Relating Nominal and Higher-Order Pattern Unification

James Cheney

University of Edinburgh
`jcheney@inf.ed.ac.uk`

Abstract. Higher-order pattern unification and nominal unification are two approaches to unifying modulo some form of α-equivalence (consistent renaming of bound names). The higher-order and nominal approaches seem superficially dissimilar. However, we show that a natural *concretion* (or name-application) operation for nominal terms can be used to simulate the behavior of higher-order patterns. We describe a form of nominal terms called *nominal patterns* that includes concretion and for which unification is equivalent to a special case of higher-order pattern unification, and then show that full higher-order pattern unification can be reduced to nominal unification via nominal patterns.

1 Introduction

Higher-order unification is the unification of simply-typed λ-terms up to α-, β-, and (sometimes) η-equivalence. It has been studied for over thirty years.. Although it is undecidable and of infinitary unification type, Huet's algorithm [6] performs well in practice, and Miller identified a well-behaved special case called *higher-order pattern unification* [7, 13, 11, 1] that is decidable in linear time and possesses unique most general unifiers. In higher-order patterns, uses of metavariables (variables for which terms may be substituted) are limited so that the nondeterministic search needed in full higher-order unification can be avoided. A key aspect of higher-order unification is that substitution is capture-avoiding. For example, the unification problem $\lambda x.M \approx ?\ \lambda y.f\ y$ has no solution, since although both sides could be made equal by making a capturing substitution $f\ x$ for M, there is no way to make both sides equal using capture-avoiding substitution to instantiate M.

Nominal unification [15] is the unification of *nominal terms*, which include special *name* or *atom* symbols, a *name-swapping* operation, an *abstraction* operation for name-binding, and *freshness* relation. Equality and freshness for nominal terms coincide with classical definitions of α-equivalence and the "not-free-in" relation $- \notin FV(-)$, respectively. Nominal unification is decidable in at worst quadratic time (exact complexity bounds are not yet known). Nominal unification is based on *nominal logic*, a logic formalizing a novel approach to abstract syntax with bound names due to Gabbay and Pitts [3]. There are two aspects of nominal unification that contrast sharply with higher-order unification. First, abstraction is not considered to bind names, and metavariables may mention arbitrary names, so that the problem $\langle a \rangle M \approx ? \langle b \rangle f(b)$ does have solution $M = f(a)$. Second, abstractions are not considered to be functions, and there is no built-in notion of "abstraction application". Instead, nominal unifiers can be expressed in terms of the swapping operation $(a\ b) \cdot t$, which describes the result of exchanging all occurrences of a and b within t, and freshness constraints $a \# t$, which assert that a name a is fresh for a term t. For example, the unification problem $\langle a \rangle M \approx ? \langle b \rangle f(N, b)$ has most general solution $M = f((a\ b) \cdot N, a)$ subject to the constraint that $a \# N$. This unifier shows how to compute M as a function of N while excluding false solutions such as $M = f(b, a), N = a$, since $\langle a \rangle f(a, b) \not\approx \langle b \rangle f(b, a)$.

Despite these differences, nominal and higher-order pattern unification appear closely related. In fact, at first glance, one might wonder if they are not merely different presentations of the same algorithm. Both are techniques for equational reasoning about languages involving bound identifiers. Both algorithms rely on computing with permutations of bound names: the higher-order pattern restriction can be seen as a sufficient condition to ensure that such permutations always exist. In fact, as noted by Urban, Pitts, and Gabbay [15], there is a translation from nominal unification problems to higher-order pattern unification problems that preserves satisfiability. In this translation, metavariables are "lifted" so as to be functions of all the names in context. A freshness constraint such as $a \# M$ can be translated to an equation like $\lambda a, b, c.M\ a\ b\ c \approx \lambda a, b, c.N\ b\ c$, which asserts that M cannot be dependent on its first argument (namely, a). However, as argued by Urban et al., it is not straightforward to convert the resulting solutions back to solutions to the original nominal unification problem. As a result, it appears much easier to solve such problems directly using Urban et al.'s algorithm (which seems much simpler than that for higher-order pattern unification in any case).

Another reason to study the relationship between higher-order pattern unification and nominal unification is to provide a logical foundation for higher-order patterns. While both nominal and higher-order unification are grounded in clear logical foundations, higher-order patterns appear motivated solely by algorithmic concerns. If higher-order patterns can be explained using nominal terms, the semantic foundations of the latter could also be used for the former.

In this paper we argue that higher-order pattern unification can be reduced to nominal unification. This relationship helps justify the higher-order pattern restriction and explain why it works. The key idea is that the pattern restriction (that metavariable occurrences are of the form $X\ \overline{v}$ where $\overline{v}$ is a list of distinct names) is essentially the same as a natural freshness restriction on the *concretion* operation. This operation is an elimination form for abstraction that has been considered in some versions of FreshML [12, 14], but so far not incorporated into nominal logic or nominal unification.

The structure of this paper is as follows. First (Section 2), we review higher-order pattern unification and nominal unification. In Section 3, we introduce a type system that enforces the higher-order pattern restriction in a particularly convenient way. In Section 4, we identify a variant of nominal terms called *nominal patterns* that includes the concretion operation and for which unification is equivalent to a special case of higher-order pattern unification. We then (Section 5) show that full higher-order pattern unification can be reduced to nominal pattern unification and (Section 6) that nominal pattern unification can be reduced to nominal unification. Put together, these reductions show that higher-order pattern unification can be implemented via nominal unification. Section 7 and Section 8 discuss related work and conclude.

2 Background

2.1 Higher-order terms, patterns, and unification

Consider infinite sets of variable names $x, y, z, \ldots \in \mathit{Var}$ and metavariables $X, Y, Z \ldots \in \mathit{MVar}$. The terms of the λ-calculus are as follows:

$$t ::= c \mid x \mid \lambda x.t \mid t\ t' \mid X$$

We assume that common notions such as the set of free variables of a term $FV(-)$, α-equivalence, capture-avoiding substitution $-[-/-]$ etc. are defined as usual. In addition, we assume that there are some given base types δ, that types include function types $\tau \to \tau'$, and that well-formedness is defined as usual provided that types are assigned to constants via a signature Σ. A term with no free variables is called *closed*; a term with no metavariables is called *ground*. We often write $f(t_1, \ldots, t_n)$ as a shorthand for $f\ t_1\ \cdots\ t_n$.

