\rangle]$. In first-order unification we can regard variables as "holes" for which the unification algorithm calculates terms with which the holes need be "filled" by substitution. The filling operation is a simple replacement of terms for variables. However, when binders come into play, this simple picture becomes more complicated: We are no longer interested in syntactic equality since terms like

$$a.\langle a, c\rangle \approx^? b.\langle X, c\rangle \tag{1}$$

should unify, despite the fact that the binders a and b disagree. (Following [19] we write $a.t$ for the term where the name a is bound in t, and $\langle t_1, t_2\rangle$ for a pair of terms.) If we replace X with term b in (1) we obtain the instance

$$a.\langle a, c\rangle \approx b.\langle b, c\rangle \tag{2}$$

which are indeed two alpha-equivalent terms. Therefore in a setting with binders, unification has to be modulo alpha-equivalence.

What is interesting about nominal unification is the fact that it maintains the view from first-order unification of a variable being a "hole" into which a term can be filled. As can be seen, by going from (1) to (2) we are replacing X with the term b without bothering that this b will become bound by the binder. This means the operation of substitution in nominal unification is possibly capturing. A result is that many complications stemming from the fact that binders need to be renamed when a capture-avoiding substitution is pushed under a binder do not apply to nominal unification. Its definition of substitution states that in case of binders

$$\sigma(a.t) = a.\sigma(t)$$

To appear in EPTCS.

holds without any side-condition about a and σ. In order to obtain a unification algorithm that, roughly speaking, preserves alpha-equivalence, nominal unification uses the notion of freshness of a name for a term. This will be written as the judgement $a \# t$. For example in (1) it is ensured that the bound name a on the left-hand side is fresh for the term on the right-hand side, that means it cannot occur free on the right-hand side. In general two abstraction terms will *not* unify, if the binder form one side is free on the other. This condition is sufficient to ensure that unification preserves alpha-equivalence and allows us to regard variables as holes with a simple substitution operation to fill them.

Whenever two abstractions with different binders need to be unified, nominal unification uses the operation of swapping two names to rename the bound names. For example when solving the problem shown in (1), which has two binders whose names disagree, then it will attempt to unify the bodies $\langle a, c\rangle$ and $\langle X, c\rangle$, but first applies the swapping $(a\ b)$ to $\langle X, c\rangle$. While it is easy to see how this swapping should affect the name c (namely not at all), the interesting question is how this swapping should affect the variable X? Since variables are holes for which nothing is known until they are substituted for, the answer taken in nominal unification is to suspend such swapping in front of variables. Several such swapping can potentially accumulate in front of variables. In the example above, this means applying the swapping $(a\ b)$ to $\langle X, c\rangle$ gives the term $\langle (a\ b)\cdot X, c\rangle$, where $(a\ b)$ is suspended in front of X. The substitution $[X := b]$ is then determined by unifying the first components of the two pairs, namely $a \approx^? (a\ b)\cdot X$. We can extract the substitution by applying the swapping to the term a, giving $[X := b]$. This method of suspending swappings in front of variables is related to unification in explicit substitution calculi which use de Bruijn indices and which record explicitly when indices must be raised [7].

Nominal unification gives a similar answer to the problem of deciding when a name is fresh for a term containing variables, say $a \# \langle X, c\rangle$. In this case it will record explicitly that a must be fresh for X. (Since we assume $a \neq c$, it will be that a is fresh for c.) This amounts to the constraint that nothing can be substituted for X that contains a free occurrence of a. Consequently the judgements for freshness #, and also equality $\approx$, depend on an explicit freshness context recording what variables need to be fresh for. We will give the inductive definitions for # and $\approx$ in Section 2. This method of recording extra freshness constraints also allows us to regard the following two terms containing a hole (the variable X)

$$a.X \approx b.X$$

as alpha-equal—namely under the condition that both a and b must be fresh for the variable X. This is defined in terms of judgements of the form

$$\{a \# X, b \# X\} \vdash a.X \approx b.X$$

The reader can easily determine that any substitution for X that satisfies these freshness conditions will produce two alpha-equivalent terms.

Unification problems solved by nominal unification occur frequently in practice. For example typing rules are typically specified as:

$$\frac{(x, \tau) \in \Gamma}{\Gamma \vdash x : \tau}\ var \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}\ app \qquad \frac{(x, \tau_1)::\Gamma \vdash t : \tau_2 \quad x \notin dom\,\Gamma}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2}\ lam$$

Assuming we have the typing judgement $\varnothing \vdash \lambda y.s : \sigma$, we are interested how the *lam*-rule, the only one that unifies, needs to be instantiated in order to derive the premises under which $\lambda y.s$ is typable. This leads to the nominal unification problem

$$\varnothing \vdash \lambda y.s : \sigma \approx^? \Gamma \vdash \lambda x.t : \tau_1 \to \tau_2$$

which can be solved by the substitution $[\Gamma := \varnothing, t := (y\,x) \cdot s, \sigma := \tau_1 \to \tau_2]$ with the requirement that x needs to be fresh for s (in order to stay close to informal practice, we deviate here from the convention of using upper-case letters for variables and lower-case letters for names).

