Nominal Unification

Christian Urban   Andrew Pitts   Murdoch Gabbay

University of Cambridge, Cambridge, UK.

Abstract. We present a generalisation of first-order unification to the practically important case of equations between terms involving *binding operations*. A substitution of terms for variables solves such an equation if it makes the equated terms α-*equivalent*, i.e. equal up to renaming bound names. For the applications we have in mind, we must consider the simple, textual form of substitution in which names occurring in terms may be captured within the scope of binders upon substitution. We are able to take a 'nominal' approach to binding in which bound entities are explicitly named (rather than using nameless, de Bruijn-style representations) and yet get a version of this form of substitution that respects α-equivalence and possesses good algorithmic properties. We achieve this by adapting an existing idea and introducing a key new idea. The existing idea is terms involving explicit substitutions of names for names, except that here we only use *explicit permutations* (bijective substitutions). The key new idea is that the unification algorithm should solve not only equational problems, but also problems about the *freshness* of names for terms. There is a simple generalisation of the classical first-order unification algorithm to this setting which retains the latter's pleasant properties: unification problems involving α-equivalence and freshness are decidable; and solvable problems possess most general solutions.

1 Introduction

Decidability of unification for equations between first-order terms and algorithms for computing most general unifiers form a fundamental tool of computational logic with many applications to programming languages and computer-aided reasoning. However, very many potential applications fall outside the scope of first-order unification, because they involve term languages with binding operations where at the very least we do not wish to distinguish terms differing up to the renaming of bound names.

There is a large body of work studying languages with binders through the use of various λ-calculi as term representation languages, leading to *higher-order unification* algorithms for solving equations between λ-terms modulo $\alpha\beta\eta$-equivalence. However, higher-order unification is technically complicated without being completely satisfactory from a pragmatic point of view. The reason lies in the difference between substitution for first-order terms and for λ-terms. The former is a simple operation of textual replacement (sometimes called *grafting* [7], or *context substitution* [12, Sect. 2.1]), whereas the latter also involves renamings to avoid capture. Capture-avoidance ensures that substitution respects α-equivalence, but it complicates higher-order unification algorithms. Furthermore it is the simple textual form of substitution rather than the more complicated capture-avoiding form which occurs in many informal applications of 'unification modulo α-equivalence'. For example, consider the following schematic rule which might form part of the inductive definition of a binary evaluation relation $\Downarrow$ for the expressions of an imaginary functional programming language:

$$\frac{\mathtt{app}(\mathtt{fn}\, a.Y, X) \;\Downarrow\; V}{\mathtt{let}\, a = X \,\mathtt{in}\, Y \;\Downarrow\; V}. \tag{1}$$

Here X, Y and V are metavariables standing for unknown programming language expressions. The binders $\mathtt{fn}\, a.(-)$ and $\mathtt{let}\, a = X \,\mathtt{in}\,(-)$ may very well capture free occurrences of a when we instantiate the schematic rule by replacing Y with an expression. For instance, using the rule scheme in a bottom-up search for a proof of

$$\mathtt{let}\, a = 1 \,\mathtt{in}\, a \;\Downarrow\; 1 \tag{2}$$

we would use a substitution that involves capture, namely $[X := 1, Y := a, V := 1]$, to unify the goal with the conclusion of (1)—generating the new goal $\mathtt{app}(\mathtt{fn}\, a.a, 1) \Downarrow 1$ from the hypothesis of the rule. The problem with this is that in informal practice we usually identify terms up to α-equivalence, whereas textual substitution does not respect α-equivalence. For example, up to α-equivalence, the goal

$$\mathtt{let}\, b = 1 \,\mathtt{in}\, b \;\Downarrow\; 1 \tag{3}$$

is the same as (2). We might think (erroneously!) that the conclusion of (1) is the same as $\mathtt{let}\, b = X \,\mathtt{in}\, Y \Downarrow V$ without changing the rule's hypothesis—after all, if we are trying to make α-equivalence disappear into the infrastructure, then we must be able to replace any *part* of what we have with an equivalent part. So we might be tempted to unify the conclusion with (3) via the textual substitution $[X := 1, Y := b, V := 1]$, and then apply this substitution to the hypothesis to obtain a wrong goal, $\mathtt{app}(\mathtt{fn}\, a.b, 1) \Downarrow 1$. Using λ-calculus and higher-order unification saves us from such sloppy thinking, but at the expense of having to make explicit the dependence of metavariables on bindable names via the use of function variables and application. For example, (1) would be replaced by

$$\frac{\mathtt{app}\,(\mathtt{fn}\,\lambda a.F\,a)\,X \;\Downarrow\; V}{\mathtt{let}\,X\,(\lambda a.F\,a) \;\Downarrow\; V} \quad \text{or, modulo } \eta\text{-equivalence,} \quad \frac{\mathtt{app}\,(\mathtt{fn}\,F)\,X \;\Downarrow\; V}{\mathtt{let}\,X\,F \;\Downarrow\; V}. \tag{4}$$

Now goal (3) becomes $\mathtt{let}\,1\,\lambda b.b \;\Downarrow\; 1$ and there is no problem unifying it with the conclusion of (4) via a capture-avoiding substitution of 1 for X, $\lambda c.c$ for F and 1 for V.

This is all very fine, but the situation is not as pleasant as for first-order terms: higher-order unification problems can be undecidable, decidable but lack most general unifiers, or have such unifiers only by imposing some restrictions [20]; see [6] for a survey of higher-order unification. We started out wanting to compute with binders modulo α-equivalence, and somehow the process of making possibly-capturing substitution respectable has led to function variables, application, capture-avoiding substitution and $\beta\eta$-equivalence. Does it have to be so? No!

For one thing, several authors have already noted that one can make sense of possibly-capturing substitution modulo α-equivalence by using *explicit substitutions* in the term representation language: see [7, 13, 15, 26]. Compared with those works, we make a number of simplifications. First, we find that we do *not* need to use function variables, application or $\beta\eta$-equivalence in our representation language—leaving just binders and α-equivalence. Secondly, instead of using explicit substitutions of names for names, we use *explicit permutations* of names. The idea of using name-permutations, and in particular name-swappings, when dealing with α-conversion dates back to [10] and there is growing evidence of its usefulness (see [3, 4], for example). When a name substitution is actually a permutation, the function it induces from terms to terms is a bijection; this bijectivity gives the operation of permuting names very good logical properties compared with name substitution. Consider for example the α-equivalent terms $\mathtt{fn}\,a.b$ and $\mathtt{fn}\,c.b$, where a, b and c are distinct. If we apply the substitution $[b{\mapsto}a]$ (renaming all free occurrences of b to be a) to them we get $\mathtt{fn}\,a.a$ and $\mathtt{fn}\,c.a$, which are no longer α-equivalent. Thus renaming substitutions do not respect α-equivalence in general, and any unification algorithm using them needs to take extra precautions to not inadvertently change the intended meaning of terms. The traditional solution for this problem is to introduce a more complicated form of renaming substitution that avoids capture of names by binders. In contrast, the simple operation of name-permutation respects α-equivalence; for example, applying the name-permutation $(a\,b)$ that swaps all occurrences of a and b (be they free, bound or binding) to the terms above gives $\mathtt{fn}\,b.a$ and $\mathtt{fn}\,c.a$, which are still α-equivalent. We exploit such good properties of name-permutations to give a conceptually simple unification algorithm.