Terms are considered equal up to α-equivalence plus two additional equations: β-reduction and η-expansion

$$\begin{array}{lrl} (\beta) & (\lambda x.t)\ u & \approx t[u/x] \\ (\eta) & t & \approx \lambda x.(t\ x) \quad (t : \tau \to \tau', x \notin FV(t)) \end{array}$$

We write θ for a substitution mapping metavariables to λ-terms. Such a substitution may be applied to any λ-term by replacing each metavariable X with $\theta(X)$. If Γ and Γ' are contexts consisting only of metavariables, we write substitution is well-formed ($\Gamma' \vdash \theta : \Gamma$) provided that for each $X : \tau \in \Gamma$, $\Gamma' \vdash \theta(X) : \tau$. Thus, there is no danger of variable capture during substitution, and we have:

Lemma 1. *If $\Gamma \vdash \theta : \Gamma'$ and $\Gamma' \vdash t : \tau$, then $\Gamma \vdash \theta(t) : \tau$.*

We consider higher-order unification to be unification of λ-terms up to the above equational theory, that is, up to $\alpha\beta\eta$-equivalence. (Higher-order unification sometimes refers to unification up to only α and β-equivalence, but for this paper, we do not consider this problem.) Huet [6] gave an algorithm for generating complete sets of higher-order unifiers which performs well in practice. Technically, we consider only problems of the form $\exists \overline{X}.\forall \overline{y}.t \approx ?\ u$, since substitutions θ cannot mention free variables. This excludes problems such as $\forall y.\exists X.X \approx ?\ y$. However, such problems can always be transformed to equivalent $\exists\forall$-problems by *raising* [9] metavariables in order to make their dependence on other variables explicit: for example, transforming $\forall y.\exists X.X \approx? y$ to $\exists F.\forall y.F\ y \approx? y$, where $X \approx F\ y$.

Miller investigated a decidable special case of higher-order unification called *higher-order patterns*. To define higher-order patterns, we first recall that any λ-term (possibly involving metavariables) can be put into a normal form called *η-long, β-normal form* (or ηlβn form), such that (a) no β-redices exist, and (b) no η-expansions can be performed without introducing a β-redex. Note that this normal form is dependent on the types of metavariables. For example, the normal form of $\lambda y, z.(\lambda x.xy)\ (F\ G)$ is $\lambda y, z, b.F\ (\lambda a.G\ a)\ y\ b$, provided $F : (\tau \to \tau') \to \sigma_1 \to \sigma_2 \to \sigma$ and $G : \tau \to \tau'$. Such normal forms conform to the following grammar:

$$t ::= \lambda\overline{v}.x\ \overline{t} \mid \lambda\overline{v}.X\ \overline{t}$$

The insight behind higher-order pattern unification is that all the nondeterminism in higher-order unification comes about because of uncertainty concerning how an unknown X can act on its arguments $\overline{t}$. In general, $\overline{t}$ may include repeated variables or more complex terms involving other metavariables. In higher-order patterns, this uncertainty is eliminated by requiring the argument list $\overline{t}$ in each subterm of the form $\lambda\overline{v}.X\ \overline{t}$ to be a list of distinct bound variables $\overline{w}$. Thus, the above example ηlβn-normal form $\lambda y, z, b.F\ (\lambda a.G\ a)\ y\ b$ is not a pattern, while $\lambda x, y.c\ (F\ y\ x)$ is a pattern.

Higher-order patterns are closed under substitution modulo β-normalization; in fact, the only redices introduced by substituting a higher-order pattern for a metavariable in another higher-order pattern are of the form called β_0 by Miller:

$$(\beta_0) \quad (\lambda x.t)\ y = t[y/x]$$

Unification for higher-order patterns is decidable (in linear time [13]) and most general unifiers exist.

2.2 Nominal terms and unification

We now consider a different language called *nominal terms*[1]. Let ν, ν' be basic *name types*. Let Nm be a set of *names* $a_\nu, b_{\nu'}, \ldots$ tagged with name types and let $MVar$ be a set of metavariables $X, Y, Z, \ldots$. The set of nominal terms is generated by the grammar

$$\begin{aligned} t &::= a_\nu \mid \langle a_\nu \rangle t \mid c \mid t_1\ t_2 \mid \pi \cdot X \quad \pi ::= \mathsf{id} \mid (a_\nu\ b_\nu) \circ \pi \\ \tau &::= \sigma \mid \sigma \to \tau \quad \sigma ::= \delta \mid \nu \mid \langle \nu \rangle \delta \end{aligned}$$

Metavariables are annotated with *suspended permutations* of names, that are to be applied to any value substituted for the variable. A nominal term with no metavariables is called *ground*.

[1] Our version of nominal terms is superficially different from that used on Urban, Pitts, and Gabbay's paper, in order to minimize the number of unimportant differences from higher-order patterns.

Terms of the form $\langle a\rangle t$ are called *abstractions*. An abstraction is an object with a single bound name. However, the name is not considered syntactically bound as in a λ-abstraction; instead, an abstraction describes a semantic value with a bound name. For example, $\langle a\rangle b$ and $\langle b\rangle b$ are not considered to be the same term; however, they have the same meaning. In particular, while term equality behaves like (and is intended to model) α-equivalence for *ground terms*, this is not the case for terms mentioning metavariables (e.g., the equation $\langle a\rangle X \approx \langle b\rangle X$ is not valid in general).

