Edinburgh Research Explorer

Nominal Matching Logic

Nominal Matching Logic

James Cheney
Laboratory for Foundations of Computer Science,
University of Edinburgh
Edinburgh, United Kingdom
jcheney@inf.ed.ac.uk

Maribel Fernández
Department of Informatics,
King's College London
London, United Kingdom
Maribel.Fernandez@kcl.ac.uk

ABSTRACT

We introduce Nominal Matching Logic (NML) as an extension of Matching Logic with names and binding following the Gabbay-Pitts nominal approach. Matching logic is the foundation of the $\mathbb{K}$ framework, used to specify programming languages and automatically derive associated tools (compilers, debuggers, model checkers, program verifiers). Matching logic does not include a primitive notion of name binding, though binding operators can be represented via an encoding that internalises the graph of a function from bound names to expressions containing bound names. This approach is sufficient to represent computations involving binding operators, but has not been reconciled with support for inductive reasoning over syntax with binding (e.g., reasoning over λ-terms). Nominal logic is a formal system for reasoning about names and binding, which provides well-behaved and powerful principles for inductive reasoning over syntax with binding, and NML inherits these principles. We discuss design alternatives for the syntax and the semantics of NML, prove meta-theoretical properties and give examples to illustrate its expressive power. In particular, we show how induction principles for λ-terms (α-structural induction) can be defined and used to prove standard properties of the λ-calculus.

CCS CONCEPTS

• **Theory of computation → Logic and verification**; **Lambda calculus**; **Equational logic and rewriting**.

KEYWORDS

Binding operator, Matching Logic, Nominal Logic, Lambda-Calculus, Verification

1 INTRODUCTION

Nominal logic is a formal system for reasoning about names and name-binding, building on the Gabbay-Pitts approach to name-binding founded on the concepts of permutation groups acting on sets, freshness, and finite support [Gabbay and Pitts 2001; Pitts 2003]. Its semantics and proof theory have been extensively investigated and provide a foundation for nominal extensions to rewriting [Fernández and Gabbay 2007], functional programming [Shinwell et al. 2003], logic programming [Cheney and Urban 2008a], and specification testing [Cheney and Momigliano 2017].

For example, the syntax of the λ-calculus can be specified using symbols var, lam and app (to represent variables, λ-abstraction and application respectively); a term of the form $\lambda x.e$ is represented as $lam([x]\mathbf{e})$ where $\mathbf{e}$ is the representation of e and $[x]\mathbf{e}$ is a nominal abstraction. The α-equivalence relation between λ-terms holds by construction, thanks to the properties of the abstraction construct

in nominal logic. Capture-avoiding substitution can be formally defined in nominal logic in a way that resembles the familiar informal definition (see Example 2.1).

Nominal logic also provides well-behaved and powerful principles for inductive reasoning over abstract syntax with binding, as supported in Nominal Isabelle [Urban 2008]. However, using nominal logic for specification and reasoning is sometimes awkward due to the partial or nondeterministic nature of some operations involving names: for example the nondeterministic *fresh name* operation which chooses a fresh name, or the partial *concretion* operation which renames a bound, abstracted name to a fresh one. These operations are difficult to work with directly in first-order logic where all function symbols must denote total, deterministic functions.

Matching logic is a formal system for reasoning about patterns, which may fail to match (partiality) or match multiple ways (nondeterminism). It has been introduced as a formal foundation for modeling and reasoning about operational semantics in the language framework $\mathbb{K}$ [Rosu 2017], which has been used to develop semantics for a number of programming languages.

Matching logic does not contain built-in support for binding, which limits its appeal for modeling high-level languages, such as functional programming languages based on the λ-calculus. Recently, matching logic has been extended with facilities for modelling and reasoning about binding by Chen and Rosu [2020], for example to capture the equational theory of the lambda-calculus or model other binding constructs. In essence, the approach taken is to define binding by internalizing the notion of the graph of a function (e.g. a mapping from bound names to expressions that might contain the bound name). However, as Chen and Rosu note, it is not yet clear how to extend this approach further to support inductive reasoning over syntax with binding.

As noted above, both nominal logic and matching logic have strengths and weaknesses. Nominal logic provides support for inductive reasoning over syntax with binding, but reasoning about instantiating abstractions with specific choices of names, or choices of fresh names, often involves partiality or nondeterminism issues that necessitate abandoning a calculational style of proof in nominal logic. On the other hand, matching logic provides support for partiality and non-determinism but does not support inductive reasoning over syntax with binding. Since nominal logic is definable as an ordinary first-order theory, it may be considerably easier to incorporate into matching logic and implementations such as $\mathbb{K}$ than alternative approaches to binding.

In this paper we show that matching logic and nominal logic are indeed compatible, and show how they can be combined in a single system that generalizes each of them. Thus, a wealth of existing nominal techniques for reasoning about languages with names

and binding can be imported into matching logic. We discuss two approaches to combine matching logic and nominal logic. In the first approach, the axioms of nominal logic are specified as a theory in matching logic, obtaining a system we call NLML. We show that NLML has the expected properties, in particular, it allows us to use concepts such as freshness of names and abstraction in matching logic specifications. Morever, existing proof systems and implementations of matching logic can be used to reason or execute specifications with binding operators. However, NLML does not include a primitive notion of name, and the Ⅵ quantifier of nominal logic cannot be used in NLML patterns. To address these shortcomings, in the second approach we extend the syntax of matching logic with a distinguished category of names and a Ⅵ constructor for patterns, and provide a semantics for these extensions that is compatible with the standard matching logic semantics. This also avoids the need to include nominal logic as a theory in matching logic.

We first review basic concepts of nominal logic and matching logic and their semantics (Sec. 2). We then briefly present NLML before introducing *Nominal Matching Logic* (NML), a single system that incorporates the capabilities of nominal logic for specifying and reasoning about name-binding into matching logic (Sec. 3). We illustrate the applications of NML via standard lambda-calculus examples (Sec. 4). The simplicity of this approach offers potential advantages over previous approaches to specify binders in matching logic, for example, there is no theoretical obstacle to adding fixed point operators to NML to support reasoning by induction about structures involving names and binding. To substantiate this point we provide a detailed comparison with Chen and Rosu's approach to binding in Applicative Matching Logic (AML) (Sec. 5).

The question of how to model and reason about binding in formalizing programming languages, logics and calculi has a long history, and nominal abstract syntax is but one of many approaches. This paper focuses somewhat narrowly on the question how to augment matching logic with support for binding, and our claim is that nominal logic (with some adjustment) is a promising candidate for doing this. We do not claim that this sets a new standard for reasoning about binding syntax that is superior to existing systems such as Nominal Isabelle [Urban 2008], locally nameless [Aydemir et al. 2008], Abella [Baelde et al. 2014a], Beluga [Pientka 2010], etc., and it seems premature to compare our work so far (which is mostly theoretical and not yet implemented) with established, mature systems that are already widely used for formalizing metatheory. We only claim that if we wish to extend matching logic with binding, nominal logic seems better suited than other approaches. We discuss how our work fits into the broader landscape in more detail in Section 6.

All proofs are relegated to the appendix.

2 BACKGROUND

We take for granted familiarity with standard sorted first-order logic syntax and semantics. In this section we recall the syntax and semantics of matching logic and nominal logic. In both cases, a signature $\Sigma = (S, \mathcal{V}ar, \Sigma)$ specifies S a set of sorts, $\mathcal{V}ar$ a sort-indexed family of countably many variables for each sort, and Σ

a family of sets $\Sigma_{\tau_1,\ldots,\tau_n;\tau}$ indexed by $S^* \times (S \cup \{Pred\})$, assigning symbols σ their input and output sorts. For convenience and consistency between nominal and matching logic we assume that sorts are closed under finite products, that is, $\Pi_{i=1}^{n}\tau_i$ is a sort whenever $\tau_1, \ldots, \tau_n$ are sorts, with associated function symbols for constructing n-tuples $(\_, \ldots, \_) \in \Sigma_{\tau_1,\ldots,\tau_n;\Pi_{i=1}^{n}\tau_i}$ and for projecting $\pi_j^n : \Pi_{j=1}^{n}\tau_j; \tau_i$. In first-order (and nominal) logic Σ also needs to specify the sorts of relation symbols p, and we use the special sort *Pred* and write $p \in \Sigma_{\tau_1,\ldots,\tau_n;Pred}$ to indicate this.

2.1 Nominal Logic

Nominal logic was introduced by Pitts as a sorted first-order theory [Pitts 2003]. In nominal logic, signatures are augmented by distinguishing some sorts as *name sorts*, and sorts include a construction called *abstraction* that builds a new sort $[\alpha]\tau$ from a name sort α and any sort τ. Nominal logic in general allows for the possibility of multiple name sorts, but we will consider the case of a single name-sort for simplicity, since handling the general case requires additional bureaucracy that is not needed to illustrate the main points. Abstractions in $[\alpha]\tau$ correspond to elements of τ with a distinguished *bound* name. The signature of any instance of nominal logic includes function symbols $(- -) \cdot - : \alpha \times \alpha \times \tau \to \tau$ and $[-]- : \alpha \times \tau \to [\alpha]\tau$ denoting name swapping and abstraction, for any name sort α and sort τ, and atomic formulas for equality ($=$) at any sort and freshness ($\#$) relating any name sort and any sort. Finally, nominal logic includes an extra quantifier, $\mathsf{Ⅵ}a.\phi$, which is pronounced "for fresh a, ϕ holds."