In addition to the use of explicit name-permutations, we also compute symbolically with predicates expressing *freshness* of names for terms. This seems to be the key novelty of our approach. Although it arises naturally from the work reported in [11, 23], it is easy to see directly why there is a need for computing with freshness, given that we take a 'nominal' approach to binders. (In other words we stick with concrete versions of binding and α-equivalence in which the bound entity is named explicitly, rather than using de Bruijn-style representations, as for example in [7, 26].) A basic instance of our generalised form of α-equivalence identifies $\mathtt{fn}\,a.X$ with $\mathtt{fn}\,b.(a\,b){\cdot}X$ provided b *is fresh for* X, where the subterm $(a\,b){\cdot}X$ indicates an explicit permutation—namely the swapping of a and b—waiting to be applied to X. We write 'b is fresh for X' symbolically as $b \mathrel{\#} X$; the intended meaning of this relation is that b does not occur free in any (ground) term that may be substituted for X. If we know more about X we may be able to eliminate the explicit permutation in $(a\,b){\cdot}X$; for example, if we knew that $a \mathrel{\#} X$ holds as well as $b \mathrel{\#} X$, then $(a\,b){\cdot}X$ can be replaced by X. It should already be clear from these simple examples that in our setting the appropriate notion of term-equality is not a bare equation, $t \approx t'$, but rather a hypothetical judgement of the form

$$\nabla \vdash t \approx t' \tag{5}$$

where ∇ is a *freshness environment*—a finite set $\{a_1 \# X_1, \ldots, a_n \# X_n\}$ of freshness assumptions. For example $\{a \# X, b \# X\} \vdash \mathtt{fn}\,a.X \approx \mathtt{fn}\,b.X$ is a valid judgement of our *nominal equational logic*. Similarly, judgements about freshness itself will take the form

$$\nabla \vdash a \mathrel{\#} t\ . \tag{6}$$

To summarise: We will represent languages involving binders using the usual notion of first-order terms over a many-sorted signature. These give us terms with: distinguished constants naming bindable entities, that we call *atoms*; terms $a.t$ expressing a generic form of *binding* of an atom a in a term t; and terms $\pi{\cdot}X$ representing an explicit *permutation* of atoms π waiting to be applied to whatever term is substituted for the variable X. Section 2 presents this term-language together with a syntax-directed inductive definition of the provable judgements of the form (5) and (6) which for *ground* terms (i.e. ones with no variables) agrees with the usual notions of α-equivalence and 'not a free variable of'. However, on open terms our judgements *differ* from these standard notions and appear to be an extension that has not yet been studied in the literature (including [11, 24, 23]). Section 3 considers unification in this setting. Solving equalities between abstractions $a.t \approx ?\ a'.t'$ entails solving both equalities $t \approx ?\ (a\,a')\boldsymbol{\cdot} t'$ and freshness problems $a \; \#? \; t'$. Therefore our general form of *nominal unification problem* is a finite collection of individual equality and freshness problems. Such a problem P is solved by providing not only a substitution σ (of terms for variables), but also a freshness environment ∇ (as above), which together have the property that $\nabla \vdash \sigma(t) \approx \sigma(t')$ and $\nabla \vdash a \,\#\, \sigma(t'')$ hold for each individual equality $t \approx ?\ t'$ and freshness $a \; \#? \; t''$ in the problem P. Our main result with respect to unification is that *solvability is decidable and that solvable problems possess most general solutions* (for a reasonably obvious notion of 'most general'). The proof is via a unification algorithm that is very similar to the first-order algorithm given in the now-common transformational style [18]. (See [17, Sect. 2.6] or [1, Sect. 4.6] for expositions of this.) Section 4 considers the relationship of our version of 'unification modulo α-equivalence' to existing approaches. Section 5 assesses what has been achieved and the prospects for applications. To appreciate the kind of problem that nominal unification solves, you might like to try the following quiz about the λ-calculus [2] before we apply our algorithm to solve it at the end of Section 3.

Quiz *Assuming that a and b are distinct variables*, is it possible to find λ-terms $M_1, \ldots, M_7$ that make the following pairs of terms α-equivalent?

1. $\lambda a.\lambda b.(M_1\, b)$ and $\lambda b.\lambda a.(a\, M_1)$
2. $\lambda a.\lambda b.(M_2\, b)$ and $\lambda b.\lambda a.(a\, M_3)$
3. $\lambda a.\lambda b.(b\, M_4)$ and $\lambda b.\lambda a.(a\, M_5)$
4. $\lambda a.\lambda b.(b\, M_6)$ and $\lambda a.\lambda a.(a\, M_7)$.

If it is possible to find a solution for any of these four problems, can you describe what all possible solutions for that problem are like?

Answers: see Example 2.

2 Nominal equational logic

We take a concrete approach to the syntax of binders in which bound entities are explicitly named. Furthermore we separate the names of bound entities from the names of variables, which is inspired for example by the π-calculus [21], in which the restriction operator binds channel names and these are quite different from names of unknown processes. Names of bound entities will be called *atoms*. This is partly for historical reasons (stemming from the work by the second two authors [11]) and partly to indicate that the internal structure of such names is irrelevant to us: all we care about is their identity (i.e. whether or not one atom is the same as another) and that the supply of atoms is inexhaustible.

Although there are several general frameworks in the literature for specifying languages with binders, not all of them meet the requirements mentioned in the previous paragraph. Use of the simply typed λ-calculus for this purpose is common; but as discussed in the Introduction, it leads to a problematic unification theory. Among *first-order* frameworks, Plotkin's notion of *binding signature* [25, 9], being unsorted, equates names used in binding with names of variables standing for unknown terms; so it is not sufficiently general for us. A first-order framework that does meet our requirements is the notion of *nominal algebras* in [16]. The *nominal signatures* that we use in this paper are a mild (but practically useful) generalisation of nominal algebras in which name-abstraction and pairing can be mixed freely in arities (rather than insisting as in [16] that the argument sort of a function symbol be normalised to a tuple of abstractions).

Definition 1. *A* ***nominal signature*** *is specified by: a set of* ***sorts of atoms*** *(typical symbol ν); a disjoint set of* ***sorts of data*** *(typical symbol δ); and a set of* ***function symbols*** *(typical symbol f), each of which has an* ***arity*** *of the form $\tau \to \delta$. Here τ ranges over (compound)* ***sorts*** *given by the grammar $\tau ::= \nu \mid \delta \mid 1 \mid \tau \times \tau \mid \langle\nu\rangle\tau$. Terms of sort $\langle\nu\rangle\tau$ are binding abstractions of atoms of sort ν over terms of sort τ. We will explain the syntax and properties of such terms in a moment.*

Example 1. Here is a nominal signature for expressions in a small fragment of ML:

sort of atoms:	vid	function symbols:	$\mathtt{vr} : \mathit{vid} \to \mathit{exp}$
sort of data:	exp		$\mathtt{app} : \mathit{exp} \times \mathit{exp} \to \mathit{exp}$
			$\mathtt{fn} : \langle\mathit{vid}\rangle\mathit{exp} \to \mathit{exp}$
			$\mathtt{lv} : \mathit{exp} \times \langle\mathit{vid}\rangle\mathit{exp} \to \mathit{exp}$
			$\mathtt{lf} : \langle\mathit{vid}\rangle((\langle\mathit{vid}\rangle\mathit{exp}) \times \mathit{exp}) \to \mathit{exp}$.