We assume that there is a signature Σ assigning types τ to constants c, such that there are no constants or other closed terms inhabiting any name type. A permutation is considered well-formed if it is composed of swappings of names of the same type only. Contexts Γ associate metavariables to types. The following well-formedness rules are considered:

$$\frac{}{\Gamma \vdash a_\nu : \nu} \quad \frac{c : \tau \in \Sigma}{\Gamma \vdash c : \tau} \quad \frac{\pi \text{ well-formed}}{\Gamma, X : \tau \vdash \pi \cdot X : \tau}$$
$$\frac{\Gamma \vdash t : \tau \to \tau' \quad \Gamma \vdash u : \tau'}{\Gamma \vdash t\ u : \tau'} \quad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \langle a_\nu\rangle t : \langle\nu\rangle\tau}$$

We define a *swapping* function on nominal terms as follows:

$$(a\ b) \cdot a' = \begin{cases} b & (a = a') \\ a & (b = a') \\ a' & (a \neq a' \neq b) \end{cases} \qquad \begin{aligned} (a\ b) \cdot c &= c \\ (a\ b) \cdot (t_1\ t_2) &= ((a\ b) \cdot t_1)\ ((a\ b) \cdot t_2) \\ (a\ b) \cdot \langle a\rangle t &= \langle (a\ b) \cdot a\rangle (a\ b) \cdot t \\ (a\ b) \cdot (\pi \cdot X) &= (a\ b) \circ \pi \cdot X \end{aligned}$$

Also, we define $\pi \cdot t$ as follows:

$$\mathsf{id} \cdot t = t \qquad (a\ b) \circ (\pi \cdot t) = ((a\ b) \cdot \pi) \cdot t$$

We are now in a position to define the meaning of nominal terms. We do this by introducing axioms describing equality and an auxiliary *freshness* relation.

$$\frac{}{a \approx a} \quad \frac{}{c \approx c} \quad \frac{t \approx t' \quad u \approx u'}{t\ u \approx t'\ u'} \quad \frac{t \approx u}{\langle a\rangle t \approx \langle a\rangle u} \quad \frac{t \approx (a\ b) \cdot u \quad a \# u \quad (a \neq b)}{\langle a\rangle t \approx \langle b\rangle u}$$
$$\frac{a \neq b}{a \# b} \quad \frac{}{a \# c} \quad \frac{a \# t \quad a \# u}{a \# t\ u} \quad \frac{}{a \# \langle a\rangle t} \quad \frac{a \# t \quad (a \neq b)}{a \# \langle b\rangle t}$$

Given a substitution function θ mapping metavariables to terms, we write $\theta(t)$ for the result of applying substitution θ to term t. To be precise, the definition of substitution is as follows.

$$\begin{aligned} \theta(a) &= a & \theta(\langle a\rangle t) &= \langle a\rangle\theta(t) \\ \theta(c) &= c & \theta(\pi \cdot X) &= \pi \cdot \theta(X) \end{aligned} \quad \theta(t_1\ t_2) = \theta(t_1)\ \theta(t_2)$$

We require that substitutions are well-formed so that they preserve types, but (unlike for higher-order unification) substitutions are allowed to mention both metavariables and names, so "capturing" substitutions are allowed. For example, if $\theta(X) = a$ then $\theta(\langle a\rangle X) = \langle a\rangle a$.

If θ is a valuation (ground substitution), we write $\theta \vDash t \approx u$ to indicate that $\theta(t) \approx \theta(u)$ and write $\theta \vDash a \# t$ if $a \# \theta(t)$. As usual, a formula A is valid (satisfiable) if for all (resp. some) well-formed substitutions, $\theta \vDash A$ holds. This is extended to validity or satisfiability of sets of formulas P in the obvious way. Similarly, if P is a set of formulas, we write $P \vDash A$ to indicate that whenever $\theta \vDash P$, we also have $\theta \vDash A$.

Urban et al.'s nominal unification algorithm solves the satisfiability problem for sets of equations and freshness constraints. Given a problem P, it produces a unique (up to renaming) most general answer of the form θ, ∇, where θ is a substitution and ∇ is a set of freshness constraints of the form $a \# X$. This answer has the property that $\nabla \vDash \theta(P)$. Moreover, for any other answer ∇', θ' having this property, there exists a substitution ρ such that $\rho(\nabla) \vDash \nabla'$ and $\rho(\nabla) \vDash \rho \circ \theta \approx \theta'$ (where $\theta \approx \theta'$ means $dom(\theta) = dom(\theta')$ and $\forall X \in dom(\theta).\theta(X) \approx \theta'(X)$).

Urban et al. argue that their algorithm can be implemented in quadratic time; however, the exact complexity has not been established. We omit the precise details of the algorithm.

3 A Refined Type System for Higher-Order Patterns

We modify the notation of λ-terms to distinguish between *rigid* applications involving terms $t\ u$ where the head of t is rigid (i.e., a constant or bound variable), and *flexible* applications $t\,\hat{}\,a$, where the head of t is flexible (a metavariable). Also, we assume that variables are tagged with their types: for example, x_τ indicates that x is a variable of type τ. The grammar of such terms is as follows:

$$t ::= c \mid x_\tau \mid \lambda x_\tau.t \mid t\ t' \mid X \mid t\,\hat{}\,x_\tau$$

We use a type system for ηlβn-normalized terms that enforces the pattern restriction. Contexts Γ bind metavariables to types. There are three judgment forms: $\Gamma \vdash t \downarrow \tau$, indicating that t is a rigid atomic term of type τ; $\Gamma \vdash t \Downarrow \tau$, indicating that t is a flexible atomic term of type τ; and $\Gamma \vdash t \uparrow \tau$, indicating that t is a normal term of type τ. Examples of rigid atomic, flexible atomic, and normal terms include $x\ (\lambda y.y)z$, $X\,\hat{}\,y\,\hat{}\,z$, and $\lambda x, y, z.y\ (x\ (\lambda y.y)z)\ (X\,\hat{}\,y\,\hat{}\,z)$, respectively. The well-formedness rules for nominal patterns are as follows:

$$\frac{c : \tau \in \Sigma}{\Gamma \vdash c \downarrow \tau} \quad \frac{}{\Gamma \vdash x_\tau \downarrow \tau} \quad \frac{\Gamma \vdash t \uparrow \tau'}{\Gamma \vdash \lambda x_\tau.t \uparrow \tau \to \tau'} \quad \frac{\Gamma \vdash t \downarrow \tau \to \tau' \quad \Gamma \vdash u \uparrow \tau}{\Gamma \vdash t\ u \downarrow \tau'}$$

$$\frac{}{\Gamma, X : \tau \vdash X \Downarrow \tau} \quad \frac{\Gamma \vdash t \Downarrow \tau \to \tau' \quad (x \notin FV(t))}{\Gamma \vdash t\,\hat{}\,x_\tau \Downarrow \tau'} \quad \frac{\Gamma \vdash t \downarrow \delta}{\Gamma \vdash t \uparrow \delta} \quad \frac{\Gamma \vdash t \Downarrow \delta}{\Gamma \vdash t \uparrow \delta}$$

There is no way to bind a metavariable: λ binds ordinary variables only. We can only convert from an atomic typing to a normal typing at base types δ; this ensures that all the necessary η-expansions take place.