Most closely related to nominal unification is higher-order pattern unification by Miller [14]. Indeed Cheney has shown that higher-order pattern unification problems can be solved by an encoding to nominal unification problems [4]. Levy and Villaret have presented an encoding for the other direction [12]. However, there are crucial differences between both methods of unifying terms with binders. One difference is that nominal unification regards variables as holes for which terms can be substituted in a possibly capturing manner. In contrast, higher-order pattern unification is based on the notion of capture-avoiding substitutions. Hence, variables are not just holes, but always need to come with the parameters, or names, the variable may depend on. For example in order to imitate the behaviour of (1), we have to write $X\ a\ b$, explicitly indicating that the variable X may depend on a and b. If we replace X with an appropriate lambda-abstraction, then the dependency can by "realised" via a beta-reduction. This results in unification problems involving lambda-terms to be unified modulo alpha, beta and eta equivalence. In order to make this kind of unification problems to be decidable, Miller introduced restrictions on the form of the lambda-terms to be unified. With this restriction he obtains unification problems that are not only decidable, but also possess (if solvable) most general solutions.

Another difference between nominal unification and higher-order pattern unification is that the former uses first-order terms, while the latter uses alpha-equivalence classes. This makes the implementation of higher-order pattern unification in a programming language like ML substantially harder than an implementation of nominal unification. One possibility is to implement elements of alpha-equivalence classes as trees and then be very careful in the treatment of names, generating new ones on the fly. Another possibility is to implement them with de-Bruijn indices. Both possibilities, unfortunately, give rise to rather complicated unification algorithms. This complexity is one reason that higher-order unification has up to now not been formalised in a theorem prover, whereas nominal unification has been formalised twice [19, 10]. One concrete example for the higher-order pattern unification algorithm being more complicated than the nominal unification algorithm is the following: higher-order pattern unification has been part of the infrastructure of the Isabelle theorem prover for many years [17]. The problem, unfortunately, with this implementation is that it unifies a slightly enriched term-language (which allows general beta-redexes) and it is not completely understood how eta and beta equality interact in this algorithm. A formalisation of Isabelle's version of higher-order pattern unification and its claims is therefore very much desired, since any bug can potentially compromise the correctness of Isabelle.

In a formalisation it is important to have the simplest possible argument for establishing a property, since this nearly always yields a simple formalisation. In [19] we gave a rather clunky proof for the property that the equivalence relation $\approx$ is transitive. This proof has been slightly simplified in [8]. The main purpose of this paper is to further simplify this proof. The idea behind the simplification is taken from the work of Kumar and Norrish who formalised nominal unification in the HOL4 theorem prover [10], but did not report about their simplification in print. After describing the simpler proof in detail, we sketch the nominal unification algorithm and outline some results obtained about nominal unification.

2 Equality and Freshness

Two central notions in nominal unification are names, which are called *atoms*, and *permutations* of atoms. We assume in this paper that there is a countably infinite set of atoms and represent permutations as finite lists of pairs of atoms. The elements of these lists are called *swappings*. We therefore write permutations

as $(a_1\ b_1)\ (a_2\ b_2)\ \ldots\ (a_n\ b_n)$; the empty list $[]$ stands for the identity permutation. A permutation π *acting* on an atom a is defined as

$$\pi \cdot a \stackrel{\text{def}}{=} a \qquad (a_1\ a_2)::\pi \cdot a \stackrel{\text{def}}{=} \begin{cases} a_2 & \text{if } \pi \cdot a = a_1 \\ a_1 & \text{if } \pi \cdot a = a_2 \\ \pi \cdot a & \text{otherwise} \end{cases}$$

where $(a_1\ a_2)::\pi$ is the composition of a permutation followed by the swapping $(a_1\ a_2)$. The composition of π followed by another permutation π' is given by list-concatenation, written as $\pi' \mathbin{@} \pi$, and the inverse of a permutation is given by list reversal, written as π^{-1}.

The advantage of our representation of permutations-as-lists-of-swappings is that we can easily calculate the composition and the inverse of permutations, which are basic operations in the nominal unification algorithm. However, the list representation does not give unique representatives for permutations (for example $(a\ a) \neq []$). This is is different from the usual representation of permutations given for example in [9]. There permutations are (unique) bijective functions from atoms to atoms. For permutations-as-lists we can define the *disagreement set* between two permutations as the set of atoms given by

$$ds\ \pi\ \pi' \stackrel{def}{=} \{a \mid \pi \cdot a \neq \pi' \cdot a\}$$

and then regard two permutations as equal provided their disagreement set is empty. However, we do not explicitly equate permutations.