The function symbol `vr` constructs terms of sort exp representing value identifiers (named by atoms of sort vid); `app` constructs application expressions from pairs of expressions; `fn`, `lv` and `lf` construct terms representing respectively function abstractions (`fn x => e`), local value declarations (`let val x = e1 in e2 end`) and local recursive function declarations (`let fun f x = e1 in e2 end`). The arities of the function symbols specify which are binders and in which way their arguments are bound. This kind of specification of binding scopes is of course a feature of *higher-order abstract syntax* [22], using function types $\nu \to \tau$ in simply typed λ-calculus where we use abstraction sorts $\langle\nu\rangle\tau$. We shall see that the latter have much more elementary (indeed, first-order) properties compared with the former. To make this point clear we deliberately use a first-order syntax for terms, and *not* higher-order abstract syntax, although we often refer to abstractions, binders and free atoms by analogy with the λ-calculus.

Definition 2. *Given a nominal signature, we assume that there are countably infinite and pairwise disjoint sets of* ***atoms*** *(typical symbol a) for each sort of atoms ν, and* ***variables*** *(typical symbol X) for each sort τ. The* ***terms*** *over a nominal signature and their sorts are inductively defined as follows, where we write $t : \tau$ to indicate that a term t has sort τ.*

Unit value $\langle\rangle : 1$.
Pairs $\langle t_1, t_2\rangle : \tau_1 \times \tau_2$, *if $t_1 : \tau_1$ and $t_2 : \tau_2$.*
Data $f\,t : \delta$, *if f is a function symbol of arity $\tau \to \delta$ and $t : \tau$.*
Atoms $a : \nu$, *if a is an atom of sort ν.*
Atom-abstraction $a.t : \langle\nu\rangle\tau$, *if a is an atom of sort ν and $t : \tau$.*
Suspension $\pi{\cdot}X : \tau$, *if $\pi = (a_1\,b_1)(a_2\,b_2)\cdots(a_n\,b_n)$ is a finite list whose elements $(a_i\,b_i)$ are pairs of atoms, with a_i and b_i of the same sort, and X is a variable of sort τ. In the case that π is the empty list $[]$, we just write X for $\pi{\cdot}X$.*

Recall that every finite permutation can be expressed as a composition of swappings $(a_i\,b_i)$; the list π of pairs of atoms occurring in a suspension term $\pi{\cdot}X$ specifies a finite permutation of atoms waiting to be applied once we know more about the variable

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

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

Fig. 1. Permutation action on terms, $\pi\cdot t$.

X (by substituting for it, for example). We represent finite permutations in this way because it is really the operation of swapping which plays a fundamental role in the theory. Since, semantically speaking (see Remark 1 below about semantics), swapping commutes with all term-forming operations, we can normalise terms involving an explicit swapping operation by pushing the swap in as far as it will go, until it reaches a variable (cf. Fig. 1); the terms in Def. 2 are all normalised in this way, with explicit swappings 'piled up' in front of variables giving what we have called *suspensions*.

We wish to give a definition of α-equivalence for terms over a nominal signature that is respected by substitution of terms for variables, even though the latter may involve capture of atoms by binders. To do so we will need to make use of an auxiliary relation of *freshness* between atoms and terms, whose intended meaning is that the atom does not occur free in any substitution instance of the term. As discussed in the Introduction, our judgements about term equivalence ($t \approx t'$) need to contain hypotheses about the freshness of atoms with respect to variables ($a \# X$); and the same goes for our judgements about freshness itself ($a \# t$). Figure 2 gives a syntax-directed inductive definition of equivalence and freshness using judgements of the form

$$\nabla \vdash t \approx t' \quad \text{and} \quad \nabla \vdash a \# t$$

where t and t' are terms of the same sort over a given nominal signature, a is an atom, and the **freshness environment** ∇ is a finite set of **freshness constraints** $a \# X$, each specified by an atom and a variable. Rules ($\approx$-abstraction-2), ($\approx$-suspension) and ($\#$-suspension) in Fig. 2 make use of the following definitions.

Definition 3. *Recall from Def. 2 that we specify* ***finite permutations of atoms*** *by finite lists $(a_1\, b_1,)(a_2\, b_2)\cdots(a_n\, b_n)$ representing the composition of finitely many swappings $(a_i\, b_i)$, with a_i and b_i of the same sort. Since we will apply permutations to terms on the left, the order of the composition is from right to left. So with this representation, the composition of a permutation π followed by a swap $(a\, b)$ is given by list-cons, written $(a\, b) :: \pi$; the composition of π followed by another permutation π' is given by list-concatenation, written as $\pi'@\pi$; the* ***identity permutation*** *is given by the empty list $[]$; and the* ***inverse*** *of a permutation is given by list reversal, written as π^{-1}. The* ***permutation action****, $\pi\cdot t$, of a finite permutation of atoms π on a term t is defined as in Fig. 1; it pushes the list π into the structure of the term t until it 'piles up' in front of suspensions (applying the actual permutation that π represents to atoms that it meets on the way). The* ***disagreement set*** *of two permutations π and π' (used in rule ($\approx$-*

$$\frac{}{\nabla \vdash \langle\rangle \approx \langle\rangle}\,(\approx\text{-unit}) \quad \frac{\nabla \vdash t_1 \approx t_1' \quad \nabla \vdash t_2 \approx t_2'}{\nabla \vdash \langle t_1, t_2\rangle \approx \langle t_1', t_2'\rangle}\,(\approx\text{-pair}) \quad \frac{\nabla \vdash t \approx t'}{\nabla \vdash f\,t \approx f\,t'}\,(\approx\text{-function symbol})$$

$$\frac{\nabla \vdash t \approx t'}{\nabla \vdash a.t \approx a.t'}\,(\approx\text{-abstraction-1}) \quad \frac{a \neq a' \quad \nabla \vdash t \approx (a\,a')\boldsymbol{\cdot} t' \quad \nabla \vdash a \# t'}{\nabla \vdash a.t \approx a'.t'}\,(\approx\text{-abstraction-2})$$

$$\frac{}{\nabla \vdash a \approx a}\,(\approx\text{-atom}) \quad \frac{(a \# X) \in \nabla \text{ for all } a \in ds(\pi, \pi')}{\nabla \vdash \pi\boldsymbol{\cdot} X \approx \pi'\boldsymbol{\cdot} X}\,(\approx\text{-suspension})$$

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

$$\frac{}{\nabla \vdash a \# a.t}\,(\#\text{-abstraction-1}) \quad \frac{a \neq a' \quad \nabla \vdash a \# t}{\nabla \vdash a \# a'.t}\,(\#\text{-abstraction-2})$$

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

Fig. 2. Inductive definition of $\approx$ and $\#$.

suspension) in Fig. 2) is defined by

$$ds(\pi, \pi') \stackrel{\mathit{def}}{=} \{a \mid \pi\boldsymbol{\cdot} a \neq \pi'\boldsymbol{\cdot} a\} \ . \tag{7}$$

Note that every disagreement set of the lists π and π' is a subset of the *finite* set of atoms occurring in either of the lists, because if a does not occur in those lists, then from Fig. 1 we get $\pi\boldsymbol{\cdot} a = a = \pi'\boldsymbol{\cdot} a$. To illustrate the use of disagreement sets, consider

$$\{a \# X, c \# X\} \vdash (a\,c)(a\,b)\boldsymbol{\cdot} X \approx (b\,c)\boldsymbol{\cdot} X$$

which holds by ($\approx$-suspension), since the disagreement set of $(a\,c)(a\,b)$ and $(b\,c)$ is $\{a,c\}$.