All well-formed terms in this system are ηlβn-normalized. Moreover, as for higher-order patterns generally, only β_0 reductions $(\lambda x_\tau.t)\,\hat{}\,y_\tau \to t[y_\tau/x_\tau]$ need to be performed after a substitution of higher-order patterns for metavariables.

Lemma 2 (Renaming). *Let R be one of $\downarrow, \Downarrow, \uparrow$. If $y_\tau \notin FV(t)$ and $\Gamma \vdash t \; R \; \tau$, then $\Gamma \vdash t[y_\tau/x_\tau] \; R \; \tau$, respectively.*

Lemma 3 (Substitution). *If $\Gamma, X : \tau' \vdash t \uparrow \tau$ and $\Gamma \vdash u \uparrow \tau'$ for $FV(u) = \varnothing$, then $\Gamma \vdash t[u/X] \uparrow \tau$.*

4 Nominal Patterns

We now introduce a slight variant of nominal terms that provides a closer match to higher-order patterns. This language, called *nominal patterns*, is defined by the following grammar:

$$t ::= c \mid t \; t' \mid X \mid a_\nu \mid \langle a_\nu \rangle t \mid t \,@\, a_\nu$$
$$\tau ::= \sigma \mid \sigma \to \tau \quad \sigma ::= \delta \mid \nu \mid \langle \nu \rangle \sigma$$

where as before, δ denotes a base (data) type and ν denotes a name type. As before, we assume that there is a signature assigning τ-types to constants c. Metavariables may not have arbitrary types, but only σ-types (i.e., types built using only data, name, and abstraction types). In addition, we assume that name symbols a_ν are tagged with their name types ν. As for nominal terms, we assume that the only ground terms inhabiting name-types are literal names.

The main difference between ordinary nominal terms and nominal patterns is the presence of the *concretion* operation $(-)@(-)$ that has also been considered in some versions of FreshML [12, 14]. Our type system requires well-formed nominal patterns to satisfy an analogue of the higher-order pattern restriction: in every subterm of the form $t@a$, we require that $a \# t$ holds. In order to simplify this check (and to make nominal patterns more similar to higher-order patterns), we only consider substitutions of patterns such that $FN(t) = \varnothing$, where

$$\begin{array}{rcl} FN(c) = \varnothing & & FN(X) = \varnothing \\ FN(a) = \{a\} & & FN(t \; u) = FN(t) \cup FN(u) \\ FN(t@a) = FN(t) \cup \{a\} & & FN(\langle a \rangle t) = FN(t) - \{a\} \end{array}$$

As a result, $\vDash a \# X$ and $\vDash (a \; b) \cdot X \approx X$ are valid for any names a, b and metavariable X; using these facts we can lift the freshness relation and swapping function to patterns involving metavariables.

As before, signatures Σ map constants to types and names to name types, whereas contexts map metavariables to metavariable types σ. We require patterns to be well-typed, subject to the following rules:

$$\frac{}{\Gamma \vdash a_\nu \downarrow \nu} \quad \frac{c : \tau \in \Sigma}{\Gamma \vdash c \downarrow \tau} \quad \frac{}{\Gamma, X : \sigma \vdash X \downarrow \sigma} \quad \frac{\Gamma \vdash t \downarrow \tau \to \tau' \quad \Gamma \vdash u \uparrow \tau}{\Gamma \vdash t \; u \downarrow \tau'}$$
$$\frac{\Gamma \vdash t \uparrow \sigma}{\Gamma \vdash \langle a_\nu \rangle t \uparrow \langle \nu \rangle \sigma} \quad \frac{\Gamma \vdash t \downarrow \langle \nu \rangle \sigma \quad a \notin FN(t)}{\Gamma \vdash t@a_\nu \downarrow \sigma} \quad \frac{\Gamma \vdash t \downarrow \epsilon \quad (\epsilon = \delta, \nu)}{\Gamma \vdash t \uparrow \epsilon}$$

Abstraction and concretion are construction and destruction operations for the abstraction sort. Thus, nominal patterns are subject to the following β_α- and

η_α-laws:

$$\begin{array}{lrl} (\beta_\alpha) & (\langle a \rangle t)@b \approx (a\ b) \cdot t & \\ (\eta_\alpha) & t \approx \langle a \rangle (t@a) & (t : \langle \nu \rangle \sigma) \end{array}$$

Note that the typing rules ensure that $b \# \langle a \rangle t$ must hold in the first case and $a \# t$ must hold in the second case. We first state some basic properties of nominal patterns.

Lemma 4 (Swapping). *Let R be one of $\downarrow, \uparrow$. If $\Gamma \vdash t\ R\ \tau$ then $\Gamma \vdash (a\ b) \cdot t\ R\ \tau$.*

Lemma 5 (Substitution). *If $\Gamma, X : \tau' \vdash t \uparrow \tau$ and $\Gamma \vdash u \uparrow \tau'$ where $FN(u) = \varnothing$, then $\Gamma \vdash t[u/X] \uparrow \tau$.*

We now show that this axiomatization satisfies the previously given laws of nominal abstraction.