The purpose of unification is to make terms equal by substituting terms for variables. The paper [19] defines *nominal terms* with the following grammar:

trm	$::=$	$\langle\rangle$	Units
	$\mid$	$\langle t_1, t_2\rangle$	Pairs
	$\mid$	$f\,t$	Function Symbols
	$\mid$	a	Atoms
	$\mid$	$a.t$	Abstractions
	$\mid$	$\pi \cdot X$	Suspensions

In order to slightly simplify the formal reasoning in the Isabelle/HOL theorem prover, the function symbols only take a single argument (instead of the usual list of arguments). Functions symbols with multiple arguments need to be encoded with pairs. An important point to note is that atoms, written a, b, c, $\ldots$, are distinct from variables, written X, Y, Z, $\ldots$, and only variables can potentially be substituted during nominal unification (a definition of substitution will be given shortly). As mentioned in the Introduction, variables in general need to be considered together with permutations—therefore suspensions are pairs consisting of a permutation and a variable. The reason for this definition is that variables stand for unknown terms, and a permutation applied to a term must be "suspended" in front of all unknowns in order to keep it for the case when any of the unknowns is substituted with a term.

Another important point to note is that, although there are abstraction terms, nominal terms are first-order terms: there is no implicit quotienting modulo renaming of bound names. For example the abstractions $a.t$ and $b.s$ are *not* equal unless $a = b$ and $t = s$. This has the advantage that nominal terms can be implemented as a simple datatype in programming languages such as ML and also in the theorem prover Isabelle/HOL. In [19] a notion of *equality* and *freshness* for nominal terms is defined by two inductive predicates whose rules are shown in Figure 1. This inductive definition uses freshness environments, written ∇, which are sets of atom-and-variable pairs. We often write such environments as $\{a_1 \# X_1, \ldots, a_n \# X_n\}$. Rule ($\approx$-abstraction$_2$) includes the operation of applying a permutation to a nominal term, which can be recursively defined as

$$\frac{}{\nabla \vdash a \# \langle\rangle}(\#\text{-unit}) \qquad \frac{\nabla \vdash a \# t_1 \quad \nabla \vdash a \# t_2}{\nabla \vdash a \# \langle t_1, t_2\rangle}(\#\text{-pair}) \qquad \frac{\nabla \vdash a \# t}{\nabla \vdash a \# f\, t}(\#\text{-function symbol})$$

$$\frac{}{\nabla \vdash a \# a.t}(\#\text{-abstraction}_1) \qquad \frac{\nabla \vdash a \# t \quad a \neq b}{\nabla \vdash a \# b.t}(\#\text{-abstraction}_2)$$

$$\frac{a \neq b}{\nabla \vdash a \# b}(\#\text{-atom}) \qquad \frac{(\pi^{-1} \cdot a, X) \in \nabla}{\nabla \vdash a \# \pi \cdot X}(\#\text{-suspension})$$

$$\frac{}{\nabla \vdash \langle\rangle \approx \langle\rangle}(\approx\text{-unit}) \qquad \frac{\nabla \vdash t_1 \approx t_2 \quad \nabla \vdash s_1 \approx s_2}{\nabla \vdash \langle t_1, s_1\rangle \approx \langle t_2, s_2\rangle}(\approx\text{-pair}) \qquad \frac{\nabla \vdash t_1 \approx t_2}{\nabla \vdash f\, t_1 \approx f\, t_2}(\approx\text{-function symbol})$$

$$\frac{\nabla \vdash t_1 \approx t_2}{\nabla \vdash a.t_1 \approx a.t_2}(\approx\text{-abstraction}_1) \qquad \frac{a \neq b \quad \nabla \vdash a \# t_2 \quad \nabla \vdash t_1 \approx (a\ b) \cdot t_2}{\nabla \vdash a.t_1 \approx b.t_2}(\approx\text{-abstraction}_2)$$

$$\frac{}{\nabla \vdash a \approx a}(\approx\text{-atom}) \qquad \frac{\forall c \in ds\ \pi\ \pi'.\ (c, X) \in \nabla}{\nabla \vdash \pi \cdot X \approx \pi' \cdot X}(\approx\text{-suspension})$$

Figure 1: Inductive definitions for freshness and equality of nominal terms.

$$\begin{array}{rcl}
\pi \cdot (\langle\rangle) & \stackrel{def}{=} & \langle\rangle \\
\pi \cdot (\langle t_1, t_2\rangle) & \stackrel{def}{=} & \langle \pi \cdot t_1, \pi \cdot t_2\rangle \\
\pi \cdot (F\, t) & \stackrel{def}{=} & F\, (\pi \cdot t) \\
\pi \cdot (\pi' \cdot X) & \stackrel{def}{=} & (\pi\ @\ \pi') \cdot X \\
\pi \cdot (a.\, t) & \stackrel{def}{=} & (\pi \cdot a).\, (\pi \cdot t)
\end{array}$$

where the clause for atoms is given in (2). Because we suspend permutations in front of variables (see penultimate clause), it will in general be the case that

$$\pi \cdot t \neq \pi' \cdot t \tag{3}$$

even if the disagreement set of π and π' is empty. Note that permutations acting on abstractions will permute both, the "binder" a and the "body" t.