Lemma 1. $\nabla \vdash - \approx -$ *is an equivalence relation; it is preserved by all of the term-forming operations in Def. 2; and it respects the freshness relation (i.e. if* $\nabla \vdash a \# t$ *and* $\nabla \vdash t \approx t'$*, then* $\nabla \vdash a \# t'$*). Both* $\approx$ *and* $\#$ *are preserved by the permutation action given in Fig. 1 in the following sense: if* $\nabla \vdash t \approx t'$*, then* $\nabla \vdash \pi\boldsymbol{\cdot} t \approx \pi\boldsymbol{\cdot} t'$*; and if* $\nabla \vdash a \# t$*, then* $\nabla \vdash \pi\boldsymbol{\cdot} a \# \pi\boldsymbol{\cdot} t$.
Proof. Although reasoning about $\approx$ and $\#$ is rather pleasant once the above facts are proved, establishing them first is rather tricky—mainly because of the large number of cases, but also because the facts in the lemma are inter-dependent; in addition some further properties of the permutation action and disagreement sets need to be established first (statements omitted).[1] □

The main reason for using suspensions in the syntax of terms is to enable a definition of *substitution of terms for variables* which allows capture of free atoms by atom-abstractions while still respecting α-equivalence. The following lemma establishes this. First we give some terminology and notation for term-substitution.

[1] A machine-checked proof of all the results using the theorem prover Isabelle can be found at `http://www.cl.cam.ac.uk/~cu200/Unification`.

Definition 4. *A **substitution** σ is a sort-respecting function from variables to terms with the property that $\sigma(X) = X$ for all but finitely many variables X. We shall write* $\boldsymbol{dom(\sigma)}$ *for the finite set of variables X satisfying $\sigma(X) \neq X$. If $dom(\sigma)$ consists of variables $X_1, \ldots, X_n$ and $\sigma(X_i) = t_i$ for $i = 1..n$, we shall sometimes write σ as*

$$\sigma = [X_1 := t_1, \ldots, X_n := t_n]. \tag{8}$$

*We write $\sigma(t)$ for the result of **applying a substitution** σ to a term t; this is the term obtained from t by replacing each suspension $\pi{\cdot}X$ in t (as X ranges over $dom(\sigma)$) by the term $\pi{\cdot}\sigma(X)$ got by letting π act on the term $\sigma(X)$ using the definition in Fig. 1. For example, if $\sigma = [X := \langle b, Y\rangle]$ and $t = a.(a\,b){\cdot}X$, then $\sigma(t) = a.\langle a, (a\,b){\cdot}Y\rangle$. Given substitutions σ and σ', and freshness environments ∇ and ∇', we write*

$$(a) \quad \nabla' \vdash \sigma(\nabla) \qquad \text{and} \qquad (b) \quad \nabla \vdash \sigma \approx \sigma' \tag{9}$$

to mean that (for a) $\nabla' \vdash a \# \sigma(X)$ holds for each $(a \# X) \in \nabla$ and (for b) $\nabla \vdash \sigma(X) \approx \sigma'(X)$ holds for all $X \in dom(\sigma) \cup dom(\sigma')$.

Lemma 2 (Substitution). *Substitution commutes with the permutation action: $\sigma(\pi{\cdot}t) = \pi{\cdot}(\sigma(t))$. Substitution preserves $\approx$ and $\#$ in the following sense:*

- *if $\nabla' \vdash \sigma(\nabla)$ and $\nabla \vdash t \approx t'$, then $\nabla' \vdash \sigma(t) \approx \sigma(t')$;*
- *if $\nabla' \vdash \sigma(\nabla)$ and $\nabla \vdash a \# t$, then $\nabla' \vdash a \# \sigma(t)$.*

Proof. The first sentence follows by induction on the structure of t. The second follows by induction on the proofs of $\nabla \vdash t \approx t'$ and $\nabla \vdash a \# t$ from the rules in Fig. 2, using the first sentence and the (proof of) Lemma 1. $\square$

We claim that the relation $\approx$ defined in Fig. 2 gives the correct notion of α-equivalence for terms over a nominal signature. This is reasonable, given Lemma 1 and the fact that, by definition, it satisfies rules ($\approx$-abstraction-1) and ($\approx$-abstraction-2). Further evidence is provided by the following theorem, which shows that for ground terms $\approx$ agrees with the following more traditional definition of α-equivalence.

Definition 5 (Naïve $\boldsymbol{\alpha}$-equivalence). *Define the binary relation $t =_\alpha t'$ between terms over a nominal signature to be the least sort-respecting congruence relation satisfying $a.t =_\alpha b.[a{\mapsto}b]t$ whenever b is an atom (of the same sort as a) not occurring at all in t. Here $[a{\mapsto}b]t$ indicates the result of replacing all free occurrences of a with b in t.*

Theorem 1 (Adequacy). *If t and t' are* **ground terms** (i.e. terms with no variables and hence no suspensions) *over a nominal signature, then the relation $t =_\alpha t'$ of Def. 5 holds if and only if $\varnothing \vdash t \approx t'$ is provable from the rules in Fig. 2. Furthermore, $\varnothing \vdash a \# t$ is provable if and only if a is not in the set $FA(t)$ of free atoms of t.*
Proof. The proof is similar to the proof of [11, Proposition 2.2]. $\square$

For non-ground terms, the relations $=_\alpha$ and $\approx$ *differ*. For example $a.X =_\alpha b.X$ always holds, whereas $\varnothing \vdash a.X \approx b.X$ is not provable unless $a = b$. This disagreement is to be expected, since we noted in the Introduction that $=_\alpha$ is *not* preserved by substitution, whereas from Lemma 2 we know that $\approx$ is.

Remark 1. Further evidence for the status of $\approx$ and $\#$ is provided by a natural interpretation of judgements provable from the rules in Fig. 2 in the universe of FM-sets [11]. The details will appear in the full version of this paper.

3 Unification

Given terms t and t' of the same sort over a nominal signature, can we decide whether or not there is a substitution of terms for the variables in t and t' that makes them equal in the sense of the relation $\approx$ introduced in the previous section? Since instances of $\approx$ in general are established modulo freshness constraints, it makes more sense to ask whether or not there is both a substitution σ and a freshness environment ∇ for which $\nabla \vdash \sigma(t) \approx \sigma(t')$ holds. As for ordinary first-order unification, solving such an equational problem may throw up *several* equational subproblems; but an added complication here is that because of rule ($\approx$-abstraction-2) in Fig. 2, equational problems may generate *freshness* problems, i.e. ones involving the relation $\#$. We are thus led to the following definition of unification problems for nominal equational logic.