Pitts gives a set of axioms and axiom schemas that describe the behavior of these constructs, see Figure 1. The S axioms describe the behavior of swapping, while axioms F and A characterize freshness and abstraction respectively. The E axioms ensure equivariance, that is, that swapping preserves the behavior of function and atomic predicate symbols. Finally axiom scheme (Q) characterizes the Ⅵ-quantifier. Aside from Ⅵ, the axioms are standard first-order axioms, and (Q) can be viewed as a recipe for defining Ⅵ in terms of other formulas. Rather than discuss each axiom in detail, we review the semantics of nominal logic which validates them.

Nominal sets (cf. [Pitts 2013]) provide semantics to nominal logic. The characteristic feature of nominal sets is the use of *name permutations* acting on sets. Let G be the symmetric group $Sym(\mathbb{A})$ on some countable set $\mathbb{A}$ of atoms, which is generated by the swappings $(a\ b)$ where $a, b \in \mathbb{A}$. A *G-set* is a structure $(X, \cdot)$ equipped with carrier set X and an action of G on X, i.e. a function $\pi, x \mapsto \pi \cdot x$ satisfying the laws (1) $id \cdot x = x$ and (2) $(\pi \circ \pi') \cdot x = \pi \cdot \pi' \cdot x$. The *support* of an element of G is the set of atoms not fixed by π, i.e. $supp(\pi) = \{a \in \mathbb{A} \mid \pi(a) \neq a\}$. A function or relation on *G*-sets is called *equivariant* if it commutes with the permutation action: $\pi \cdot f(\bar{x}) = f(\pi \cdot \bar{x})$ or $R(\bar{x}) = R(\pi \cdot \bar{x})$.

G-sets and equivariant functions form a topos and so admit many standard constructions familiar from set theory, including products, sums (disjoint union), exponentials (function spaces), power sets, etc. For example, products of *G*-sets are formed by taking products of the underlying carrier sets and extending the group action pointwise.

A *support* of an element x of a *G*-set is a set S of atoms such that for all π with $supp(\pi) \cap S = \emptyset$ we have $\pi \cdot x = x$. A *nominal set*

$$(a\ a) \cdot x = x \qquad\qquad (S1)$$
$$(a\ a') \cdot (a\ a') \cdot x = x \qquad\qquad (S2)$$
$$(a\ a') \cdot a = a' \qquad\qquad (S3)$$
$$(a\ a') \cdot (b\ b') \cdot x = (((a\ a') \cdot b)\ ((a\ a') \cdot b')) \cdot (a\ a') \cdot x \qquad (E1)$$
$$b\ \#\ x \Rightarrow (a\ a') \cdot b\ \#\ (a\ a') \cdot x \qquad\qquad (E2)$$
$$(a\ a') \cdot f(\bar{x}) = f((a\ a') \cdot \bar{x}) \qquad\qquad (E3)$$
$$p(\bar{x}) \Rightarrow p((a\ a') \cdot \bar{x}) \qquad\qquad (E4)$$
$$(b\ b') \cdot [a]x = [(b\ b') \cdot a](b\ b') \cdot x \qquad\qquad (E5)$$
$$a\ \#\ x \wedge a'\ \#\ x \Rightarrow (a\ a') \cdot x = x \qquad\qquad (F1)$$
$$a\ \#\ a' \iff a \neq a' \qquad\qquad (F2)$$
$$\forall a : \alpha, a' : \alpha'.a\ \#\ a' \qquad (\alpha \neq \alpha') \qquad (F3)$$
$$\forall \bar{x}.\exists a.a\ \#\ \bar{x} \qquad\qquad (F4)$$
$$\forall \bar{x}.(\mathsf{V}a.\phi \iff \exists a.a\ \#\ \bar{x} \wedge \phi) \qquad (FV(\mathsf{V}a.\phi) \subseteq \bar{x}) \qquad (Q)$$
$$[a]x = [a']x' \iff (a = a' \wedge x = x')$$
$$\vee (a\ \#\ x' \wedge (a\ a') \cdot x = x') \qquad (A1)$$
$$\forall x : [\alpha]\tau.\exists a : \alpha, y : \tau.x = [a]y \qquad\qquad (A2)$$

Figure 1: Axioms of (classical) Nominal Logic

is a G-set in which all elements have a finite support. It is routine to show that this implies each element has a unique, least finite support which is denoted $supp(x) = \bigcap\{S \subseteq \mathbb{A} \mid \forall \pi.supp(\pi) \cap S = \emptyset \Rightarrow \pi \cdot x = x\}$. Nominal sets also form a topos [Pitts 2013], so also admit most standard set-theoretic constructions, except that functions and power sets must also be finitely supported (i.e. $X \to_{fin} Y$ and $\mathcal{P}_{fin}(X)$ consist of all functions/subsets having finite support).

Two special constructions on nominal sets are the set $\mathbb{A}$ of atoms, where $\pi \cdot a = \pi(a)$ and $supp(a) = \{a\}$, and the set of abstractions over $\mathbb{A}$ and X, written $[\mathbb{A}]X$. The latter is defined as the set of equivalence classes of pairs $\mathbb{A} \times X$ by the following relation:

$$\langle a, x\rangle \equiv_\alpha \langle b, y\rangle \iff \forall c \notin supp(a, b, x, y).(a\ c) \cdot x = (b\ c) \cdot y$$

Swapping applies to such equivalence classes pointwise, that is, $\pi \cdot E = \{\langle \pi \cdot a, \pi \cdot x\rangle \mid \langle a, x\rangle \in E\}$, and $supp([a]x) = supp(x) - \{a\}$. It is again a standard result that $\mathbb{A}$ and $[\mathbb{A}]X$, as defined above, are nominal sets if X is.

Abstract syntax with binding operators can be directly represented using the *abstraction* construct of nominal logic as shown in the example below for the λ-calculus.

Example 2.1. To represent the syntax of the λ-calculus, it is sufficient to use a signature including function symbols var, lam and app (to represent variables, λ-abstraction and application respectively), and sorts Var, Exp where Var is a name sort and Exp is equipped with the following constructors:

$$var : Var \to Exp \quad app : Exp \times Exp \to Exp \quad lam : [Var]Exp \to Exp$$

A term of the form $\lambda x.e$ is represented as $lam([x]\mathbf{e})$ where $\mathbf{e}$ is the representation of e. The α-equivalence relation between λ-terms holds by construction, thanks to the properties of the abstraction construct in nominal logic. Likewise, for example, the substitution operation $subst : Exp \times Var \times Exp \to Exp$ can be axiomatized as

follows:

$$subst(var(x), x, z) = z \qquad (Subst1)$$
$$x \neq y \Rightarrow subst(var(x), y, z) = var(x) \qquad (Subst2)$$
$$subst(app(x_1, x_2), y, z) =$$
$$app(subst(x_1, y, z), subst(x_2, y, z)) \qquad (Subst3)$$
$$a\ \#\ y, z \Rightarrow subst(lam([a]x), y, z) =$$
$$lam([a]subst(x, y, z)) \qquad (Subst4)$$

Here, the first three equations are ordinary first-order (conditional) equations, while the fourth specifies how substitution behaves when a λ-bound name is encountered: the name in the abstraction is required to be fresh, and when that is the case, the substitution simply passes inside the bound name. Even though the $(Subst4)$ axiom is a conditional equation, the specified substitution operation is still a total function, because any abstraction denotes an alpha-equivalence class that contains at least one sufficiently fresh name so that the freshness precondition holds. Other cases where the name is not sufficiently fresh, or where we attempt to substitute for the bound name, do not need to be specified, instead they can be proved from the above axioms and an induction principle for λ-terms (see e.g. Urban [2008] or Section 4).

In the original form of nominal logic introduced by Pitts, there are no constants of "name" sorts: in fact, adding such constants makes nominal logic inconsistent, thanks to the equivariance axiom $(E3)$ (considering constants as a special case of 0-ary functions). The first system we will present, NLML (Section 3.1) is based on directly importing nominal logic into matching logic and so in NLML it is also the case that there are no name constants. However, having syntax for ground names (sometimes called "atoms" or "name constants") is attractive in many practical situations, such as nominal unification, rewriting, and logic programming. For example, in standard nominal unification and matching algorithms, certain arguments such as a, b in $(a\ b) \cdot t$, $a\ \#\ t$ and $[a]t$, are required to be ground, and this is critical for efficiency and for most general unifiers to exist. Also, in logic programming the semantics of programs is defined in terms of *Herbrand models* built out of ground terms, and this does not work for Pitts' formulation of nominal logic because there are no ground terms involving names. Nominal logic was modified by Cheney [2006, 2016] to allow for ground names a, b, ... as an additional syntactic class which can be bound by the V-quantifier. We will adopt a similar approach later in the paper in the NML system (Section 3.2).

2.2 Matching Logic

There are several different recent presentations of variations of matching logic [Chen and Rosu 2019, 2020; Roşu 2017]. Our presentation is closest to that of Chen and Rosu [2019], though we believe our approach could be adapted to other systems such as Applicative Matching Logic (AML) [Chen and Rosu 2020] without problems. The most important difference between these systems is that in AML, there are no sorts governing the syntax, and instead sorts are internalized into the logic as constants. The presentation of Chen and Rosu [2019] uses a standard multisorted syntax and this presentation is closer to Pitts's presentation of Nominal Logic, so we follow this approach.

We consider a matching logic signature $\Sigma = (S, \mathcal{V}ar, \Sigma)$ as above, which we assume contains operations for product sorts and $Pred$.