In order to show the correctness of the nominal unification algorithm in [19], one first needs to establish that $\approx$ is an equivalence relation in the sense of

(i)	$\nabla \vdash t \approx t$	(reflexivity)
(ii)	$\nabla \vdash t_1 \approx t_2$ implies $\nabla \vdash t_2 \approx t_1$	(symmetry)
(iii)	$\nabla \vdash t_1 \approx t_2$ and $\nabla \vdash t_2 \approx t_3$ imply $\nabla \vdash t_1 \approx t_3$	(transitivity)

The first property can be proved by a routine induction over the structure of t. Given the "unsymmetric" formulation of the ($\approx$-abstraction$_2$) rule, the fact that $\approx$ is symmetric is at first glance surprising. Furthermore, a direct proof by induction over the rules seems tricky, since in the ($\approx$-abstraction$_2$) case one needs to infer $\nabla \vdash t_2 \approx (b\ a) \cdot t_1$ from $\nabla \vdash (a\ b) \cdot t_2 \approx t_1$. This needs several supporting lemmas about freshness and equality, but ultimately requires that the transitivity property is proved first. Unfortunately, a direct proof by rule-induction for transitivity seems even more difficult and we did not manage to find one in [19]. Instead we resorted to a clunky induction over the size of terms (since size is preserved

under permutations). To make matters worse, this induction over the size of terms needed to be loaded with two more properties in order to get the induction through. The authors of [8] managed to split up this bulky induction, but still relied on an induction over the size of terms in their transitivity proof.

The authors of [10] managed to do considerably better. They use a clever trick in their formalisation of nominal unification in HOL4 (their proof of equivalence is not shown in the paper). This trick yields a simpler and more direct proof for transitivity, than the ones given in [19, 8]. We shall below adapt the proof by Kumar and Norrish to our setting of (first-order) nominal terms[1]. First we can establish the following property.

Lemma 1. *If* $\nabla \vdash a \# t$ *then also* $\nabla \vdash (\pi \cdot a) \# (\pi \cdot t)$*, and vice versa.*

The proof is by a routine induction on the structure of t and we omit the details. Following [19] we can next attempt to prove that freshness is preserved under equality (Lemma 3 below). However here the trick from [10] already helps to simplify the reasoning. In [10] the notion of *weak equivalence*, written as $\sim$, is defined as follows

$$\frac{}{\langle\rangle \sim \langle\rangle} \qquad \frac{}{a \sim a} \qquad \frac{t \sim t'}{f\,t \sim f\,t'}$$

$$\frac{t_1 \sim s_1 \quad t_2 \sim s_2}{\langle t_1, t_2\rangle \sim \langle s_1, s_2\rangle} \qquad \frac{t \sim t'}{a.t \sim a.t'} \qquad \frac{ds\ \pi\ \pi' = \varnothing}{\pi{\cdot}X \sim \pi'{\cdot}X}$$

This equivalence is said to be weak because two terms can only differ in the permutations that are suspended in front of variables. Moreover, these permutations can only be equal (in the sense that is their disagreement set must be empty). One advantage of this definition is that we can show

$$\pi \cdot t \sim \pi' \cdot t \ \text{ provided } \ ds\ \pi\ \pi' = \varnothing \tag{4}$$

by an easy induction on t. As noted in (3), this property does not hold when formulated with $=$. It is also straightforward to show that

Lemma 2.
(*i*) *If* $\nabla \vdash a \# t_1$ *and* $t_1 \sim t_2$ *then* $\nabla \vdash a \# t_2$.
(*ii*) *If* $\nabla \vdash t_1 \approx t_2$ *and* $t_2 \sim t_3$ *then* $\nabla \vdash t_1 \approx t_3$.

by induction over the relations $\sim$ and $\approx$, respectively. The reason that these inductions go through with ease is that the relation $\sim$ excludes the tricky cases where abstractions differ in their "bound" atoms. Using these two properties together with (4), it is straightforward to establish:

Lemma 3. *If* $\nabla \vdash t_1 \approx t_2$ *and* $\nabla \vdash a \# t_1$ *then* $\nabla \vdash a \# t_2$.

Proof. By induction on the first judgement. The only interesting case is the rule ($\approx$-abstraction$_2$) where we need to establish $\nabla \vdash a \# d.t_2$ from the assumption $(*)$ $\nabla \vdash a \# c.t_1$ with the side-conditions $c \neq d$ and $a \neq d$. Using these side-condition, we can reduce our goal to establishing $\nabla \vdash a \# t_2$. We can also discharge the case where $a = c$, since we know that $\nabla \vdash c \# t_2$ holds by the side-condition of ($\approx$-abstraction$_2$). In case $a \neq c$, we can infer $\nabla \vdash a \# t_1$ from $(*)$, and use the induction hypothesis to conclude with $\nabla \vdash a \# (c\ d) \cdot t_2$. Using Lemma 1 we can infer that $\nabla \vdash (c\ d) \cdot a \# (c\ d)(c\ d) \cdot t_2$ holds, whose left-hand side simplifies to just a (we have that $a \neq d$ and $a \neq c$). For the right-hand side we can prove $(c\ d)(c\ d) \cdot t_2 \sim t_2$, since $ds\ ((c\ d)(c\ d))\ [] = \varnothing$. From this we can conclude this case using Lemma 2(i). □

[1] Their formalisation in HOL4 introduces an indirection by using a quotient construction over nominal terms. This quotient construction does not translate into a simple datatype definition for nominal terms.