Definition 6. *A* ***unification problem*** *P over a nominal signature is a finite set of atomic problems, each of which is either an* ***equational problem*** *$t \approx ?\ t'$ where t and t' are terms of the same sort over the signature, or a* ***freshness problem*** *$a \ \#?\ t$ where a is an atom and t a term over the signature. A* ***solution*** *for P consists of a pair (∇, σ) where ∇ is a freshness environment and σ is a substitution satisfying*

- $\nabla \vdash a \# \sigma(t)$ *for each* $(a \ \#?\ t) \in P$ *and*
- $\nabla \vdash \sigma(t) \approx \sigma(t')$ *for each* $(t \approx ?\ t') \in P$.

Such a pair is a ***most general solution*** *for P if given any other solution (∇', σ'), then there is a substitution σ'' satisfying $\nabla' \vdash \sigma''(\nabla)$ and $\nabla' \vdash \sigma'' \circ \sigma \approx \sigma'$. (Here we have used the notation of* (9)*; and $\sigma'' \circ \sigma$ denotes the* ***substitution composition*** *of σ followed by σ'', given by $(\sigma'' \circ \sigma)(X) \stackrel{\mathit{def}}{=} \sigma''(\sigma(X))$.)*

Theorem 2 (Nominal unification). *There is an algorithm which, given any nominal unification problem, decides whether or not it has a solution and if it does, returns a most general solution.*

Proof. We describe an algorithm using labelled transformations directly generalising the presentation of first-order unification in [17, Sect. 2.6], which in turn is based upon the approach in [18]. (See also [1, Sect. 4.6] for a detailed exposition, but not using labels.) We use two types of labelled transformation between unification problems, namely

$$P \stackrel{\sigma}{\Longrightarrow} P' \quad \text{and} \quad P \stackrel{\nabla}{\Longrightarrow} P'$$

where the substitution σ is either the identity ε, or a single replacement $[X := t]$; and where the freshness environment ∇ is either empty $\varnothing$, or a singleton $\{a \ \# \ X\}$. The legal transformations are given in Fig. 3. This figure uses the notation $P \uplus P'$ to indicate *disjoint union* of problem sets; and the notation σP to indicate the problem resulting from applying the substitution σ to all the terms occurring in the problem P.

Given a unification problem P, the algorithm proceeds in two phases. In the first phase it applies as many $\stackrel{\sigma}{\Longrightarrow}$ transformations as possible (non-deterministically). If this results in a problem containing no equational subproblems then it proceeds to the second phase; otherwise it halts with failure. In the second phase it applies as many $\stackrel{\nabla}{\Longrightarrow}$ transformations as possible (non-deterministically). If this does not result in the empty problem, then it halts with failure; otherwise overall it has constructed a transformation

($\approx$?-unit)	$\{\langle\rangle \approx? \langle\rangle\} \uplus P \stackrel{\varepsilon}{\Longrightarrow} P$
($\approx$?-pair)	$\{\langle t_1, t_2\rangle \approx? \langle t'_1, t'_2\rangle\} \uplus P \stackrel{\varepsilon}{\Longrightarrow} \{t_1 \approx? t'_1, t_2 \approx? t'_2\} \cup P$
($\approx$?-function symbol)	$\{f\,t \approx? f\,t'\} \uplus P \stackrel{\varepsilon}{\Longrightarrow} \{t \approx? t'\} \cup P$
($\approx$?-abstraction-1)	$\{a.t \approx? a.t'\} \uplus P \stackrel{\varepsilon}{\Longrightarrow} \{t \approx? t'\} \cup P$
($\approx$?-abstraction-2)	$\{a.t \approx? a'.t'\} \uplus P \stackrel{\varepsilon}{\Longrightarrow} \{t \approx? (a\,a')\boldsymbol{\cdot} t', a \mathrel{\#?} t'\} \cup P$ provided $a \neq a'$
($\approx$?-atom)	$\{a \approx? a\} \uplus P \stackrel{\varepsilon}{\Longrightarrow} P$
($\approx$?-suspension)	$\{\pi\boldsymbol{\cdot} X \approx? \pi'\boldsymbol{\cdot} X\} \uplus P \stackrel{\varepsilon}{\Longrightarrow} \{a \mathrel{\#?} X \mid a \in ds(\pi, \pi')\} \cup P$
($\approx$?-variable)	$\left.\begin{array}{r}\{t \approx? \pi\boldsymbol{\cdot} X\} \uplus P\\ \{\pi\boldsymbol{\cdot} X \approx? t\} \uplus P\end{array}\right\} \stackrel{\sigma}{\Longrightarrow} \sigma P$ with $\sigma = [X := \pi^{-1}\boldsymbol{\cdot} t]$ provided X does not occur in t
($\#$?-unit)	$\{a \mathrel{\#?} \langle\rangle\} \uplus P \stackrel{\varnothing}{\Longrightarrow} P$
($\#$?-pair)	$\{a \mathrel{\#?} \langle t_1, t_2\rangle\} \uplus P \stackrel{\varnothing}{\Longrightarrow} \{a \mathrel{\#?} t_1, a \mathrel{\#?} t_2\} \cup P$
($\#$?-function symbol)	$\{a \mathrel{\#?} f\,t\} \uplus P \stackrel{\varnothing}{\Longrightarrow} \{a \mathrel{\#?} t\} \cup P$
($\#$?-abstraction-1)	$\{a \mathrel{\#?} a.t\} \uplus P \stackrel{\varnothing}{\Longrightarrow} P$
($\#$?-abstraction-2)	$\{a \mathrel{\#?} a'.t\} \uplus P \stackrel{\varnothing}{\Longrightarrow} \{a \mathrel{\#?} t\} \cup P$ provided $a \neq a'$
($\#$?-atom)	$\{a \mathrel{\#?} a'\} \uplus P \stackrel{\varnothing}{\Longrightarrow} P$ provided $a \neq a'$
($\#$?-suspension)	$\{a \mathrel{\#?} \pi\boldsymbol{\cdot} X\} \uplus P \stackrel{\nabla}{\Longrightarrow} P$ with $\nabla = \{\pi^{-1}\boldsymbol{\cdot} a \mathrel{\#} X\}$

Fig. 3. Labelled transformations.

sequence of the form

$$P \stackrel{\sigma_1}{\Longrightarrow} \cdots \stackrel{\sigma_n}{\Longrightarrow} P' \stackrel{\nabla_1}{\Longrightarrow} \cdots \stackrel{\nabla_m}{\Longrightarrow} \varnothing \tag{10}$$

(where P' does not contain any equational subproblems) and the algorithm returns the solution $(\nabla_1 \cup \cdots \cup \nabla_m,\, \sigma_n \circ \cdots \circ \sigma_1)$.

It is not hard to devise a well-founded ordering on nominal unification problems to show that each phase of the algorithm must terminate. So one just has to show that

(a) if the algorithm fails on P, then P has no solution; and
(b) if the algorithm succeeds on P, then the result it produces is a most general solution.