Proposition 1. *For nominal patterns, we have $\langle a \rangle t \approx \langle b \rangle u$ if and only if $a \approx b, t \approx u$ or $a \# u, t \approx (a\ b) \cdot u$. Similarly, if $t : \langle \nu \rangle \tau$, then there exists a_ν and $u : \tau$ such that $t \approx \langle a \rangle u$.*

Proof. Suppose $\langle a \rangle t \approx \langle b \rangle u$. Then $a \# \langle a \rangle t \approx \langle b \rangle u$, so we have

$$t \approx (a\ a) \cdot t \approx (\langle a \rangle t)@a \approx (\langle b \rangle u)@a \approx (a\ b) \cdot u$$

There are two cases. If $a = b$ then $t \approx (a\ b) \cdot u = (a\ a) \cdot u = u$. Otherwise, $a \# u$ and $t \approx (a\ b) \cdot u$.

Now suppose $t : \langle \nu \rangle \tau$. Since $FN(t)$ is finite, we can always find a name $a \notin FN(t)$, so we can form the term $\langle a \rangle (t@a)$. By the η-rule, we have $t \approx \langle a \rangle (t@a)$, thus, a is the required name and $t@a$ the required term of type τ.

The similarity between the β_α and η_α rules for nominal patterns and the β_0 and η rules for higher-order patterns is not a coincidence. We now consider a typed translation from nominal to higher-order patterns. We assume (for convenience) that the constants, names and metavariables of nominal patterns are the same as the constants, variables, and metavariables of higher-order patterns respectively. Similarly, we assume that the name types and data types of the nominal language are base types of the higher-order language. Terms are translated as follows:

$$\begin{array}{rlrl} c^* &= c & a_\nu^* &= a_\nu \\ (t\ u)^* &= t^*\ u^* & (\langle a_\nu \rangle t)^* &= \lambda a_\nu . t^* \\ X^* &= X & (t@a_\nu)^* &= (t^*)\ \hat{}\ a_\nu \end{array}$$

The translation of types is as follows:

$$\begin{array}{rlrl} \delta^* &= \delta & (\sigma \to \tau)^* &= \sigma^* \to \tau^* \\ \nu^* &= \nu & (\langle \nu \rangle \tau)^* &= \nu \to \tau^* \end{array}$$

Contexts and signatures are translated by replacing each type with its starred form.

Example 1. The translation of $t = \langle a \rangle X \mathbin{@} a \mathbin{@} b$ is $\lambda a.X\ a\ b$, where $X : \langle\nu\rangle\langle\nu\rangle\delta$ in the former and $X : \nu \to \nu \to \delta$ in the latter.

Lemma 6. *If $a \notin FN(t)$ then $a^* \notin FV(t^*)$. Also, if $\Gamma \vdash t \uparrow \tau$ then $\Gamma^* \vdash t^* \uparrow \tau^*$.*

Theorem 1. *The translation $(-)^*$ has an inverse $(-)^\dagger$ on its range.*

Proof. Clearly $(-)^*$ is injective, and it is surjective on its range by definition.

Lemma 7. *If $\Gamma \vdash t : \tau$ is a nominal pattern and $b \notin FN(t)$, then $((a\ b) \cdot t)^* = t^*[b/a]$. Dually, if $\Gamma \vdash u : \tau$ is a higher-order pattern in the range of $(-)^*$, and $b \notin FV(t)$, then $(u[a/b])^\dagger = (a\ b) \cdot u^\dagger$.*

Proof. Proof is by induction on the structure of t. If $t = a$, then $((a\ b)\cdot t)^* = b^* = b$ and $t^*[b/a] = a[b/a] = b$. Otherwise, t is a name other than a or b, and swapping, substitution, and the $(-)^*$ translation all fix t. The case for t a constant or metavariable is similar. For $t = t_1\ t_2$, the induction step is straightforward. This leaves the case of abstraction. If $t = \langle a \rangle u$, then $b \mathrel{\#} u$ so by induction we have $((a\ b) \cdot u)^* = u^*[b/a]$, hence

$$\begin{aligned} ((a\ b) \cdot \langle a \rangle u)^* &= (\langle b \rangle (a\ b) \cdot u)^* = \lambda b.((a\ b) \cdot u)^* = \lambda b.u^*[b/a] \\ &\approx_\alpha \lambda a.u^* = (\lambda a.u^*)[b/a] = (\langle a \rangle u)^*[b/a] \end{aligned}$$

If $t = \langle b \rangle u$, then the induction hypothesis does not apply directly, but we can choose a fresh name $b' \mathrel{\#} a, b, t$ such that

$$\begin{aligned} ((a\ b) \cdot \langle b \rangle t)^* &\approx ((a\ b) \cdot \langle b' \rangle (b\ b') \cdot t)^* = (\langle b' \rangle (a\ b) \cdot (b\ b') \cdot t)^* \\ &= \lambda b'.((b\ b') \cdot t)^*[b/a] = \lambda b'.t^*[b'/b][b/a] \approx_\alpha \lambda b.t^*[b/a] = (\langle b \rangle t)^*[b/a] \end{aligned}$$

where the two middle steps rely on the facts that $b \notin FN((b\ b') \cdot t)$ and $b' \notin FN(t)$. The case for $t = \langle a' \rangle t$ for $a' \neq a, b$ is straightforward.

The second part follows immediately from the first by setting $t = u^\dagger$.

Theorem 2. *Let $t, u : \tau$ be nominal patterns. Then $t \approx u$ if and only if $t^* \approx u^*$.*

Proof. Proof is by induction on the derivation of $t \approx u$ in the forward direction. The interesting cases are for β_α and η_α rules. While η_α is straightforward, for β_α we have $(\langle a \rangle t)@b \approx (a\ b) \cdot t$ and want to show that $(\lambda a.t^*)\ b \approx ((a\ b) \cdot t)^*$. By β_0 and the previous lemma we have $(\lambda a.t^*)\ b \approx t^*[b/a] = ((a\ b) \cdot t)^*$.

The reverse direction is similar, except that we need to use the identity $(t[a/b])^\dagger = (a\ b) \cdot t^\dagger$ in the β_0 case.