The point in this proof is that without the weak equivalence and without Lemma 2, we would need to perform many more "reshuffles" of swappings than the single reference to $\sim$ in the proof above [19]. The next property on the way to establish transitivity proves the equivariance for $\approx$.

Lemma 4. *If* $\nabla \vdash t_1 \approx t_2$ *then* $\nabla \vdash \pi \cdot t_1 \approx \pi \cdot t_2$.

Also with this lemma the induction on $\approx$ does not go through without the help of weak equivalence, because in the ($\approx$-abstraction$_2$)-case we need to show that $\nabla \vdash \pi \cdot t_1 \approx \pi \cdot (a\ b) \cdot t_2$ implies $\nabla \vdash \pi \cdot t_1 \approx (\pi{\cdot}a\ \ \pi{\cdot}b) \cdot \pi \cdot t_2$. While it is easy to show that the right-hand sides are equal, one cannot make use of this fact without a notion of transitivity.

Proof. By induction on $\approx$. The non-trivial case is the rule ($\approx$-abstraction$_2$) where we know $\nabla \vdash \pi \cdot t_1 \approx \pi \cdot (a\ b) \cdot t_2$ by induction hypothesis. We can show that $\pi \mathbin{@} (a\ b) \cdot t_2 \sim (\pi{\cdot}a\ \ \pi{\cdot}b) \mathbin{@} \pi \cdot t_2$ holds (the corresponding disagreement set is empty). Using Lemma 2(ii), we can join both judgements and conclude with $\nabla \vdash \pi \cdot t_1 \approx (\pi{\cdot}a\ \ \pi{\cdot}b) \cdot \pi \cdot t_2$. □

The next lemma relates the freshness and equivalence relations.

Lemma 5. *If* $\forall a{\in}ds\ \pi\ \pi'.\ \nabla \vdash a \# t$ *then* $\nabla \vdash \pi \cdot t \approx \pi' \cdot t$, *and vice versa.*

Proof. By induction on t generalising over the permutation π'. The generalisation is needed in order to get the abstraction case through. □

The crucial lemma in [10], which will allow us to prove the transitivity property by a straightforward rule induction, is the next one. Its proof still needs to analyse several cases, but the reasoning is much simpler than in the proof by induction over the size of terms in [19].

Lemma 6. *If* $\nabla \vdash t_1 \approx t_2$ *and* $\nabla \vdash t_2 \approx \pi \cdot t_2$ *then* $\nabla \vdash t_1 \approx \pi \cdot t_2$.

Proof. By induction on the first $\approx$-judgement with a generalisation over π. The interesting case is ($\approx$-abstraction$_2$). We know $\nabla \vdash b.t_2 \approx (\pi \cdot b).(\pi \cdot t_2)$ and have to prove $\nabla \vdash a.t_1 \approx (\pi \cdot b).(\pi \cdot t_2)$ with $a \neq b$. We have to analyse several cases about a equal equal with $\pi \cdot b$, and b being equal with $\pi \cdot b$. Let us give the details for the case $a \neq \pi \cdot b$ and $b \neq \pi \cdot b$. From the assumption we can infer $(*)$ $\nabla \vdash b \# \pi \cdot t_2$ and $(**)$ $\nabla \vdash t_2 \approx (b\ \pi{\cdot}b) \cdot \pi \cdot t_2$. The side-condition on the first judgement gives us $\nabla \vdash a \# t_2$. We have to show $\nabla \vdash a \# \pi \cdot t_2$ and $\nabla \vdash t_1 \approx (a\ \pi{\cdot}b) \cdot \pi \cdot t_2$. To infer the first fact, we use $\nabla \vdash a \# t_2$ together with $(**)$ and Lemmas 3 and 1. For the second, the induction hypothesis states that for any π we have $\nabla \vdash t_1 \approx \pi \cdot (a\ b) \cdot t_2$ provided $\nabla \vdash (a\ b) \cdot t_2 \approx \pi \cdot (a\ b) \cdot t_2$ holds. We use the induction hypothesis with the permutation $\pi \stackrel{def}{=} (a\ \pi{\cdot}b) \mathbin{@} \pi \mathbin{@} (a\ b)$. This means after simplification the precondition of the IH we need to establish is $(***)$ $\nabla \vdash (a\ b) \cdot t_2 \approx (a\ \pi{\cdot}b) \cdot \pi \cdot t_2$. By Lemma 5 we can transform $(**)$ to $\forall c \in ds\ []\ ((b, \pi{\cdot}b) \mathbin{@} \pi).\ \nabla \vdash c \# t_2$. Similarly with $(***)$. Furthermore we can show that