In matching logic, unlike (say) first-order logic, there is only one syntactic class of *patterns*:

$$\phi_\tau ::= x\colon \tau \mid \phi_\tau \wedge \psi_\tau \mid \neg\phi_\tau \mid \exists x\colon \tau'.\phi_\tau \mid \sigma(\phi_{\tau_1}, \ldots, \phi_{\tau_n})$$

where in the first case $x \in \mathcal{V}ar_\tau$ and in the last case $\sigma \in \Sigma_{\tau_1,\ldots,\tau_n;\tau}$. Patterns generalize both terms and first-order logic formulas. We describe their semantics informally before providing the formal semantics. The meaning of a pattern of sort τ is a set of elements of sort τ that the pattern matches. The variable pattern $x\colon \tau$ matches just one element, the value of x. Conjunction corresponds to intersection of matching sets and negation corresponds to complementation. Existential patterns $\exists x\colon \tau'.\phi_\tau$ match those elements of τ for which there exists a value of x of sort τ' making the pattern ϕ_τ match. Finally, symbol patterns $\sigma(\phi_{\tau_1}, \ldots, \phi_{\tau_n})$ match as follows. Each symbol is interpreted as an arbitrary relation between n input arguments and an output (or equivalently, as a function from the inputs to a set of possible matches). Note that such symbols do not necessarily correspond to functions: they may be undefined, and there may be multiple matching values for a given choice of inputs. We assume there is a distinguished sort $Pred$ and unary symbols $(-)_\tau^\dagger \in \Sigma_{Pred;\tau}$ for every sort τ, called *coercion*, as well as equality symbols $=_\tau$ at every sort τ. Other patterns (disjunction $\phi_1 \vee \phi_2$, implication $\phi_1 \Rightarrow \phi_2$, universal quantification, true and false) can be defined as abbreviations, for example, $\top_\tau \stackrel{\triangle}{=} \exists x\colon \tau.x\colon \tau$ (since carriers are non-empty sets) and $\bot_\tau \stackrel{\triangle}{=} \neg\top_\tau$. Subscripts indicating the sort at which an operation is used are omitted when clear from context.

Formally, the semantics of patterns is defined as follows. A matching logic model $M = (\{M_\tau\}_{\tau \in S}, \{\sigma_M\}_{\sigma \in \Sigma})$ consists of a non-empty carrier set M_τ for each sort $\tau \in S$ and an interpretation $\sigma_M\colon M_{\tau_1} \times \ldots \times M_{\tau_n} \to \mathcal{P}(M_\tau)$ for each $\sigma \in \Sigma_{\tau_1,\ldots,\tau_n;\tau}$. A *valuation* ρ is a function $\rho\colon \mathcal{V}ar \to M$ that respects sorts so that $\rho(x) \in M_\tau$ for each $x \in \mathcal{V}ar_\tau$.

Given $\Sigma = (S, \mathcal{V}ar, \Sigma)$, a matching logic Σ-model M and valuation ρ, the extension $\|-\|_\rho$ to patterns is defined by: $\|x\|_\rho = \{\rho(x)\}$ for all $x \in \mathcal{V}ar$, $\|\phi_1 \wedge \phi_2\|_\rho = \|\phi_1\|_\rho \cap \|\phi_2\|_\rho$, $\|\neg\phi_\tau\|_\rho = M_\tau - \|\phi_\tau\|_\rho$, $\|\exists x\colon \tau'.\phi_\tau\|_\rho = \bigcup_{a \in M_{\tau'}} \|\phi_\tau\|_{\rho[a/x]}$, $\|\sigma(\phi_{\tau_1}, \ldots, \phi_{\tau_n})\|_\rho = \overline{\sigma_M}(\|\phi_{\tau_1}\|_\rho, \ldots, \|\phi_{\tau_n}\|_\rho)$, for $\sigma \in \Sigma_{\tau_1,\ldots,\tau_n;\tau}$, where we write $\overline{\sigma_M}$ to denote the pointwise extension of σ_M, i.e. $\overline{\sigma_M}(V_1, \ldots, V_n) = \bigcup\{\sigma_M(v_1, \ldots, v_n) \mid v_1 \in V_1, \ldots, v_n \in V_n\}$.

A pattern ϕ_τ is valid in M, written $M \vDash \phi$, if $\|\phi_\tau\|_\rho = M_\tau$ for all valuations $\rho\colon \mathcal{V}ar \to M$. If Γ is a set of patterns (called axioms), then $M \vDash \Gamma$ if $M \vDash \phi$ for each $\phi \in \Gamma$ and $\Gamma \vDash \phi$ if $M \vDash \phi$ for all $M \vDash \Gamma$. The pair (Σ, Γ) is a matching logic theory, and M is a model of the theory if $M \vDash \Gamma$.

Again following Chen and Rosu [2019] we assume the sort $Pred$ is interpreted as a single-element set $M_{Pred} = \{\star\}$, thus, a predicate is true if its interpretation is M_{Pred} and false if its interpretation is $\emptyset$. The interpretation of the operation $(-)^\dagger \in \Sigma_{Pred;\tau}$ maps $\star$ to M_τ. That is, a true predicate is interpreted as $\{\star\}$ which can be coerced to the set of all patterns matching some other sort, while a false predicate is interpreted as $\emptyset$ which can be coerced to the empty pattern of sort τ. Coercions may be omitted when obvious from context. Equality $(=_\tau)$ predicates are interpreted as follows: $\|\phi = \psi\|_\rho = \{\star \mid \|\phi\|_\rho = \|\psi\|_\rho\}$. We may write $\phi \subseteq \psi$ as an abbreviation for $\phi \vee \psi = \psi$, and $x \in \phi$ for $x \subseteq \phi$ to emphasize that when

x is a variable it matches exactly one value. Some presentations of matching logic include other primitive formulas that are definable using equality and coercion, such as *definedness* $\lceil \phi \rceil_\tau$ (which can be defined as $\exists x\colon \tau.x \in^\dagger \phi$). Alternatively, the coercion operator can be defined using a primitive definedness symbol. The notation $\sigma\colon \tau \to \tau'$ (resp. $\sigma\colon \tau \rightharpoonup \tau'$) indicates that σ is a *function* (resp. *partial function*), in which case an appropriate axiom for σ is assumed to be included in the theory.

Variables in patterns can be substituted by patterns: $\phi[\psi/x]$ denotes the result of substituting ψ for every free occurrence of x in ϕ, where α-renaming happens implicitly to prevent variable capture. Substituting a pattern for a universally-quantified variable does not preserve validity in general, however, if ϕ is valid and ψ is functional, i.e., it evaluates to a singleton set, then $(\forall x.\phi) \Rightarrow \phi[\psi/x]$ is valid (see [Chen and Rosu 2019]).

Hilbert-style proof systems for matching logic are available: we refer to [Chen and Rosu 2019; Roşu 2017] for details.

3 NOMINAL MATCHING LOGIC

We now consider ways to combine nominal logic and matching logic, yielding a logical framework extending the advantages of matching logic for reasoning about language semantics with the advantages of nominal logic for reasoning about name binding.

3.1 Nominal Logic as a Matching Logic Theory

The first approach is simply to consider an instance of nominal logic (i.e., a signature and theory) as an instance of matching logic. Since any (sorted) first-order logic theory can be translated to a theory of matching logic, and any instance of nominal logic is a (sorted) first-order logic theory, it is clear we can define nominal logic as a theory in matching logic. However, some adjustments are needed, as with any translation of first-order logic into matching logic: we must specify that the function symbols of the nominal logic signature satisfy the *Function* axiom (i.e., denote functions) and predicate symbols are pattern symbols with result sort $Pred$. Also, all the axioms of nominal logic must be included as translated axioms in the matching logic theory. This is not as straightforward as it might sound, since the nominal logic axioms treat function symbols and relation symbols differently, whereas in matching logic there is no built-in distinction. First, we define the syntax of the nominal logic theory NL in matching logic.

3.1.1 Syntax. The nominal logic signature includes function symbols to represent swappings and abstraction, which map to symbols in the matching logic signature, interpreted using the axioms in nominal logic. Similarly, the equality and freshness relation symbols from nominal logic map to symbols in matching logic with appropriate interpretations.

Concretely, let Σ^{NL} be a nominal logic signature, thus including $(- \ -) \cdot - \colon \alpha \times \alpha \times \tau \to \tau$ and $[-]- \colon \alpha \times \tau \to [\alpha]\tau$ (name swapping and abstraction symbols, for any name sort α and sort τ) and equality $=\colon \tau \times \tau$ at any sort τ and freshness $\#\colon \alpha \times \tau$ for any name sort α and sort τ. We define the matching logic signature Σ^{NLML} by using the predicate sort $Pred$ and considering all relation symbols $R \colon \tau_1 \times \cdots \times \tau_n$ to be matching logic symbols in $\Sigma_{\tau_1 \times \cdots \times \tau_n;Pred}$. Otherwise, constants and function symbols of nominal logic are regarded as matching logic symbols in the obvious

way. To ensure that they are interpreted as (total, deterministic) functions in matching logic we also add instances of the function axioms:

$$\exists x.c = x \quad (Fun_c) \qquad \forall \bar{z}.\exists x.f(\bar{z}) = x \quad (Fun_f)$$

Any formula of the form $\pitchfork a.\phi$ in nominal logic can be replaced by $\exists a.a \,\#\, \bar{x} \wedge \phi$ preserving its truth value (axiom (Q) justifies this translation, see Figure 1). Therefore we do not consider formulas with $\pitchfork$ quantifiers yet (we revisit this decision in Section 3.2).