When failure happens it is because of certain subproblems that manifestly have no solution (e.g. in the first phase, $a \approx? a'$ with $a \neq a'$, and $\pi\boldsymbol{\cdot} X \approx? f\,t$ or $f\,t \approx? \pi\boldsymbol{\cdot} X$ with X occurring in t; in the second phase, $a \mathrel{\#?} a$). Therefore part (a) is a consequence of the following two properties of transformations, where we write $\mathcal{U}(P)$ for the set of all solutions for a problem P:

$$\text{if } (\nabla', \sigma') \in \mathcal{U}(P) \text{ and } P \stackrel{\sigma}{\Longrightarrow} P', \text{then } (\nabla', \sigma') \in \mathcal{U}(P') \text{ and } \nabla' \vdash \sigma' \circ \sigma \approx \sigma' \tag{11}$$

$$\text{if } (\nabla', \sigma') \in \mathcal{U}(P) \text{ and } P \stackrel{\nabla}{\Longrightarrow} P', \text{then } (\nabla', \sigma') \in \mathcal{U}(P') \text{ and } \nabla' \vdash \sigma'(\nabla). \tag{12}$$

For part (b), one first shows

$$\text{if } (\nabla', \sigma') \in \mathcal{U}(P') \text{ and } P \stackrel{\sigma}{\Longrightarrow} P', \text{then } (\nabla', \sigma' \circ \sigma) \in \mathcal{U}(P) \tag{13}$$

$$\text{if } (\nabla', \sigma') \in \mathcal{U}(P'),\ P \stackrel{\nabla}{\Longrightarrow} P' \text{ and } \nabla'' \vdash \sigma'(\nabla), \text{then } (\nabla' \cup \nabla'', \sigma') \in \mathcal{U}(P). \tag{14}$$

From these and the fact that $(\varnothing, \varepsilon) \in \mathcal{U}(\varnothing)$, one gets that if a sequence like (10) exists, then $(\nabla, \sigma) \stackrel{\text{def}}{=} (\nabla_1 \cup \cdots \cup \nabla_m,\, \sigma_n \circ \cdots \circ \sigma_1)$ is in $\mathcal{U}(P)$. Furthermore from (11) and (12), we get that any other solution $(\nabla', \sigma') \in \mathcal{U}(P)$ satisfies $\nabla' \vdash \sigma'(\nabla)$ and $\nabla' \vdash \sigma' \circ \sigma \approx \sigma'$, so that (∇, σ) is indeed a most general solution. $\square$

$$
\begin{array}{ll}
P_1 \stackrel{\varepsilon}{\Longrightarrow} \{\mathtt{fn}\,b.\mathtt{app}\langle X_1, \mathtt{vr}\,b\rangle \approx?\ \mathtt{fn}\,b.\mathtt{app}\langle \mathtt{vr}\,b, (a\,b)\cdot X_1\rangle,\ a \mathrel{\#?} \mathtt{fn}\,a.\mathtt{app}\langle \mathtt{vr}\,a, X_1\rangle\} & (\approx?\text{-abstraction-2}) \\
\quad \stackrel{\varepsilon}{\Longrightarrow} \{\mathtt{app}\langle X_1, \mathtt{vr}\,b\rangle \approx?\ \mathtt{app}\langle \mathtt{vr}\,b, (a\,b)\cdot X_1\rangle,\ a \mathrel{\#?} \mathtt{fn}\,a.\mathtt{app}\langle \mathtt{vr}\,a, X_1\rangle\} & (\approx?\text{-abstraction-1}) \\
\quad \cdots\cdots & \cdots \\
\quad \stackrel{\varepsilon}{\Longrightarrow} \{X_1 \approx?\ \mathtt{vr}\,b,\ \mathtt{vr}\,b \approx?\ (a\,b)\cdot X_1,\ a \mathrel{\#?} \mathtt{fn}\,a.\mathtt{app}\langle \mathtt{vr}\,a, X_1\rangle\} & (\approx?\text{-pair}) \\
\quad \stackrel{\sigma}{\Longrightarrow} \{\mathtt{vr}\,b \approx?\ \mathtt{vr}\,a,\ a \mathrel{\#?} \mathtt{fn}\,a.\mathtt{app}\langle \mathtt{vr}\,a, \mathtt{vr}\,b\rangle\} \quad \text{with } \sigma = [X_1 := \mathtt{vr}\,b] & (\approx?\text{-variable}) \\
\quad \stackrel{\varepsilon}{\Longrightarrow} \{b \approx?\ a,\ a \mathrel{\#?} \mathtt{fn}\,a.\mathtt{app}\langle \mathtt{vr}\,a, \mathtt{vr}\,b\rangle\} & (\approx?\text{-fnctn symbol}) \\
\qquad \text{FAIL} & \\
P_4 \stackrel{\varepsilon}{\Longrightarrow} \{\mathtt{fn}\,b.\mathtt{app}\langle \mathtt{vr}\,b, X_6\rangle \approx?\ \mathtt{fn}\,a.\mathtt{app}\langle \mathtt{vr}\,a, X_7\rangle\} & (\approx?\text{-abstraction-1}) \\
\quad \stackrel{\varepsilon}{\Longrightarrow} \{\mathtt{app}\langle \mathtt{vr}\,b, X_6\rangle \approx?\ \mathtt{app}\langle \mathtt{vr}\,b, (b\,a)\cdot X_7\rangle,\ b \mathrel{\#?} \mathtt{app}\langle \mathtt{vr}\,a, X_7\rangle\} & (\approx?\text{-abstraction-2}) \\
\quad \cdots\cdots & \cdots \\
\quad \stackrel{\varepsilon}{\Longrightarrow} \{b \approx?\ b,\ X_6 \approx?\ (b\,a)\cdot X_7,\ b \mathrel{\#?} \mathtt{app}\langle \mathtt{vr}\,a, X_7\rangle\} & (\approx?\text{-fnctn symbol}) \\
\quad \stackrel{\varepsilon}{\Longrightarrow} \{X_6 \approx?\ (b\,a)\cdot X_7,\ b \mathrel{\#?} \mathtt{app}\langle \mathtt{vr}\,a, X_7\rangle\} & (\approx?\text{-atom}) \\
\quad \stackrel{\sigma}{\Longrightarrow} \{b \mathrel{\#?} \mathtt{app}\langle \mathtt{vr}\,a, X_7\rangle\} \quad \text{with } \sigma = [X_6 := (b\,a)\cdot X_7] & (\approx?\text{-variable}) \\
\quad \stackrel{\varnothing}{\Longrightarrow} \{b \mathrel{\#?} \langle \mathtt{vr}\,a, X_7\rangle\} & (\#?\text{-fnctn symbol}) \\
\quad \cdots\cdots & \cdots \\
\quad \stackrel{\varnothing}{\Longrightarrow} \{b \mathrel{\#?} a,\ b \mathrel{\#?} X_7\} & (\#?\text{-fnctn symbol}) \\
\quad \stackrel{\varnothing}{\Longrightarrow} \{b \mathrel{\#?} X_7\} & (\#?\text{-atom}) \\
\quad \stackrel{\nabla}{\Longrightarrow} \varnothing \quad \text{with } \nabla = \{b \mathrel{\#} X_7\} & (\#?\text{-suspension})
\end{array}
$$