Corollary 1. *$t \approx u$ is satisfiable if and only if $t^* \approx u^*$ is; moreover, the satisfying valuations θ, θ^* are in bijective correspondence via $(-)^*$.*

This shows that nominal pattern unification coincides with a special case of higher-order pattern unification: specifically, the case for terms in which the only form of binding is λ-abstraction over void base types ν. In fact, many applications of higher-order patterns are possible within this fragment: it is commonplace to use an abstract or empty type for the "type of variable names" in, for example, a higher-order abstract syntax encoding of the π-calculus [10]. However, applications involving λ-abstraction over non-void types are also common [7].

This translation is interesting, but we have only shown that there is a correspondence between two very limited special cases of the two problems. Next we show how to translate full higher-order pattern unification to nominal pattern unification.

5 Higher-order pattern unification as nominal pattern unification

In higher-order patterns, λ-term variables are not limited to a collection of void base types, but may be of any type, so variables may be applied to argument lists including repeated variables, metavariables, or more general terms (that is, the pattern restriction is not required of argument lists whose head is not a variable). This permits the formation of terms such as $\lambda x, y.y\ (\lambda z.Fz)\ x\ x$ which are not in the range of $(-)^*$; i.e., which do not correspond to a nominal pattern. Such terms are not in the domain of $(-)^\dagger$, so the approach investigated in the last section does not apply.

However, there is another translation that works. The reason the idea of the previous section doesn't work is that in higher-order patterns, variables play one of two roles: they can be passed as arguments to metavariables, but they can also act as functions on lists of arguments. The latter role is not supported directly by nominal patterns, because name types ν are populated only by names.

Given a higher-order language L, we construct a nominal language L^{**} possessing a name-type ν_τ for each simple type τ of L and a data type δ for each basic type δ of L. We define a translation on L-types as follows:

$$\delta^{**} = \delta \quad (\tau_1 \to \tau_2)^{**} = \langle \nu_{\tau_1} \rangle {\tau_2}^{**}$$

Note that each τ-type of L translates to a σ-type of L^{**}. Given a signature Σ, we write Σ^{**} for the result of replacing all the types in Σ with their $(-)^{**}$ translations; similarly for contexts Γ^{**}. Moreover, we add the following new constants to L^{**}:

$$\begin{aligned} var_\tau &: \nu_\tau \to \tau^{**} \\ app_{\tau_1\tau_2} &: (\tau_1 \to \tau_2)^{**} \to {\tau_1}^{**} \to {\tau_2}^{**} \end{aligned}$$

This signature is infinite, since the function symbols var_τ and $app_{\tau_1\tau_2}$ are indexed with types. However, in any particular situation, only finitely many ν_τ types and finitely many constants of the above signature need to be considered. After unwinding definitions, the type of $app_{\tau_1\tau_2}$ is $\langle \nu_{\tau_1} \rangle {\tau_2}^{**} \to {\tau_1}^{**} \to {\tau_2}^{**}$. The types of these constants are legal τ-types in L^{**}.

Intuitively, ν_τ is the type of *names of variables of type* τ, and *var* "casts" a ν_τ to its value, simulating the evaluation of a variable at the head of an application in a higher-order term. Similarly, *app* simulates application: given an abstraction

$\langle\nu_{\tau_1}\rangle{\tau_2}^{**}$ and a translated term of type ${\tau_1}^{**}$, application produces a term of type ${\tau_2}^{**}$.

The idea of the translation is to use the *var*, *app*, and *lam* constructors to represent ground λ-term structure, and use names, abstraction and concretion to represent subterms involving metavariables. In this translation, we assume that λ-calculus variables are the same kinds of symbols as names in nominal patterns.

$$\begin{array}{rlrl} c^{**} &= c & (t\ u)^{**} &= app(t^{**}, u^{**}) \\ {x_\tau}^{**} &= var(x_{\nu_\tau}) & X^{**} &= X \\ (\lambda x.t)^{**} &= \langle x\rangle t^{**} & (t\,\hat{}\,a)^{**} &= t^{**} \mathbin{@} a \end{array}$$

Example 2. Note that variable occurrences are treated differently depending on context: variables on the left-hand side of a flexible application $(-)\,\hat{}\,(-)$ are left alone, while others are encapsulated in a $var(-)$-constructor which casts a variable name of type ν_τ to an expression of type τ^{**}. Thus, the translation of $\lambda x, y.c\ (F\,\hat{}\,x\,\hat{}\,y)$ is $\langle x\rangle\langle y\rangle app(var(c), F \mathbin{@} x \mathbin{@} y)$, where $F : \tau_1 \rightarrow \tau_2 \rightarrow \delta$ in the former is mapped to $F : \langle\nu_{\tau_1}\rangle\langle\nu_{\tau_2}\rangle\delta$ in the latter.

The translation preserves well-formedness and is invertible; these facts are easy to show by induction.

Proposition 2. *If $x \notin FV(t)$ then $x^{**} \notin FN(t^{**})$. If $\Gamma \vdash t \uparrow \tau$ where t is a higher-order pattern, then $\Gamma^{**} \vdash t^{**} \uparrow \tau^{**}$. Similarly, if $\Gamma \vdash t \downarrow \tau$ or $\Gamma \vdash t \Downarrow \tau$, then $\Gamma^{**} \vdash t^{**} \downarrow \tau^{**}$. Also, the translation has an inverse $(-)^{\dagger\dagger}$.*

As observed by Miller, in a ηlβn higher-order pattern unification problem, the only kinds of redices that occur are β_0 redices. Since the $\beta_0\eta$ theory is simulated by the $\beta_\alpha\eta_\alpha$ theory in nominal patterns, higher-order pattern unification is equivalent to nominal pattern unification.

Theorem 3. *A higher-order pattern unification problem $t \approx ?\ u$ in ηlβn-normal form has a solution if and only if its translation $t^{**} \approx ?\ u^{**}$ has a nominal pattern unifier.*

Proof. For the forward direction, suppose that t, u are normalized and have a higher-order pattern unifier θ, so that $\theta(t) \approx \theta(u)$ up to ηlβn-normalization. Moreover, this normalization process can only involve β_0-redices, because there are no metavariables of extensional function types in t, u (as argued above). Let $\theta^{**} = [X := \theta(X)^{**} \mid X \in Dom(\theta)]$ be the translation of θ. Following a similar argument to the one used in Theorem 2, β_0-normalization can be simulated in the nominal pattern calculus via β_α-normalization. Thus, θ^{**} is a nominal pattern unifier of $t^{**} \approx ?\ u^{**}$.