3.1.2 Axioms. All the nominal logic axioms (see Figure 1) may be viewed as matching logic axioms; however since there is no distinction between constants, function symbols and relation symbols in matching logic we can replace the various equivariance axiom schemes with a single scheme stating that all symbols in Σ^{NLML} are equivariant:

$$(a\ b) \cdot \sigma(\bar{x}) = \sigma((a\ b) \cdot \bar{x}) \qquad (EV)$$

There is a minor subtlety in dealing with equivariance for predicates, since for a predicate the above axiom scheme does not ensure that $R(\bar{x}) \implies R((a\ b) \cdot \bar{x})$. To recover axioms $(E2)$, $(E4)$ of nominal logic, we also include an axiom ensuring that there is just one (necessarily equivariant) value of sort $Pred$ (note that x in this axiom can only be interpreted by the single element of M_{Pred}):

$$\forall x : Pred.x = \top_{Pred} \qquad (P)$$

Note that although we assume $Pred$ is interpreted as a single-element set in all models, we need this axiom in order to use this fact in proofs.

Concretely the axioms are:

$$Ax_{NLML} = \{(S1), (S2), (S3), (EV), (P), (F1), (F2), (F3), (F4),$$
$$(A1), (A2), (Fun_f), (Fun_c)\}$$

where f and c are the function symbols and constants of the nominal logic signature. We write $NLML$ to refer to matching logic instantiated by (translated) nominal logic axioms as above.

3.1.3 Models. Let M be a matching logic Σ_{NLML}-model satisfying the axioms in Ax_{NLML}. Axiom $(F4)$ ensures that there are infinitely many atoms in the carrier sets of name sorts.

True in nominal logic maps to $\top$ of sort $Pred$, matching every element of the domain. False in nominal logic maps to $\bot$ of sort $Pred$, matching nothing.

Axioms $(S1)$–$(S3)$, (EV), (P) ensure that the interpretation of the swapping symbol $(-\ -) \cdot - \in \Sigma_{NLML}$ is a swapping operator: $(S1)$–$(S3)$ define swapping for atoms, and (EV), (P) define swapping for the interpretation of patterns built using function symbols or relation symbols. Axioms $(F1)$–$(F4)$ ensure that the freshness symbol $\#$ is interpreted as a freshness relation. Axioms $(A1)$, $(A2)$ define α-equivalence of abstraction patterns as expected.

We could further restrict the class of matching logic models considered by requiring that the carrier sets of sorts be nominal sets (i.e., elements have finite support), however, this is not strictly necessary. There are also interesting models of nominal logic where the carrier sets are not finitely-supported nominal sets (elements could have infinite support as long as axiom $(F4)$ is satisfied, that is, the support of an element cannot include all the atoms); see Cheney [2006] and Gabbay [2007] for examples.

3.1.4 Correctness of the representation of nominal logic. Given an ordinary nominal logic theory Γ, we write $ML_{NL}(\Gamma)$ for the theory obtained by translating formulas in Γ to matching logic patterns inductively and combining with the axioms of nominal logic as modified above. The translation of formulas is the same as translating FOL to matching logic (see Chen and Rosu [2019], section II.D). In the next propositions, provability for patterns is defined with respect to the complete Hilbert-style proof system for matching logic defined in previous work [Chen and Rosu 2019; Roşu 2017], and provability of nominal logic formulas with respect to the Hilbert-style proof system with Pitts' nominal logic axioms, which was shown complete in [Cheney 2006].

PROPOSITION 3.1. *The translations of $(E1)$–$(E5)$ are provable in matching logic from the axioms Ax_{NLML}.*

PROPOSITION 3.2. *There exists a translation from closed nominal logic formulas to NLML formulas, $\phi \mapsto \phi'$, such that $\phi' : Pred$ and:*

(1) *If ϕ is provable in nominal logic then $\phi' = \top$ in NLML.*
(2) *If ϕ is not provable in nominal logic then $\phi' = \bot$ in NLML.*

Although this approach to represent nominal logic in matching logic allows us to incorporate nominal features into matching logic with minimal effort, it restricts users to work within a specific theory and it does not exploit the full capabilities of matching logic to define partial and non-deterministic operators. However, while axioms (Fun_f) and (Fun_c) force function symbols and constants imported from NL to behave like functions, nothing stops us having non-deterministic or partial operations. We explore this possibility next.

3.1.5 Partial and nondeterministic operations. Matching logic supports function-like syntax for operations that are partial or multivalued (or both). There is a natural elimination form for abstraction in nominal logic, called concretion [Pitts 2013], which takes an abstraction t and a name a, and returns the body of the abstraction with the bound name replaced by a. Concretion is only defined when the name is fresh for the abstraction. Thus, in standard first-order logic concretion cannot be a function symbol, but in matching logic we can include concretion as a partial operator. Likewise in NL axiom $(F4)$ asserts that for any finite collection of values, there exists a name fresh for all of these values. In matching logic, we can directly define a freshness operator $fresh(-)$ which matches any name that is fresh for its argument.

The concretion operator is definable in NLML as follows, where the $=$ symbol is coerced from $Pred$ to the sort of y (recall that the interpretation of $\wedge$ is intersection).

$$x@a \triangleq \exists y.y \wedge ([a]y =^{\dagger} x) \qquad (*)$$

Let us walk through this definition to ensure its meaning is clear. Informally, concretion instantiates the abstracted name of x with a, but only if it is possible to safely rename to a. The right-hand side encodes this meaning in matching logic. First, we existentially quantify over some y which will stand for the body of the abstraction with the bound name renamed to a. The conjunction takes the intersection of the interpretations of the two patterns y and $[a]y =^{\dagger} x$. The pattern y simply matches y itself, while $[a]y =^{\dagger} x$ either matches everything (if there is an α-variant of x where a

is the bound name) or nothing. Thus we can read the pattern as "$x@a$ equals some y provided $[a]y = x$."

Thus, Axioms (A1) and (A2) can be reformulated using abstraction as follows:

$$\forall a, b : \alpha, x : [\alpha]\tau.([a]x)@b = ((a\ b) \cdot x \wedge b \,\#^{\dagger}\, [a]x) \quad (A1')$$

$$\forall x : [\alpha]\tau.x = \exists a.a \,\#^{\dagger}\, x \wedge [a](x@a) \quad (A2')$$

Axiom $(A1')$ asserts a conditional form of beta-equivalence: if b is fresh for $[a]x$ then $([a]x)@b = (a\ b) \cdot x$ (and in particular if $a = b$ then $([a]x)@b = x$), otherwise if the freshness assumption does not hold the concretion pattern matches nothing. Axiom $(A2')$ asserts eta-equivalence: any abstraction is equivalent to one where the bound name is fresh, and the body is concreted at that name. (This axiom can be written more concisely using the N-quantifier which we will consider later.) Note the use of a coercion operator $((-)^{\dagger})$ on $\#$, to transform $Pred$, the output of $\#$, into the sort $[\alpha]\tau$ of x so the intersection pattern $\wedge$ intersects sets of matching patterns at that sort.

These two axioms are equivalent to the classic nominal logic ones for abstraction:

PROPOSITION 3.3. *Axioms $(A1')$ and $(A2')$ are valid in NLML where concretion is defined using (*). Conversely, $(A1)$ and $(A2)$ follow from $(A1')$ and $(A2')$ in NLML without $(A1)$, $(A2)$.*

The freshness operation $fresh_{\tau;\alpha}(-)$ is definable as follows:

$$fresh_{\tau;\alpha}(x) \triangleq \exists a : \alpha.a \wedge a \,\#^{\dagger}\, x \quad (**)$$

This operation matches any name that is fresh for x. According to the semantics of patterns, $fresh(\phi)$, the pointwise extension of this operator, will therefore match any name that is fresh for something matching ϕ; thus perhaps counterintuitively if a, b are distinct names then $fresh(a \vee b) = \top$. To see why, note that $a \vee b$ denotes a two-element set $\{a', b'\}$. Every name is fresh for either a' or b' (or both), so all names are included in $\top$. Indeed, it follows from the semantics of operations that for any unary operation $\sigma(\phi \vee \psi) = \sigma(\phi) \vee \sigma(\psi)$, so clearly $fresh(a \vee b) = fresh(a) \vee fresh(b) = \neg a \vee \neg b = \neg(a \wedge b) = \neg\bot = \top$.

This definition satisfies the following axioms which are reformulated versions of $(F1)$–$(F4)$.

$$(fresh(x)\ fresh(x)) \cdot x = x \quad (F1')$$

$$fresh_{\alpha;\alpha}(a : \alpha) = \neg a \quad (F2')$$

$$fresh_{\alpha;\alpha'}(a : \alpha) = \top_{\alpha'} \quad (\alpha \neq \alpha') \quad (F3')$$

$$\forall x : \tau.\exists a : \alpha.a \in fresh(x) \quad (F4')$$

PROPOSITION 3.4. *Axioms $(F1')$–$(F4')$ are valid in NLML where $fresh(-)$ is defined using $(**)$. Conversely, axioms $(F1)$–$(F4)$ are provable from $(F1')$–$(F4')$.*

3.1.6 Assessment. We have now shown that nominal logic can be embedded as a first-order theory in matching logic, using the known embedding of FOL in matching logic, to obtain the combined system NLML. We have even shown that matching logic offers some facilities that combine nicely with needs of nominal logic that are not well-addressed in a conventional FOL setting, such as catering for partial or nondeterministic operations like concretion and fresh name matching. Are we done? Why not stop here?