$$ds\ (a\ b)\ ((a\ \pi{\cdot}b) \mathbin{@} \pi) \subseteq ds\ []\ ((b\ \pi{\cdot}b) \mathbin{@} \pi) \cup \{a, \pi{\cdot}b\}$$

holds. This means it remains to show that $\nabla \vdash a \# t_2$ (which we already inferred above) and $\nabla \vdash \pi{\cdot}b \# t_2$ hold. For the latter, we consider the cases $b = \pi \cdot \pi \cdot b$ and $b \neq \pi \cdot \pi \cdot b$. In the first case we infer $\nabla \vdash \pi{\cdot}b \# t_2$ from $(*)$ using Lemma 1. In the second case we have that $\pi \cdot b \in ds\ []\ ((b\ \pi{\cdot}b) \mathbin{@} \pi)$. So finally we can use the induction hypothesis, which simplified gives us $\nabla \vdash t_1 \approx (a\ \pi{\cdot}b) \cdot \pi \cdot t_2$ as needed. □

With this lemma under our belt, we are finally in the position to prove the transitivity property.

Lemma 7. *If* $\nabla \vdash t_1 \approx t_2$ *and* $\nabla \vdash t_2 \approx t_3$ *then* $\nabla \vdash t_1 \approx t_3$.

Proof. By induction on the first judgement generalising over t_3. We then analyse the possible instances for the second judgement. The non-trivial case is where both judgements are instances of the rule ($\approx$-abstraction$_2$). We have $\nabla \vdash t_1 \approx (a\ b) \cdot t_2$ and $(*)$ $\nabla \vdash t_2 \approx (b\ c) \cdot t_3$ with a, b and c being distinct. We also have $(**)$ $\nabla \vdash a \# t_2$ and $(***)$ $\nabla \vdash b \# t_3$. We have to show $\nabla \vdash a \# t_3$ and $\nabla \vdash t_1 \approx (a\ c) \cdot t_3$. The first fact is a simple consequence of $(*)$ and the Lemmas 1 and 3. For the other case we can use the induction hypothesis to infer our proof obligation, provided we can establish that $\nabla \vdash (a\ b) \cdot t_2 \approx (a\ c) \cdot t_3$ holds. From $(*)$ we have $\nabla \vdash (a\ b) \cdot t_2 \approx (a\ b)(b\ c) \cdot t_3$ using Lemma 4. We also establish that $\nabla \vdash (a\ b)(b\ c) \cdot t_3 \approx (b\ c)(a\ b)(b\ c) \cdot t_3$ holds. By Lemma 5 we have to show that all atoms in the disagreement set are fresh w.r.t. t_3. The disagreement set is equal to $\{a, b\}$. For b the property follows from $(***)$. For a we use $(*)$ and $(**)$. So we can use Lemma 6 to infer $(****)$ $\nabla \vdash (a\ b) \cdot t_2 \approx (b\ c)(a\ b)(b\ c) \cdot t_3$. It remains to show that $\nabla \vdash (a\ b) \cdot t_2 \approx (a\ c) \cdot t_3$ holds. We can do so by using $(****)$ and Lemma 2, and showing that $(b\ c)(a\ b)(b\ c) \cdot t_3 \sim (a\ c) \cdot t_3$ holds. This in turn follows from the fact that the disagreement set $ds\ ((b\ c)(a\ b)(b\ c))\ (a\ c)$ is empty. This concludes the case. □

Once transitivity is proved, reasoning about $\approx$ is rather straightforward. For example symmetry is a simple consequence.

Lemma 8. *If* $\nabla \vdash t_1 \approx t_2$ *then* $\nabla \vdash t_2 \approx t_1$.

Proof. By induction on $\approx$. In the ($\approx$-abstraction$_2$) we have $\nabla \vdash (a\ b) \cdot t_2 \approx t_1$ and need to show $\nabla \vdash t_2 \approx (b\ a) \cdot t_1$. We can do so by inferring $\nabla \vdash (b\ a)(a\ b) \cdot t_2 \approx (b\ a) \cdot t_1$ using Lemma 4. We can also show $\nabla \vdash (b\ a)(a\ b) \cdot t_2 \approx t_2$ using Lemma 5. We can join both facts by transitivity to yield the proof obligation. □

To sum up, the neat trick with using $\sim$ from [10] has allowed us to give a direct, structural, proof for equivalence of $\approx$. The formalisation of this direct proof in Isabelle/HOL is approximately half the size of the formalised proof given in [19].

3 An Algorithm for Nominal Unification

In this section we sketch the algorithm for nominal unification presented in [19]. We refer the reader to that paper for full details.