The reverse direction is similar. Since only β_α-redices can be introduced in a nominal pattern unifier θ for $t^{**} \approx ?\ u^{**}$, we can use the reverse translation $(-)^{\dagger\dagger}$ to translate θ to a higher-order pattern unifier $\theta^{\dagger\dagger}$.

6 Nominal Pattern Unification as Nominal Unification

In this section, we show how to reduce nominal pattern unification to nominal unification. This is not as trivial as it sounds, for nominal patterns include the concretion operation not found in ordinary nominal terms, and so nominal pattern unification is not an immediate special case of nominal unification. In addition, nominal unification permits metavariables to be instantiated with terms containing free names, whereas nominal pattern unifiers must be closed.

We deal with the second problem first. Given a problem P with metavariables $\overline{X}$ and names $\overline{a}$, let $\#(\overline{a}, \overline{X}) = \{a \# X \mid a \in \overline{a}, X \in \overline{X}\}$. This set of constraints ensures that no name mentioned in P can appear free in any substitution for P's metavariables. Moreover, the nominal unifiers produced by Urban et al.'s algorithm only involve the names mentioned in the original problem.

Concretion can be eliminated from nominal pattern unification problems as follows. If $t \mathbin{@} a$ is a subterm of a nominal pattern unification problem $P[t \mathbin{@} a]$, then that problem is equivalent to the problem $P[Y], \langle a\rangle Y \approx ?\ t$, where Y is a fresh metavariable. This is because we know that a must be fresh for t (because of the well-formedness constraint) and so by the η-rule, we know that t can always be expressed as $\langle a\rangle Y$ for some value Y; this is precisely the value denoted by $t \mathbin{@} a$.

Given a nominal pattern unification problem P over names $\overline{a}$ and variables $\overline{X}$, we write $P^{\#}$ for the result of eliminating concretions from P and adding the freshness constraints $\#(\overline{a}, \overline{X})$. We claim that $P^{\#}$ and P are equivalent problems, and in addition that the answer to P can be computed from that of $P^{\#}$. However, the final step in this process is complicated, so we will illustrate it via examples first.

Example 3. Consider the problem $\langle a\rangle X \approx ?\ \langle a\rangle Y \mathbin{@} a$. In this case the translation is

$$\#(\{a\}, \{X, Y\}), Y \approx \langle a\rangle Y', \langle a\rangle X \approx \langle a\rangle Y'$$

The nominal unifier is $a \# X, Y = \langle a\rangle X$. Since $a \# X$, and a is the only name in scope, $Y = \langle a\rangle X$ is a nominal pattern unifier for the original problem.

Example 4. The translation of the problem $\langle a\rangle\langle b\rangle X \mathbin{@} a \mathbin{@} b \approx ?\ \langle a\rangle\langle c\rangle Y \mathbin{@} c \mathbin{@} a$ is (after some trivial simplifications)

$$\#(\{a, b\}, \{X, Y\}), X \approx \langle a\rangle\langle b\rangle X', Y \approx \langle c\rangle\langle a\rangle Y', \langle a\rangle\langle b\rangle X' \approx \langle a\rangle\langle c\rangle Y'$$

The most general unifier is $b \# Y', X \approx \langle a\rangle\langle b\rangle(b\ c) \cdot Y', Y \approx \langle a\rangle\langle c\rangle Y'$. Since $b \# Y'$, we know that Y' can only depend on a and c, so there must be a Z such that $Y' = Z \mathbin{@} a \mathbin{@} c$, where Z is a nominal pattern metavariable (i.e., can be substituted only with closed patterns). Solving for X, Y in terms of Z, we obtain $X = \langle a\rangle\langle b\rangle(a\ b) \cdot (Z \mathbin{@} a \mathbin{@} c) = \langle a\rangle\langle b\rangle Z \mathbin{@} a \mathbin{@} b$, $Y = \langle a\rangle\langle c\rangle Z \mathbin{@} a \mathbin{@} c$; this is a nominal pattern unifier of the original problem.

Remark 1. There is a minor hitch in this argument, due to the fact that there may be types that have no closed terms. For example, if data type δ consists only of terms of the form $v(a)$ for names a_ν and $v : \nu \to \delta$, then the unification problem $X \approx ? X$, where $X : \delta$, has no solution among closed terms, but it does have a nominal unifier.

This is similar to the difficulty in ordinary (typed) unification in the presence of possibly-void types. It is customary to either ignore this problem or assume that all types have at least one (closed) term. In our case, it is decidable (for finite signatures) whether each σ-type possesses any closed terms. We call such types *nonvoid*. For example, ν and δ (where δ is as in the previous paragraph) obviously possesses no closed terms, while $\langle\nu\rangle\nu$ and $\langle\nu\rangle\delta$ are nonvoid. Moreover, a substitution θ is called nonvoid if all the metavariables mentioned in its range are nonvoid.

Theorem 4. *If P is satisfiable then its translation $P^{\#}$ is satisfiable. Furthermore, if $P^{\#}$ is satisfiable then its unifier can be translated to a substitution which unifies P if and only if it is nonvoid.*

Proof. For the forward direction, clearly if θ satisfies P then θ satisfies each constraint in $\#(\overline{a}, \overline{X})$. In addition, it is easy to show that $\#(\overline{a}, \overline{X}), P[t \mathbin{@} a]$ is satisfiable if and only if $\#(\overline{a}, \overline{X}), P[Y], \langle a\rangle Y \approx ? t$ is satisfiable; thus, by induction if P is satisfiable then so is $P^{\#}$.