We have three reasons for considering alternatives. First, as noted in Section 2, in other nominal settings such as rewriting, logic

programming and program verification, it has been found helpful to provide ground names instead of just variables of name sort (see [Cheney 2005; Cheney and Urban 2004; Fernández and Gabbay 2007; Pitts 2011; Urban et al. 2004]), and these are not available in NLML for the same reasons as in the original axiomatization of nominal logic. Second, the N-quantifier is a distinctive feature of nominal logic, and it is not clear whether, or how, it can be incorporated into matching logic in a first-class way, in particular, whether it can be made into a N-*pattern* that can be used anywhere in a pattern just like the other pattern connectives. Third, and related to the first two points, as we shall discuss in Section 4, typical reasoning in NLML usually requires considerable maintenance of freshness constraints, which can be mitigated using the N-quantifier. Therefore, in the next section we consider a second system called NML (Nominal Matching Logic) that adopts ground names and a N-quantifier as pattern constructs, where N binds names, and N can occur anywhere in a pattern, just like the other pattern constructs.

Note further that, as has been the case throughout the paper, we largely focus on syntax and semantics of the logics, and not on codifying their proof systems. For NLML this is no great loss since the established proof systems for matching logic can be used directly. In the NML system presented in the next section this is not literally possible since we are adding to the syntax of patterns. However as we shall see these additional patterns can be translated away if needed (although this has a cost in terms of freshness constraint maintenance and efficiency of unification algorithms); thus if desired the underlying proof system of matching logic can still be used. We conjecture that a nicer proof system combining the techniques of Chen and Rosu [2019] and Cheney [2016] could be developed but leave this for future investigation.

3.2 NML: Matching Logic with Names and N

Finally we consider a further extension of matching logic in which syntax for names is introduced and the freshness quantifier (N) is added as a general pattern construct and is given an interpretation directly in the semantics instead of being defined by an axiom scheme. We call this system Nominal Matching Logic (NML). Below we define the syntax and semantics of NML patterns and give some illustrative examples.

3.2.1 NML Syntax. A nominal matching logic signature Σ consists of $(S, \mathcal{V}ar, Name, \Sigma)$ where

- S is a non-empty set of sorts $\tau, \tau_1, \tau_2 \ldots$, split into a set NS of name sorts $\alpha, \alpha_1, \alpha_2, \ldots$, a set DS of data sorts $\delta, \delta_1, \delta_2, \ldots$ including a sort $Pred$, and a set AS of abstraction sorts $[\alpha]\tau$ — there is one abstraction sort for each pair α, τ,
- $\mathcal{V}ar$ is a S-indexed $\{\mathcal{V}ar_\tau \mid \tau \in S\}$ of countable sets of variables $x : \tau, y : \tau, \ldots$,
- $Name$ is an NS-indexed family $\{Name_\alpha \mid \alpha \in NS\}$ of countable sets of names $a : \alpha, b : \alpha, \ldots$ and
- Σ is a $(S^* \times S)$-indexed family of sets of many-sorted symbols σ, written $\Sigma_{\tau_1,\ldots,\tau_n;\tau}$.

Definition 3.5 (NML Syntax). The syntax of matching logic patterns is extended by including a distinguished category of names

of name sort and a И pattern as follows:

$$\phi_\tau \quad ::= \quad x:\tau \mid \mathsf{a}:\alpha \mid \phi_\tau \wedge \psi_\tau \mid \neg\phi_\tau \mid \exists x{:}\tau'.\phi_\tau \mid$$
$$\sigma(\phi_{\tau_1}, \ldots, \phi_{\tau_n}) \mid \mathsf{И}\mathsf{a}{:}\alpha.\phi_\tau$$

where in the first case $x \in \mathcal{V}ar_\tau$ and in the second $\mathsf{a} \in Name_\alpha$ where τ is a name sort α, and in the case of $\sigma(\phi_{\tau_1}, \ldots, \phi_{\tau_n})$ the operation symbol σ must be in $\Sigma_{\tau_1,\ldots,\tau_n;\tau}$. Both $\exists$ and $\mathsf{И}$ are binders (i.e., we work modulo α-equivalence): variables that are not under the scope of a $\exists$ and names that are not under the scope of a $\mathsf{И}$ are said to be free. Substitution of variables by patterns avoids capture of free variables or free names. Finally, names a are regarded as functional (single-valued) terms.

Note that in $\mathsf{И}\mathsf{a}:\alpha.\phi_\tau$, α is required to be a name sort ($\mathsf{И}$ binds names rather than variables) and names are now a separate syntactic class (in the semantics, names behave like constants so that we always know that distinct free names occurring in a pattern are always different). In addition we assume that the set Σ of symbols includes the following families of symbols indexed by the relevant sorts (in the following we sometimes omit the subscripts when they are obvious from the context):

$(--)\cdot-$	:	$\alpha \times \alpha \times \tau \to \tau$	swapping (function)
$[-]-$	:	$\alpha \times \tau \to [\alpha]\tau$	abstraction (function)
$-@-$	:	$[\alpha]\tau \times \alpha \rightharpoonup \tau$	concretion (partial function)
$fresh_{\tau,\alpha}$	$\in$	$\Sigma_{\tau;\alpha}$	freshness (multivalued operation)
$-\#_{\alpha,\tau}-$	:	$\alpha \times \tau \to Pred$	freshness relation (predicate)
$-^\dagger$	$\in$	$\Sigma_{Pred;\tau}$	coercion operator, often implicit

3.2.2 Semantics. We now formalize the semantics of nominal matching logic. This semantics simultaneously generalizes the semantics of nominal logic [Pitts 2003] and of matching logic [Chen and Rosu 2019], and takes into account the use of name constants [Cheney 2006; Urban et al. 2004]. That is, an instance of nominal logic can be considered an instance of nominal matching logic in which all symbols correspond to function symbols or predicates, while an instance of matching logic can be considered an instance of nominal matching logic in which there are no name-sorts (and thus no abstraction sorts and no associated nominal pattern constructors).

Definition 3.6 (NML Model). Given an NML signature Σ with components $(S, \mathcal{V}ar, Name, \Sigma)$, let $\mathbb{A}$ be $\bigcup_{\alpha \in NS} \mathbb{A}_\alpha$ where each $\mathbb{A}_\alpha$ is an infinite countable set of atoms and the $\mathbb{A}_\alpha$ are pairwise disjoint, and let G be a product of permutation groups $\prod_i Sym(\mathbb{A}_i)$ (i.e., G is the group of all sort-respecting permutations). A nominal matching logic model $M = (\{M_\tau\}_{\tau \in S}, \{\sigma_M\}_{\sigma \in \Sigma})$ consists of

(1) a non-empty nominal G-set M_τ for each $\tau \in S$;
(2) an equivariant interpretation

$$\sigma_M : M_{\tau_1} \times \cdots \times M_{\tau_n} \to \mathcal{P}_{fin}(M_\tau)$$

for each $\sigma \in \Sigma_{\tau_1,\ldots,\tau_n;\tau}$.

We say a model is *standard* if:

(1) the interpretation of each name sort α is the countably infinite set $\mathbb{A}_\alpha$;
(2) the interpretation of the sort $Pred$ is a singleton set $\{\star\}$, where $\star$ is equivariant, hence $\{\star\}$ is a nominal set whose powerset is isomorphic to Bool (as in standard models of matching logic, $\top$ is the full set and represents true and $\bot$ is the emptyset and represents false);

(3) the interpretation of each abstraction sort $[\alpha]\tau$ is $[M_\alpha]M_\tau$
(4) the interpretation of the swapping symbol $(--)\cdot- : \alpha \times \alpha \times \tau \to \tau$ is the swapping function on elements of M_τ;
(5) the interpretation of the abstraction symbol is the quotienting function mapping $\langle \mathsf{a}, x \rangle$ to its alpha-equivalence class, i.e. $\langle \mathsf{a}, x \rangle \mapsto \langle \mathsf{a}, x \rangle /_{\equiv_\alpha}$ where $\equiv_\alpha$ is as defined in Section 2.1;
(6) the interpretation of the concretion symbol is the (partial) concretion function $(X, \mathsf{a}) \mapsto \{y \mid (\mathsf{a}, y) \in X\}$, more precisely, concretion applies to an abstraction $[\mathsf{a}]v$ and a name b: if the name is fresh for the abstraction, it returns $(\mathsf{a}\,\mathsf{b}) \cdot v$, otherwise it is undefined;
(7) the interpretation of the freshness operation $fresh_{\tau,\alpha}$ is the function that returns all the names in $\mathbb{A}_\alpha$ that are fresh for the argument, i.e., the function $x \mapsto \{\mathsf{a} \mid \mathsf{a} \notin supp(x)\}$;
(8) the interpretation of the freshness relation $\#_{\alpha,s}$ is the freshness predicate on $\mathbb{A}_\alpha \times M_\tau$, i.e., it holds for the tuples $\{(\mathsf{a}, x) \mid \mathsf{a} \notin supp(x)\}$.