The purpose of nominal unification algorithm is to calculate substitutions that make terms $\approx$-equal. The substitution operation for nominal terms is defined as follows:

$$\begin{array}{rcl} \sigma(a) & \stackrel{def}{=} & a \\ \sigma(\pi{\cdot}X) & \stackrel{def}{=} & \begin{cases} \pi \cdot \sigma(X) & \text{if } X \in \mathit{dom}\ \sigma \\ \pi{\cdot}X & \text{otherwise} \end{cases} \\ \sigma(a.t) & \stackrel{def}{=} & a.\sigma(t) \\ \sigma(\langle t_1, t_2\rangle) & \stackrel{def}{=} & \langle \sigma(t_1), \sigma(t_2)\rangle \\ \sigma(f\,t) & \stackrel{def}{=} & f\ \sigma(t) \end{array}$$

There are two kinds of problems the nominal unification algorithms solves:

$$t_1 \approx^? t_2 \qquad\qquad a \#^? t$$

The first are called equational problems, the second freshness problems. Their respective interpretation is "can the terms t_1 and t_2 be made equal according to $\approx$?" and "can the atom a be made fresh for t according to #?". A solution for each kind of problems is a pair (∇, σ) consisting of a freshness environment and a substitution such that

$$\nabla \vdash \sigma(t_1) \approx \sigma(t_2) \qquad \nabla \vdash a \# \sigma(t)$$

hold. Note the difference with first-order unification and higher-order pattern unification where a solution consists of a substitution only. An example where nominal unification calculates a non-trivial freshness environment is the equational problem

$$a.X \approx^? b.X$$

which is solved by the solution $(\{a \# X, b \# X\}, [])$. Solutions in nominal unification can be ordered so that the unification algorithm produces always most general solutions. This ordering is defined very similar to the standard ordering in first-order unification.

The nominal unification algorithm in [19] is defined in the usual style of rewriting rules that transform sets of unification problems to simpler ones calculating a substitution and freshness environment on the way. The transformation rule for pairs is

$$\{\langle t_1, t_2\rangle \approx^? \langle s_1, s_2\rangle, \ldots\} \Longrightarrow \{t_1 \approx^? s_1, t_2 \approx^? s_2, \ldots\}$$

There are two rules for abstractions depending on whether or not the binders agree.

$$\{a.t \approx^? a.s, \ldots\} \Longrightarrow \{t \approx^? s, \ldots\}$$
$$\{a.t \approx^? b.s, \ldots\} \Longrightarrow \{t \approx^? (a\ b) \cdot s, a \#^? s, \ldots\}$$

One rule that is also interesting is for unifying two suspensions with the same variable

$$\{\pi \cdot X \approx^? \pi' \cdot X, \ldots\} \Longrightarrow \{a \#^? X \mid a \in ds\ \pi\ \pi'\} \cup \{\ldots\}$$

What is interesting about nominal unification is that it never needs to create fresh names. As can be seen from the abstraction rules, no new name needs to be introduced in order to unify abstractions. It is the case that all atoms in a solution, occur already in the original problem. This has the attractive consequence that nominal unification can dispense with any new-name-generation facility. This makes it easy to implement and reason about the nominal unification algorithm. Clearly, however, the running time of the algorithm using the rules sketched above is exponential in the worst-case, just like the simple-minded first-order unification algorithm without sharing.

4 Applications and Complexity of Nominal Unification

Having designed a new algorithm for unification, it is an obvious step to include it into a logic programming language. This has been studied in the work about αProlog [5] and αKanren [1]. The latter is a system implemented on top of Scheme and is more sophisticated than the former. The point of these variants of Prolog is that they allow one to implement inference rule systems in a very concise and declarative manner. For example the typing rules for simply-typed lambda-terms shown in the Introduction can be implemented in αProlog as follows:

```
type (Gamma, var(X), T) :- member (X,T) Gamma.

type (Gamma, app(M,N), T2) :-
   type (Gamma, M, arrow(T1, T2)), type (Gamma, N, T1).

type (Gamma, lam(x.M), arrow(T1, T2)) / x # Gamma :-
   type ((x,T1)::Gamma, M, T2).

member X X::Tail.
member X Y::Tail :- member X Tail.
```

The shaded boxes show two novel features of αProlog. Abstractions can be written as `x.(-)`; but note that the binder `x` can also occur as a "non-binder" in the body of clauses—just as in the clauses on "paper." The side-condition `x # Gamma` ensures that `x` is not free in any term substituted for `Gamma`. The novel features of αProlog and αKanren can be appreciated when considering that similarly simple implementations in "vanilla" Prolog (which, surprisingly, one can find in textbooks [15]) are *incorrect*, as they give types to untypable lambda-terms. An simple implementation of a first-order theorem prover in αKanren has been given in [16].