For the reverse direction, suppose ∇, θ is the most general nominal unifier for $P^{\#}$. Let $\overline{a}$ be the names of P. Suppose that the free variables in ∇, θ are $\overline{Y}$. For each $Y \in \overline{Y}$, there is a list of names $\overline{a_{Y_i}}$ such that $a \in \overline{A}$ but $a \# Y_i \notin \nabla$. Thus, we have $Y_i = Z_i \mathbin{@} \overline{a_{Y_i}}$ for some fresh metavariables Z_i such that $\overline{a} \# Z_i$. If we make this substitution, then we obtain a nominal pattern possibly involving swappings, but these swappings can be eliminated since each Z_i satisfies $(a_i\ a_j) \cdot Z \approx Z$. This produces the desired substitution θ'. If θ' is nonvoid, i.e. each σ_i is nonvoid for $Z_i : \sigma_i$, then each Z_i can be replaced with a closed term 0_{σ_i} to obtain a satisfying valuation for P. Conversely, it is not difficult to show that if θ' is not nonvoid, then P is unsatisfiable, since (by the first part) any valuation satisfying P can be used to construct a valuation satisfying $P^{\#}$, which would have to be an instance of ∇, θ because it is most general. Closed instantiations of all the Z_i could be extracted from such a valuation.

7 Related work

As discussed by Urban et al. [15], nominal unification can apparently be reduced to higher-order pattern unification, but it is difficult to see how to translate the resulting higher-order pattern unifier to a nominal unifier. Nevertheless, such a translation is of interest because if nominal unifiers can be extracted from higher-order pattern unifiers in linear time, this would give a linear algorithm for nominal unification. We believe that it would be equivalent (and notationally simpler) to investigate the reduction of full nominal unification to nominal pattern unification.

Miller [8] showed that higher-order unification problems (and higher-order logic programs) can be translated to logic programs in L_λ [7], a logic programming language based on higher-order pattern unification. This reduction takes advantage of hereditary Harrop goals and clauses featured in L_λ. We believe that a similar reduction could be performed in a nominal logic programming language that provides hereditary Harrop goals and program clauses. While such features are present in the current implementation of the nominal logic programming language αProlog, the semantics of goals of the form $\forall x.G$ and $D \supset G$ have not been studied carefully yet for nominal logic programming. This question, and more generally, the question of whether L_λ programs can be translated to nominal logic programs (or vice versa) is of interest.

Hamana [4] has investigated the problem of unification modulo the β_0-rule for *binding algebra terms* [2]. Such terms are similar to nominal or higher-order patterns except that the lists of names supplied to metavariables may include repeated names. As a result, unification appears to require some searching for suitable renamings, and most general unifiers appear not to be unique. Obviously, this is a special case of higher-order unification, but it appears to be at worst of nondeterministic polynomial time complexity (since one can guess a sequence of appropriate renamings to find a unifier in polynomial time). We are interested in seeing whether this form of unification can also be implemented via nominal logic programming using Miller's approach.

Finally, we are interested in combining nominal and higher-order unification, or more generally, developing *nominal equational unification* techniques that include higher-order unification, Hamana's β_0-unification, structural equivalence in the π-calculus, and other equational theories involving name-binding as special cases. We believe that nominal equational unification techniques would be extremely useful for programming, prototyping, and formalizing programming languages, logics, and type systems.

8 Conclusion

We have shown that higher-order pattern unification can be reduced to nominal unification via an intermediate language of *nominal patterns.* This shows that any computation that can be performed using higher-order pattern unification can also be performed using nominal unification. It also shows that higher-order patterns are not just an ad-hoc invention of interest for efficiency reasons, but that they can be given formal status using nominal logic: in particular, semantic models of binding syntax for higher-order patterns can be constructed using the same techniques as for nominal logic.

Previous work has been focused on determining whether nominal unification is really "new" (that is, whether it is trivially reducible to higher-order pattern matching). We agree with Urban et al. that while it may not be new, there are good reasons for studying nominal unification directly rather than through the lens of higher-order unification. Moreover, our experience has been that nominal unification is much closer to first-order unification and considerably simpler to explain and implement than higher-order pattern unification (compare Urban et al. [15] to treatments such as Miller [7, 8], Nipkow [11], Dowek et al. [1], or Hamana [5]). This is not meant as a criticism of these works! Instead, our point is that even if one does *not* believe that nominal techniques are worth investigating as an *alternative* to higher-order abstract syntax, we believe that they are of value as an aid to *understanding* higher-order abstract syntax, particularly higher-order patterns.

References

1. G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Unification via explicit substitutions: The case of higher-order patterns. Technical Report Rapport de Recherche 3591, INRIA, December 1998.
2. M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In Giuseppe Longo, editor, *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, pages 193–202, Washington, DC, 1999. IEEE, IEEE Press.
3. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.
4. Makoto Hamana. A logic programming language based on binding algebras. In *Proc. Theoretical Aspects of Computer Science (TACS 2001)*, number 2215 in Lecture Notes in Computer Science, pages 243–262. Springer-Verlag, 2001.
5. Makoto Hamana. Simple β_0-unification for terms with context holes. In *Proceedings of the 16th International Workshop on Unification (UNIF 2002)*, pages 9–13, 2002.
6. Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–67, 1975.
7. Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic and Computation*, 1(4):497–536, 1991.
8. Dale Miller. Unification of simply typed lambda-terms as logic programming. In Koichi Furukawa, editor, *Logic Programming, Proceedings of the Eighth International Conference*, pages 255–269, Paris, France, June 24–28 1991. MIT Press.
9. Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, 1992.
10. Dale Miller and Alwen Tiu. A proof theory for generic judgments: extended abstract. In *Proc. 18th Symp. on Logic in Computer Science (LICS 2003)*, pages 118–127. IEEE Press, 2003.
11. Tobias Nipkow. Functional unification of higher-order patterns. In *Proc. 8th IEEE Symp. Logic in Computer Science*, pages 64–74, 1993.
12. A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Proc. 5th Int. Conf. on Mathematics of Programme Construction (MPC2000)*, number 1837 in Lecture Notes in Computer Science, pages 230–255, Ponte de Lima, Portugal, July 2000. Springer-Verlag.
13. Zhenyu Qian. Linear unification of higher-order patterns. In *Proceedings of the International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 391–405. Springer-Verlag, 1993.
14. M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programmming with binders made simple. In *Proc. 8th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP 2003)*, pages 263–274, Uppsala, Sweden, 2003. ACM Press.
15. C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1–3):473–497, 2004.