As usual in matching logic, the pointwise extension

$$\overline{\sigma_M} : \mathcal{P}_{fin}(M_{\tau_1}) \times \cdots \times \mathcal{P}_{fin}(M_{\tau_n}) \to \mathcal{P}_{fin}(M_\tau)$$

is defined as follows:

$$\overline{\sigma_M}(X_1, \ldots, X_n) \quad = \quad \bigcup\{\sigma_M(x_1, \ldots, x_n) \mid x_1 \in X_1, \ldots, x_n \in X_n\}$$

In particular, $\psi_\alpha \# \phi_\tau$ is interpreted as $\top_{Pred}$ (that is, $M_{Pred} = \{\star\}$) if an instance of ψ (a pattern of name sort) is fresh for an instance of ϕ (for example, $\mathsf{a} \# \mathsf{a} \vee \mathsf{b}$ is interpreted as $\top$). Similarly, $fresh(\phi_\tau)$ matches all the names that are fresh for some instance of ϕ_τ (for example $fresh(\mathsf{a} \vee \mathsf{b}) = \mathbb{A}$). On the other hand, note that $\mathsf{a} = \mathsf{a} \vee \mathsf{b}$ does *not* hold, because equality of patterns tests whether the two patterns have the same denotation. Equality can therefore not be defined as a symbol, since the meaning of symbols is always determined by their behavior on individual values. It would be possible to have an "equality symbol" $\sigma_=$ which, when restricted to single-element inputs, tests equality; however, for general patterns this operation would test whether the two argument patterns overlap, not whether they are equal.

Similarly, $[\phi_\alpha]\psi_\tau$ is a pattern of sort $[\alpha]\tau$ interpreted by the pointwise extension of the quotienting function associated with the abstraction symbol. For example, $[\mathsf{a} \vee \mathsf{b}](\mathsf{a} \vee \mathsf{b})$ matches $[\mathsf{a}]\mathsf{a}$, $[\mathsf{a}]\mathsf{b}$, and $[\mathsf{b}]\mathsf{a}$ (there are only three different values since $[\mathsf{a}]\mathsf{a} = [\mathsf{b}]\mathsf{b}$).

Definition 3.7 (Valuation). A function $\rho : \mathcal{V}ar \cup Name \rightharpoonup M$ with finite domain that is compatible with sorting (i.e., $\rho(x:\tau) \in M_\tau$, $\rho(\mathsf{a}:\alpha) \in M_\alpha$), injective on names and finitely supported is called a *valuation*. Injectivity ensures that two different names in the syntax are interpreted by different elements in the valuation. We say ρ is a ϕ-valuation when $dom(\rho) \supseteq FV(\phi) \cup FN(\phi)$, that is, all variables and names free in ϕ are assigned values by ρ.

Below, when we define the semantics of a pattern ϕ we implicitly assume that the valuation is a ϕ-valuation.

Valuations, and more generally (finite-domain, partial) functions from X to Y where X and Y are nominal sets, can be seen as elements of the G-set $(Y_\perp)^X$ of all finitely-supported partial functions from X to Y (see [Pitts 2011] for details). Therefore if π is a well-sorted permutation in $\mathbb{A}$, $\pi \cdot \rho$ is well defined: it is the valuation that maps $\mathsf{a} \in Name$ to $\pi \cdot \rho(\mathsf{a})$ and $x \in \mathcal{V}ar$ to $\pi \cdot \rho(x)$ (names

$$\|x : \tau\|_\rho = \{\rho(x)\}$$

$$\|\mathsf{a} : \alpha\|_\rho = \{\rho(\mathsf{a})\}$$

$$\|\sigma(\phi_1,\ldots,\phi_n)\|_\rho = \overline{\sigma_M}(\|\phi_1\|_\rho,\ldots,\|\phi_n\|_\rho)$$

$$\|\phi_1 \wedge \phi_2\|_\rho = \|\phi_1\|_\rho \cap \|\phi_2\|_\rho$$

$$\|\neg\phi\|_\rho = M_\tau - \|\phi\|_\rho$$

$$\|\exists x : \tau.\phi\|_\rho = \bigcup_{a \in M_\tau} \|\phi\|_{\rho[a/x]}$$

$$\|\mathsf{Иa} : \alpha.\phi\|_\rho = \bigcup_{a \in \mathbb{A}_\alpha - supp(\rho)} \{v \mid v \in \|\phi\|_{\rho[a/\mathsf{a}]} \wedge a \notin supp(v)\}$$

Figure 2: Semantics of NML

in *Name* and variables in $\mathcal{V}ar$ have empty support and are not affected by permutations of atoms in $\mathbb{A}$). Note that permutations preserve the injectivity, sort-respecting, and finite support properties so the result of applying π to ρ is also a valuation.

Recall that a function F is equivariant if $F(\pi \cdot e) = \pi \cdot F(e)$ for every e in the domain of F and every π. In particular, a valuation is equivariant if $(\pi \cdot \rho)(e) = \pi \cdot \rho(e)$ for every element in its domain.

A valuation need not be equivariant but must be finitely supported, i.e. $\pi \cdot \rho = \rho$ whenever $supp(\pi) \cap supp(\rho) = \emptyset$. The support of a valuation is the union of the supports of the image of elements in its domain.

Definition 3.8 (NML Pattern Semantics). The meaning (set of matching elements) of a pattern ϕ for a given valuation ρ is defined as shown in Figure 2.

A pattern $\mathsf{Иa} : \alpha.\phi_\tau$ matches those elements that match ϕ_τ where a is instantiated with an atom $a \in \mathbb{A}_\alpha$ fresh for ρ, and which do not have a in their support. It could be written equivalently as: $\|\mathsf{Иa} : \alpha.\phi\|_\rho = \{v \in \|\phi\|_{\rho[a/\mathsf{a}]} \mid a \in (\mathbb{A}_\alpha - supp(\rho)) - supp(v)\}$.

Note that in the interpretation of the $\mathsf{И}$ pattern, the valuation ρ is extended by assigning to a any element a of $\mathbb{A}_\alpha$ that is fresh (not in the support of the interpretations of free names and free variables or instances of ϕ). Compared to nominal logic, this differs because $\mathsf{И}$ is defined as a predicate symbol by an axiom scheme, whereas here the $\mathsf{И}$ quantifier can appear in an arbitrary place in a pattern. So for example $\exists x : \tau.\mathsf{Иa} : \alpha.\langle[\mathsf{a}]x, x\rangle$ is a pattern that characterizes pairs of abstractions and elements of M_τ where the abstracted name is fresh for the element. Such a pattern has no direct equivalent in nominal logic. This usage of $\mathsf{И}$ amounts to a form of local fresh name generation, sometimes denoted ν in other settings (e.g. Pitts' $\lambda\alpha\nu$-calculus 2011) to distinguish it from the fresh name quantifier occurring as a formula. In NML there is no distinction between formulas and terms, so we use $\mathsf{И}$ in both places; also, $\mathsf{И}$ behaves differently than ν in $\lambda\alpha\nu$, for example $\nu a.a$ denotes an anonymous name rather than the empty set.

Before studying properties of NML we provide some simple examples.

Example 3.9. • Suppose ϕ does not contain a as a free name; then $\mathsf{Иa}.\phi$ is equivalent to ϕ.

This result follows directly from the semantics of $\mathsf{И}$: if a is not free in ϕ then $\| - \|_{\rho[a/\mathsf{a}]}$ and $\| - \|_\rho$ produce the same result.

• $\phi_1 = \mathsf{Иa}.\mathsf{a}$ is a pattern that matches nothing (its interpretation is the empty set). Likewise $\phi_2 = \mathsf{Иa}.\langle\mathsf{a}, \mathsf{a}\rangle$ and $\phi_3 = \mathsf{Иa}.\mathsf{Иb}.\langle\mathsf{a}, \mathsf{b}\rangle$ are also empty.

• $\phi_4 = \mathsf{Иa}.[\mathsf{a}]\mathsf{a}$ matches any abstraction whose body is the abstracted name. Note that $\mathsf{Иa}.[\mathsf{a}]\mathsf{a}$ is a closed pattern, whereas $[\mathsf{a}]\mathsf{a}$ has a free name a and can only be interpreted in a valuation that assigns a value to a (and then its denotation is the same as that of $\mathsf{Иa}.[\mathsf{a}]\mathsf{a}$).

• $\phi_5 = \mathsf{Иa}.\mathsf{a} = \mathsf{a}$ is a valid predicate (equivalent to $\top$)

• $\phi_6 = \exists x.\mathsf{Иa}.\mathsf{a} = x$ is false/empty since whatever x is, a must be chosen fresh for (and in particular distinct from) it. However, $\mathsf{Иa}.\exists x.\mathsf{a} = x$ is true since we may choose $x = \mathsf{a}$.

• $\phi_7 = \mathsf{Иa}.[\mathsf{a}]\phi$ is a pattern that matches any abstraction where the abstracted atom is not in the support of (the interpretations of) free variables and free names of ϕ (aside from a itself) and the body is an instance of ϕ that does not contain the abstracted atom in its support.

To illustrate this kind of pattern, consider three possible rules representing eta-equivalence for the lambda-calculus:

$$x : Exp = lam([\mathsf{a}]app(x, var(\mathsf{a})))$$
$$x : Exp = lam(\exists \mathsf{a}.[\mathsf{a}]app(x, var(\mathsf{a})))$$
$$x : Exp = lam(\mathsf{Иa}.[\mathsf{a}]app(x, var(\mathsf{a})))$$

In the first rule, only the specific name a can be used, and if it is also present in the support of x then the result will be wrong (e.g. if $x = var(\mathsf{a})$ then a is captured.) The second rule allows eta-expanding using any name a, but still permits inadvertent capture of names in x. The third rule correctly permits eta-expanding using any sufficiently fresh name a while ruling out variable capture. The right-hand side of the third rule is also equivalent to $\exists \mathsf{a}.\mathsf{a} \# x \wedge lam([\mathsf{a}]app(x, var(\mathsf{a})))$ and $\forall \mathsf{a}.\mathsf{a} \# x \Rightarrow lam([\mathsf{a}]app(x, var(\mathsf{a})))$ (Prop. 3.16).