When implementing a logic programming language based on nominal unification it becomes important to answer the question about its complexity. Surprisingly, this turned out to be a difficult question. Surprising because nominal unification, like first-order unification, uses simple rewrite rules defined over first-order terms and uses a substitution operation that is a simple replacement of terms for variables. One would hope the techniques from efficient first-order unification algorithms carry over to nominal unification. This is unfortunately only partially the case. Quadratic algorithms for nominal unification were obtained by Calves and Fernandez [3, 2] and independently by Levy and Villaret [13]. These are the best bounds we have for nominal unification so far.

5 Conclusion

Nominal unification was introduced in [19]. It unifies terms involving binders modulo a notion of alpha-equivalence. In this way it is more powerful than first-order unification, but is conceptually much simpler than higher-order pattern unification. Unification algorithms are often critical infrastructure in theorem provers. Therefore it is important to formalise these algorithms in order to ensure correctness. Nominal unification has been formalised twice, once in [19] in Isabelle/HOL and another in [10] in HOL4. The latter formalises a more efficient version of nominal unification based on triangular substitutions. The main purpose of this paper is to simplify the transitivity proof for $\approx$. This in turn simplified the formalisation in Isabelle/HOL.

There have been several fruitful avenues of research that use nominal unification as basic building block. For example the work on αLeanTap [16]. There have also been several works that go beyond the limitation of nominal unification where bound names are restricted to be constant symbols that are not substitutable [11, 6].

References

[1] W. Byrd and D. Friedman. αKanren: A Fresh Name in Nominal Logic Programming Languages. In *Proc. of the 8th Workshop on Scheme and Functional Programming*, pages 79–90, 2007.

[2] C. Calvès. *Complexity and Implementation of Nominal Algorithms*. PhD thesis, King's College London, 2010.

[3] C. Calvès and M. Fernández. A Polynomial Nominal Unification Algorithm. *Theoretical Computer Science*, 403(2-3):285–306, 2008.

[4] J. Cheney. Relating Nominal and Higher-Order Pattern Unification. In *Proc. of the 19th International Workshop on Unification (UNIF)*, pages 104–119, 2005.

[5] J. Cheney and C. Urban. Alpha-Prolog: A Logic Programming Language with Names, Binding, and α-Equivalence. In *Proc. of the 20th International Conference on Logic Programming (ICLP)*, volume 3132 of *LNCS*, pages 269–283, 2004.

[6] R. Clouston and A. Pitts. Nominal Equational Logic. In *Computation, Meaning and Logic, Articles dedicated to Gordon Plotkin*, volume 172 of *ENTCS*, pages 223–257. 2007.

[7] G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Higher-Order Unification via Explicit Substitutions: the Case of Higher-Order Patterns. In *Proc. of the Joint International Conference and Symposium on Logic Programming (JICSLP)*, pages 259–273, 1996.

[8] M. Fernández and J. Gabbay. Nominal Rewriting. *Information and Computation*, 205:917–965, 2007.

[9] B. Huffman and C. Urban. A New Foundation for Nominal Isabelle. In *Proc. of the 1st Interactive Theorem Prover Conference (ITP)*, volume 6172 of *LNCS*, pages 35–50, 2010.

[10] R. Kumar and M. Norrish. (Nominal) Unification by Recursive Descent with Triangular Substitutions. In *Proc. of the 1st Interactive Theorem Prover Conference (ITP)*, volume 6172 of *LNCS*, pages 51–66, 2010.

[11] M. Lakin and A. Pitts. Resolving Inductive Definitions with Binders in Higher-Order Typed Functional Programming. In *Proc. of the 18th European Symposium on Programming (ESOP)*, volume 5502 of *LNCS*, pages 47–61, 2009.

[12] J. Levy and M. Villaret. Nominal Unification from a Higher-Order Perspective. In *Proc. of the 19th International Conference on Rewriting Techniques and Applications (RTA)*, volume 5117 of *LNCS*, pages 246–260, 2008.

[13] J. Levy and M. Villaret. An Efficient Nominal Unification Algorithm. In *Proc. of the 21th International Conference on Rewriting Techniques and Applications (RTA)*, volume 6 of *LIPIcs*, pages 246–260, 2010.

[14] D. Miller. A Logic Programming Language with Lambda-Abstraction, Function Variables, and Simple Unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[15] J. C. Mitchell. *Concepts in Programming Languages*. CUP Press, 2003.

[16] J. Near, W. Byrd, and D. Friedman. αLeanTAP: A Declarative Theorem Prover for First-Order Classical Logic. In *Proc. of the 24th International Conference on Logic Programming (ICLP)*, volume 5366 of *LNCS*, pages 238–252, 2008.

[17] T. Nipkow. Functional Unification of Higher-Order Patterns. In *Proc. of the 8th IEEE Symposium of Logic in Computer Science (LICS)*, pages 64–74, 1993.

[18] J. A. Robinson. A Machine Oriented Logic Based on the Resolution Priciple. *JACM*, 12(1):23–41, 1965.

[19] C. Urban, A.M. Pitts, and M.J. Gabbay. Nominal Unification. *Theoretical Computer Science*, 323(1-3):473–497, 2004.