Fig. 4. Example derivations

Example 2. Using the first three function symbols of the nominal signature of Example 1 to represent λ-terms, the Quiz at the end of the Introduction translates into the following four unification problems over that signature, where a and b are distinct atoms of sort vid and $X_1, \ldots, X_7$ are distinct variables of sort exp:

$$
\begin{aligned}
P_1 &\stackrel{\text{def}}{=} \{\mathtt{fn}\,a.\mathtt{fn}\,b.\mathtt{app}\langle X_1, \mathtt{vr}\,b\rangle \approx?\ \mathtt{fn}\,b.\mathtt{fn}\,a.\mathtt{app}\langle \mathtt{vr}\,a, X_1\rangle\},\\
P_2 &\stackrel{\text{def}}{=} \{\mathtt{fn}\,a.\mathtt{fn}\,b.\mathtt{app}\langle X_2, \mathtt{vr}\,b\rangle \approx?\ \mathtt{fn}\,b.\mathtt{fn}\,a.\mathtt{app}\langle \mathtt{vr}\,a, X_3\rangle\},\\
P_3 &\stackrel{\text{def}}{=} \{\mathtt{fn}\,a.\mathtt{fn}\,b.\mathtt{app}\langle \mathtt{vr}\,b, X_4\rangle \approx?\ \mathtt{fn}\,b.\mathtt{fn}\,a.\mathtt{app}\langle \mathtt{vr}\,a, X_5\rangle\},\\
P_4 &\stackrel{\text{def}}{=} \{\mathtt{fn}\,a.\mathtt{fn}\,b.\mathtt{app}\langle \mathtt{vr}\,b, X_6\rangle \approx?\ \mathtt{fn}\,a.\mathtt{fn}\,a.\mathtt{app}\langle \mathtt{vr}\,a, X_7\rangle\}.
\end{aligned}
$$

Applying the nominal unification algorithm described above, we find that

- P_1 has no solution;
- P_2 has a most general solution given by $\nabla_2 = \varnothing$ and $\sigma_2 = [X_2 := \mathtt{vr}\,b, X_3 := \mathtt{vr}\,a]$;
- P_3 has a most general solution given by $\nabla_3 = \varnothing$ and $\sigma_3 = [X_4 := (a\,b)\cdot X_5]$;
- P_4 has a most general solution given by $\nabla_4 = \{b \mathrel{\#} X_7\}$ and $\sigma_3 = [X_6 := (b\,a)\cdot X_7]$.

Derivations for P_1 and P_4 are sketched in Fig. 4. Using the Adequacy Theorem 1, one can interpret these solutions as the following statements about the λ-terms from the quiz.

Quiz answers

1. There is no λ-term M_1 making the first pair of terms α-equivalent.
2. The only solution for the second problem is to take $M_2 = b$ and $M_3 = a$.
3. For the third problem we can take M_5 to be any λ-term, so long as we take M_4 to be the result of swapping all occurrences of a and b throughout M_5.
4. For the last problem, we can take M_7 to be any λ-term that *does not contain free occurrences of* b, so long as we take M_6 to be the result of swapping all occurrences of b and a throughout M_7, or equivalently (since b is not free in M_7), taking M_6 to be the result of replacing all free occurrences of a in M_7 with b.

Remark 2 (Atoms are not variables). Nominal unification unifies variables, but it does not unify atoms. Indeed the operation of identifying two atoms by renaming does not necessarily preserve the validity of the judgements in Fig. 2. For example, $\varnothing \vdash a.b \approx c.b$ holds if $b \neq a, c$; but renaming b to be a in this judgement we get $\varnothing \vdash a.a \approx c.a$, which does not hold so long as $a \neq c$. Referring to Def. 2, you will see that we do allow variables ranging over sorts of atoms; and such variables can be unified like any other variables. However, if A is such a variable, then it cannot appear in abstraction position, i.e. as $A.t$. This is because we specifically restricted abstraction to range over atoms, rather than over arbitrary terms of atom sort. Such a restriction seems necessary to obtain single, most general, solutions to nominal unification problems. For without such a restriction, because of rule ($\approx$-abstraction-2) we would also have to allow variables to appear on the left-hand side of freshness relations and in suspended permutations. So then we would get unification problems like $\{(A\,B){\boldsymbol\cdot}C \approx ?\ C\}$, where A, B and C are variables of atom sort; this has two incomparable solutions, namely $(\varnothing, [A := B])$ and $(\{A \# C, B \# C\}, \varepsilon)$.

4 Related work

Most previous work on unification for languages with binders is based on forms of higher-order unification, i.e. solving equations between λ-terms modulo $\alpha\beta\eta$-equivalence by capture-avoiding substitution of terms for function variables. Notable among that work is Miller's *higher-order pattern unification* used in his L_λ logic programming language [20]. This kind of unification retains the good properties of first-order unification: a linear-time decision procedure and existence of most general unifiers. However it imposes a restriction on the form of λ-terms to be unified; namely that function variables may only be applied to distinct bound variables. An empirical study by Michaylov and Pfenning [19] suggests that most unifications arising dynamically in higher-order logic programming satisfy Miller's restriction, but that it rules out some useful programming idioms. For us, the main disadvantage of L_λ is one common to most approaches based on higher-order abstract syntax: one cannot *directly* express the common idiom of possibly-capturing substitution of terms for metavariables. Instead one has to replace metavariables, X, with function variables applied to distinct lists of (bound) variables, $F\,x_1 \ldots x_n$, and use capture-avoiding substitution.

Dowek *et al* [8] present a version of higher-order pattern unification for $\lambda\sigma$ (a λ-calculus with de-Bruijn indices and explicit substitutions) in which unification problems are solved, like in nominal unification, by textual replacements of terms for variables; however a 'pre-cooking' operation ensures that the textual replacements can be (faithfully) related to capture-avoiding substitutions. It seems that nominal unification problems can be encoded into higher-order pattern unification problems using a *non-trivial* translation (the details of this encoding still remain to be investigated). But even if it turns out that it is possible to simulate nominal unification by higher-order pattern unification, the calculations involved in translating our terms into higher-order pattern terms and then using higher-order pattern unification are far more intricate than our simple algorithm that solves nominal unification problems directly.

Hamana [13, 14] manages to add possibly-capturing substitution to a language like Miller's L_λ. This is achieved by adding syntax for explicit renaming operations and by recording implicit dependencies of variables upon bindable names in a typing context. The mathematical foundation for Hamana's system is the model of binding syntax of Fiore *et al* [9]. The mathematical foundation for our work appeared concurrently [10] and is in a sense complementary. For in Hamana's system the typing context restricts which terms may be substituted for a variable by giving a finite set of names that *must contain* the free names of such a term; whereas we give a finite set of names which the term's free variables *must avoid*. Since α-conversion is phrased in terms of avoidance, i.e. freshness of names, our approach seems more natural if one wants to compute α-equivalences concretely. On top of that, our use of name permutations, rather than arbitrary renaming functions, leads to technical simplifications. In any case, the bottom line is that Hamana's system seems more complicated than the one presented here and does not possess most general unifiers.