3.2.3 Properties. In this section we show that the semantics is well defined in standard models, that is, the semantic interpretation $\rho \mapsto \|\phi\|_\rho$ is an equivariant map from ϕ-valuations to M_τ for every ϕ_τ (Theorem 3.12 and Cor. 3.13) and the expected equivalences of patterns are satisfied (Prop. 3.11, 3.14 - 3.17).

Definition 3.10 (Semantic equivalence of patterns in standard models). We say ϕ and ψ are equivalent patterns, written $\phi \iff \psi$, if for all suitable ρ (whose domain includes the free variables and atoms of ϕ and ψ), $\|\phi\|_\rho = \|\psi\|_\rho$. Alternatively, equivalence of patterns can be defined using equality: ϕ and ψ are equivalent if for all suitable ρ we have $\|\phi = \psi\|_\rho = \top$.

Recall that in standard models of NML, abstractions are interpreted as equivalence classes, therefore the alpha-equivalence relation generated by abstractions is already built into the semantics. Moreover, the following properties of abstraction patterns follow directly from the definition of NML pattern semantics:

PROPOSITION 3.11. (1) $[a](\phi_1 \wedge \phi_2) \iff [a]\phi_1 \wedge [a]\phi_2$
(2) $[a](\neg\phi) \iff [a]\top \wedge \neg[a]\phi$
(3) $([a]\phi)@b \iff (b \# [a]\phi) \wedge (a\ b) \cdot \phi$

THEOREM 3.12 (EQUIVARIANT SEMANTICS). *If* $v \in \|\phi\|_\rho$ *then* $(a\,a') \cdot v \in \|\phi\|_{(a\,a')\cdot\rho}$. *I.e., for all* ϕ, $(a\,a') \cdot \|\phi\|_\rho = \|\phi\|_{(a\,a')\cdot\rho}$.

COROLLARY 3.13. *If* $v \in \|\phi_\tau\|_\rho$, $\rho(a) = a$ *and* $\rho(a') = a'$ *then* $(a\,a')\cdot v \in \|(a\,a')\cdot\phi_\tau\|_\rho$ *(because the swapping symbol is interpreted by the swapping function in the nominal set M_τ).*

In particular, for any pattern ϕ_τ, if a, a' are fresh for v then $v \in \|\phi_\tau\|_\rho$ if and only if $v \in \|(a\,a') \cdot \phi_\tau\|_\rho$.

Since Иa.ϕ patterns are matched by instances of ϕ obtained by assigning to a a fresh atom, it is easy to see that if a is not free in ϕ then Иa.ϕ and ϕ have the same instances. Also, the order in which fresh atoms are quantified is not important (Proposition 3.14).

PROPOSITION 3.14. (1) *If* a *is not free in* ϕ *then* Иa.$\phi \iff \phi$.
(2) Иa: $\alpha.$Иb: $\alpha'.\phi_\tau \iff$ Иb: $\alpha'.$Иa: $\alpha.\phi_\tau$.

PROPOSITION 3.15. *The* И *pattern satisfies the following properties:*

(1) Иa: $\alpha.\phi_\tau \wedge$ Иa: $\alpha.\psi_\tau \iff$ Иa: $\alpha.(\phi_\tau \wedge \psi_\tau)$. *In particular,* Иa.$(\phi \wedge \psi) \iff ((\text{Иa}.\phi) \wedge \psi)$ *if* a *is not free in* ψ. *A similar property holds for* $\vee$ *patterns:* Иa: $\alpha.\phi_\tau \vee$ Иa: $\alpha.\psi_\tau \iff$ Иa: $\alpha.(\phi_\tau \vee \psi_\tau)$.
(2) $\neg$Иa: $\alpha.\phi \iff$ Иa: $\alpha.\neg\phi$
(3) $[a]$Иb: $\alpha.\phi_\tau \iff$ Иb: $\alpha.[a]\phi_\tau$.
(4) $\langle$Иa.$\phi, \psi\rangle \iff$ Иa.$\langle\phi, \psi\rangle$ *if* a *does not occur free in* ψ. *More generally, if* $\forall x_1, \ldots, x_n, a\#\sigma(x_1, ..., x_n) \Rightarrow a\#x_1 \wedge \ldots \wedge a\#x_n$ *then* $\sigma(\text{Иa}.\phi_1, \ldots, \text{Иa}.\phi_n) \iff$ Иa.$\sigma(\phi_1, \ldots, \phi_n)$.

The И pattern does not commute with abstraction for the same name: Иa.$[a]$a $\neq [a]$Иa.a (the first one matches abstractions where the body is the abstracted name, whereas the second does not match anything). It does commute if the names are different: $[a]$Иb.ϕ and Иb.$[a]\phi$ are equivalent patterns that match an abstraction where the body refers to some fresh name b different from the abstracted name. Note that when И occurs inside an abstraction the И-bound name can always be renamed away from the abstracted term, this means that we can always hoist И out of the body of an abstraction. However, $[\text{Иa}.\phi_\alpha]\psi_\tau \neq$ Иa.$[\phi_\alpha]\psi_\tau$ even if a is not free in ψ. For example, $[\text{Иa}.a]$b is matched by nothing, whereas Иa.$[a]$b is matched by $[a]$b for two different a, b. Note that the abstraction symbol does not satisfy the condition in item 4 of Proposition 3.15.

The И pattern can be defined using existential (or universal) patterns and the freshness relation symbol as shown in Proposition 3.16. Since the existential and universal patterns bind variables and И binds names, to define the Иa.ϕ using $\exists$ or $\forall$ we need to replace the bound name a by a bound variable z_a which we assume does not occur anywhere else. Finally, notice that in both existential and universal equivalent formulas, we introduce an existential variable y standing for a matched result of the pattern, and the fresh name z_a must be fresh for this as well, corresponding to the constraint $a \notin supp(v)$ in the semantics of И. This is critical for the self-duality of И and to ensure that the universal and existential characterizations are equivalent.

PROPOSITION 3.16. *The* И *pattern satisfies the following equivalences, where the notation $\phi_\tau(a, \bar{b}, \bar{x})$ indicates that $\bar{x}$ are the free variables of ϕ and $\bar{b}$ (and possibly also a) are the free names of ϕ, z_a is a variable associated with the name a, which we assume is not used*

anywhere else, and $\phi_\tau\{a \mapsto z_a\}$ is the result of replacing the name a with z_a in ϕ_τ.

(1) Иa: $\alpha.\phi_\tau(a, \bar{b}, \bar{x}) \Leftrightarrow$
$\exists z_a: \alpha.((\exists y: \tau.y \wedge z_a\#_s^\dagger(\bar{b}, \bar{x}, y)) \wedge \phi_\tau\{a \mapsto z_a\})$
(2) Иa: $\alpha.\phi_\tau(a, \bar{b}, \bar{x}) \Leftrightarrow$
$\forall z_a: \alpha.((\exists y: \tau.y \wedge z_a\#_s^\dagger(\bar{b}, \bar{x}, y)) \Rightarrow \phi_\tau\{a \mapsto z_a\})$

The following property links abstraction, concretion and И, in a similar way as axiom $(A1')$ in NLML.

PROPOSITION 3.17. (Иa.$[a]\phi$)@b = Иa.$(a\,b) \cdot \phi$.

We end this section with some observations relating NL, ML, NLML and NML. The translation of the axioms of nominal logic are valid in the standard models of NML: $(S1)$–$(S3)$ are satisfied since we interpret the swapping symbol with the swapping operation in the model, similarly $(E1)$–$(E5)$ are satisfied since the interpretation of symbols σ in the signature is equivariant, $(F1)$–$(F4)$ follow from the interpretation of the freshness operator and the interpretation of sorts as nominal sets (which ensures $(F4)$ holds). Axiom (Q) follows from Prop. 3.16. $(A1)$ and $(A2)$ are consequence of the interpretation of abstraction sorts as abstractions and the interpretation of the abstraction symbol.

Conversely, NML can be translated to NLML, by introducing (suitably freshness-constrained) variables to represent ground names and using Prop. 3.16 to eliminate И-quantifiers. Alternatively, NML can be translated to ordinary nominal logic with ground names (e.g. Cheney's system NLSeq [2016]) following the same approach taken in translating ML to FOL. Thus the proof systems already available for ML or NL can be leveraged to reason about NML. A more appealing approach would be to develop a proof system for NML directly, perhaps buiding on NLSeq. We leave this issue for future investigation.

4 EXAMPLES

We now fix a specific model for reasoning about the typed lambda-calculus, with sorts including *Exp* (expressions), *Ty* (types), and *Var* (variables, a name-sort). These sorts are interpreted as nominal sets M_{Var}, M_{Exp}, and M_{Ty} satisfying the following equations:

$$M_{Exp} = M_{Var} + (M_{Exp} \times M_{Exp}) + [M_{Var}]M_{Exp}$$
$$M_{Ty} = 1 + M_{Ty} \times M_{Ty} + \cdots$$

The three cases for expressions correspond to *var*, *app* and *lam* as discussed in Section 2. We assume at least one constant type (e.g. *int* or *unit*) and assume there is a binary constructor $fn : Ty \times Ty \to Ty$ for function types but leave the exact set of types unspecified otherwise. It is well-established that the abstraction construction satisfies the required properties (monotonicity, continuity) so that least fixed points of equations involving abstractions exist according to nominal versions of standard fixed-point theorems [Pitts 2006], and the resulting models are finitely-supported. Therefore the initial algebra M_{Exp} is essentially the set of lambda-terms quotiented by alpha-equivalence. We fix M_Λ as the standard model obtained taking M_{Exp} and M_{Ty} as defined above, and in the rest of this section statements are relative to this model.