5 Conclusion

In this paper we have proposed a solution to the problem of finding possibly-capturing substitutions that unify terms involving binders up to α-conversion. To do so we considered a many-sorted first-order term language with distinguished collections of constants called *atoms* and with *atom-abstraction* operations for binding atoms in terms. This provides a simple, but flexible, framework for specifying binding operations and their scopes, in which the bound entities are explicitly named. By using variables prefixed with suspended permutations, one can have substitution of terms for variables both allow capture of atoms by binders and respect α-equivalence (renaming of bound atoms). The definition of α-equivalence for the term language makes use of an auxiliary *freshness* relation between atoms and terms which generalises the 'not a free atom of' relation from ground terms to terms with variables; furthermore, because variables stand for unknown terms, hence with unknown free atoms, it is necessary to make hypotheses about the freshness of atoms for variables in judgements about term equivalence and freshness. This reliance on 'freshness' is the main novelty—it arises from the work reported in [11, 23]. It leads to a new notion of unification problem in which instances of both equivalence and freshness have to be solved by giving term-substitutions and (possibly) freshness conditions on variables in the solution. We showed that this unification problem is decidable and unitary.

Cheney and Urban [5] are investigating the extent to which nominal unification can be used in resolution-based proof search for a form of first-order logic programming for languages with binders (with a view to providing better machine-assistance for structural operational semantics). Such a logic programming language should permit a concrete, 'nominal' approach to bound entities in programs while ensuring that computation (which in this case is the computation of answers to queries) respects α-equivalence between terms. This is illustrated with the Prolog-like program in Fig. 5, which implements a simple typing algorithm for λ-terms. Interesting is the third clause. First, note the term `(lam x.M)`, which unifies with any λ-abstraction. The binder `x`, roughly speaking, has in the 'nominal' approach a value which can be used in the body of the clause, for example for adding `(pair x A)` to the context `Gamma`. Second, the

```
type Gamma (var X) A :- mem (pair X A) Gamma.
type Gamma (app M N) B :- type Gamma M (arrow A B), type Gamma N A.
type Gamma (lam x.M) (arrow A B) / x#Gamma :- type (pair x A)::Gamma M B.
mem A A::Tail.
mem A B::Tail :- mem A Tail.
```

Fig. 5. A Prolog-like program implementing the typing rules for the simply-typed λ-calculus.

freshness constraint `x#Gamma` ensures that `Gamma` cannot be replaced by a term that contains `x` freely. Since this clause is intended to implement the usual rule for typing λ-abstractions

$$\frac{\{x : A\} \cup \Gamma \, \triangleright \, M : B}{\Gamma \, \triangleright \, \lambda x.M : A \supset B}$$

its operational behaviour is given by: choose fresh names for `Gamma`, `x`, `M`, `A` and `B` (this is standard in Prolog-like languages), unify the head of the clause with the goal formula, apply the resulting unifier to the body of the clause and make sure that `Gamma` is not replaced by a term that contains freely the fresh name we have chosen for `x`. Similar facilities for *functional programming* already exist in the FreshML language, built upon the same foundations: see [27] and `www.freshml.org`. We are also interested in the special case of 'nominal matching' and its application to term-rewriting modulo α-equivalence.

If these applications show that nominal unification is practically useful, then it becomes important to study its complexity. The presentations of the term language in Section 2 and of the algorithm in Section 3 were chosen for clarity and to make the proof of correctness easier[2] rather than for efficiency. In any case, it remains to be investigated whether the swapping and freshness computations that we have added to ordinary, first-order unification result in greater than linear-time complexity.

Acknowledgements: We thank Gilles Dowek, Roy Dyckhoff, Dale Miller, Francois Pottier and Helmut Schwichtenberg for comments on this work. This research was supported by UK EPSRC grants GR/R29697 (Urban) and GR/R07615 (Pitts and Gabbay).

References

1. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
2. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.
3. L. Caires and L. Cardelli. A spatial logic for concurrency II. In *Proc. of CONCUR 2002*, volume 2421 of *LNCS*, pages 209–225. Springer-Verlag, 2002.
4. L. Cardelli, P. Gardner, and G. Ghelli. Manipulating trees with hidden labels. In *Proc. of FOSSACS*, volume 2620 of *LNCS*, pages 216–232. Springer-Verlag, 2003.
5. J. Cheney and C. Urban. αProlog, a fresh approach to logic programming modulo α-equivalence. In *Proc. of UNIF 2003*, number DSIC-II/12/03 in Departamento de Sistemas Informáticos y Computación Technical Report Series. Universidad Politécnica de Valencia, 2003.

[2] See `http://www.cl.cam.ac.uk/~cu200/Unification` for the Isabelle proof scripts.

6. G. Dowek. Higher-order unification and matching. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 16, pages 1009–1062. Elsevier, 2001.
7. G. Dowek, T. Hardin, and C. Kirchner. Higher-order unification via explicit substitutions. In *10th Symposium of LICS*, pages 366–374. IEEE Computer Society Press, 1995.
8. G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Higher-order unification via explicit substitutions: the case of higher-order patterns. In *Proc. of JICSLP*, pages 259–273, 1996.
9. M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In *14th Symposium of LICS*, pages 193–202. IEEE Computer Society Press, 1999.
10. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *14th Symposium of LICS*, pages 214–224. IEEE Computer Society Press, 1999.
11. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.
12. C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.
13. M. Hamana. A logic programming language based on binding algebras. In *Proc. of TACS 2001*, volume 2215 of *LNCS*, pages 243–262. Springer-Verlag, 2001.
14. M. Hamana. Simple β_0-unification for terms with context holes. In *Proc. of UNIF 2002*, 2002. Unpublished proceedings.
15. M. Hashimoto and A. Ohori. A typed context calculus. *TCS*, 266:249–271, 2001.
16. F. Honsell, M. Miculan, and I. Scagnetto. An axiomatic approach to metareasoning on nominal algebras in HOAS. In *Proc. of ICALP 2001*, volume 2076 of *LNCS*, pages 963–978. Springer-Verlag, 2001.
17. J. W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume 2*, pages 1–116. OUP, 1992.
18. A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Programming Languages and Systems*, 4(2):258–282, 1982.
19. S. Michaylov and F. Pfenning. An empirical study of the runtime behaviour of higher-order logic programs. In *Proc. Workshop on the λProlog Programming Language*, pages 257–271, 1992. CIS Technical Report MS-CIS-92-86.
20. D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1:497–536, 1991.
21. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes (parts I and II). *Information and Computation*, 100:1–77, 1992.
22. F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proc. ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988.
23. A. M. Pitts. Nominal logic: A first order theory of names and binding. In *Proc. of TACS 2001*, volume 2215 of *LNCS*, pages 219–242. Springer-Verlag, 2001.
24. A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In *Proc. of MPC2000*, volume 1837 of *LNCS*, pages 230–255. Springer-Verlag, 2000.
25. G. D. Plotkin. An illative theory of relations. In *Situation Theory and its Applications*, volume 22 of *CSLI Lecure Notes*, pages 133–146. Stanford University, 1990.
26. M. Sato, T. Sakurai, and Y. Kameyama. A simply typed context calculus with first-class environments. *Journal of Functional and Logic Programming*, 2002(4), 2002.
27. M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Proc. of ICFP 2003*. ACM Press, 2003.