As indicated at the end of the previous section, neither NLML nor NML have been implemented as part of an automated system.

The examples and proofs in this section are carried out by hand and have not been mechanically checked.

4.1 The Lambda Calculus in NLML

Recall the signature for lambda-terms and axioms for substitution from Section 2, see Example 2.1. Consider the induction principle, which is schematic over a predicate $P \in \Sigma_{Exp,\tau_1,\ldots,\tau_n;Pred}$:

$$
\begin{aligned}
(\forall a\colon Var.P(var(a),\bar{y})) &\implies \\
(\forall t_1\colon Exp, t_2\colon Exp.P(t_1,\bar{y}) \wedge P(t_2,\bar{y}) \implies P(app(t_1,t_2),\bar{y})) &\implies \\
(\forall a,t.a \,\#\, \bar{y} \implies P(t,\bar{y}) \implies P(lam([a]t)),\bar{y}) &\implies \\
\forall t\colon Exp.P(t,\bar{y})
\end{aligned}
$$

This induction principle deserves some explanation. We make any other parameters of the induction hypothesis (e.g. variables with respect to which any bound names must be chosen fresh) explicit via a parameter list $\bar{y}$. The cases for variables and application are otherwise standard. In the case for lambda, the induction step we need to show is that if $P(t,\bar{y})$ holds for some *sufficiently fresh* name a (i.e. for $a \,\#\, \bar{y}$) then $P(lam([a]t),\bar{y})$ holds. For more details, see Pitts [2006] and Urban [2008].

Now suppose we wish to prove the following (standard) property of substitution:

$$
\begin{aligned}
P(x,y,z,y',z') \quad\triangleq\quad & y \,\#\, y',z' \implies subst(subst(x,y,z),y',z') \\
& = subst(subst(x,y',z'),y,subst(z,y',z')).
\end{aligned}
$$

We proceed by induction on x. The base case where x is a variable is straightforward by case analysis on y. The case where x is an application is also straightforward.

For the case of lambda-abstraction, let a,t be given such that $a \,\#\, y,z,y',z'$. We need to show:

$$ subst(subst(x,y,z),y',z') = subst(subst(x,y',z'),y,subst(z,y',z')) $$

implies

$$
\begin{aligned}
& subst(subst(lam([a]x),y,z),y',z') \\
=\ & subst(subst(lam([a]x),y',z'),y,subst(z,y',z')).
\end{aligned}
$$

Accordingly, reason as follows:

$$
\begin{aligned}
& subst(subst(lam([a]x),y,z),y',z') \\
=\ & subst(lam([a](subst(x,y,z))),y',z') \\
=\ & lam([a]subst(subst(x,y,z),y',z'))
\end{aligned}
$$

using the substitution axiom ($Subst4$) (see Example 2.1) twice since we know $a \,\#\, y,z,y',z'$.

We can now apply the induction hypothesis, relying again on the freshness assumptions for a:

$$
\begin{aligned}
& lam([a]subst(subst(x,y,z),y',z')) \\
=\ & lam([a]subst(subst(x,y',z'),y,subst(z,y',z')))
\end{aligned}
$$

Finally, we apply again the substitution axioms twice:

$$
\begin{aligned}
& lam([a]subst(subst(x,y',z'),y,subst(z,y',z'))) \\
=\ & subst(lam([a]subst(subst(x,y',z')),y,subst(z,y',z')) \\
=\ & subst(subst(lam([a]x,y',z'),y,subst(z,y',z'))
\end{aligned}
$$

The final two steps require the freshness conditions: $a \,\#\, y',z',y,subst(z,y',z')$. By assumption, $a \,\#\, y',z',y$ holds. The final freshness assertion requires an additional lemma:

$$ a \,\#\, x,y,z \implies a \,\#\, subst(x,y,z) $$

which holds by equivariance of symbols.

Nominal logic supports proofs by induction for syntax with binding operators. NLML inherits the inductive reasoning techniques available in nominal logic, as illustrated in this example. Since NLML is an instance of matching logic with a specific theory, it can be used in any implementation of matching logic. However, as discussed in Section 3.1.6, there is an alternative, more concise way to specify the syntax of the lambda calculus and the induction principle using Ⅴ, which can be written in NML as shown below.

4.2 The Lambda Calculus in NML

Using NML we can axiomatize the substitution operation in the λ-calculus as follows:

$$
\begin{aligned}
subst(var(a),a,z) &= z \\
subst(var(a),\neg a,z) &= var(a) \\
subst(app(x_1,x_2),y,z) &= app(subst(x_1,y,z),subst(x_2,y,z)) \\
subst(lam(x),y,z) &= lam(Ⅴa.[a]subst(x@a,y,z))
\end{aligned}
$$

Notice that this is now a completely equational definition, no side conditions are required (cf. the axiomatization of substitution given in Example 2.1). The last axiom can be replaced with the equivalent:

$$ subst(lam(x),y,z) = Ⅴa.lam([a]subst(x@a,y,z)) $$

(see Proposition 3.15, part 4, which applies since *lam* acts as a constructor).

The induction principle for expressions can be formulated using Ⅴ and concretion to deal with the case of lambda-abstractions, avoiding the need for freshness constraints (compare with the version of the induction principle in the previous section):

$$
\begin{aligned}
(\forall x\colon Var.P(var(x))) &\implies \\
(\forall t_1\colon Exp, t_2\colon Exp.P(t_1) \wedge P(t_2) \implies P(app(t_1,t_2))) &\implies \\
(\forall t\colon [Var]Exp.Ⅴa\colon Var.P(t@a) \implies P(lam(t)) &\implies \\
\forall t\colon Exp.P(t)
\end{aligned}
$$

Note that because we use the Ⅴ-quantifier in the case for lambda, the freshness side-conditions $a \,\#\, \bar{y}$ are implicit, and so we can omit the additional parameters $\bar{y}$, which makes this induction principle simpler and cleaner. In the lambda case, we need to prove that if $P(t@a)$ holds for a fresh name a then $P(lam(t))$ holds; it is somewhat easier to see how to generalize this principle systematically to any nominal datatype.

Also note that thanks to the distinction between names and variables, the Substitution Lemma can now be stated as shown below with just one freshness condition, which formalizes the usual side-condition in textbook statements [Barendregt 1984] (we do not need to specify a $\neq$ b explicitly since this always holds for two different atoms).

$$
\begin{aligned}
& a \,\#\, z' \implies \\
& subst(subst(x,a,z),b,z') = subst(subst(x,b,z'),a,subst(z,b,z'))
\end{aligned}
$$

This means that the induction step in the lambda case becomes:

$$
\begin{aligned}
(Ⅴc.a \,\#\, z' \implies subst(subst(x@c,a,z),b,z') \\
= subst(subst(x@c,b,z'),a,subst(z,b,z')) \implies \\
(a \,\#\, z' \implies subst(subst(lam(x),a,z),b,z') \\
= subst(subst(lam(x),b,z'),a,subst(z,b,z')))
\end{aligned}
$$

The reasoning for the lambda case is as follows using the fourth axiom for substitution twice:

$$subst(subst(lam(x), a, z), b, z')$$
$$= \quad subst(lam(\sqcap c.[c]subst(x@c, a, z)), b, z')$$
$$= \quad lam(\sqcap d.[d]subst((\sqcap c.[c]subst(x@c, a, z))@d, b, z')))$$

By Prop. 3.17, $(\sqcap c.[c]subst(x@c, a, z))@d = \sqcap c.(c\ d)\cdot subst(x@c, a, z)$. The latter is equivalent to $\sqcap c.subst(x@d, a, z)$, since c, d are new (not in the support of the variables in the pattern). By Prop. 3.14(1), the latter is equivalent to $subst(x@d, a, z)$. Therefore,

$$subst(subst(lam(x), a, z), b, z')$$
$$= \quad lam(\sqcap d.[d]subst(subst(x@d, a, z), b, z')))$$

By induction hypothesis, the above equals

$$lam(\sqcap d.[d]subst(subst(x@d, b, z'), a, subst(z, b, z')))\ .$$

Applying again Prop. 3.17 and Prop. 3.14(1), the latter equals to

$$lam(\sqcap d.[d]subst((\sqcap c.[c]subst(x@c, b, z'))@d, a, subst(z, b, z'))).$$

We can now use ($Subst4$) to complete the proof:

$$subst(subst(lam(x), a, z), b, z')$$
$$= \quad subst(lam(\sqcap c.[c]subst(x@c, b, z')), a, subst(z, b, z'))$$
$$= \quad subst(subst(lam(x, b, z')), a, subst(z, b, z'))$$

4.3 Reduction and Well-Formedness

Using the same signature for λ-terms as in the previous section, reduction $red \in \Sigma_{Exp;Exp}$ can be defined in NML as follows:

$$red(app(x, y)) \quad = \quad app(red(x), y) \vee app(x, red(y))$$
$$\qquad\qquad\qquad \vee \quad (\exists z.x = lam(z) \wedge \sqcap a.subst(z@a, a, y))$$
$$red(lam([a]y)) \quad = \quad lam([a]red(y))$$

These can be viewed as first-order axioms or as an inductive definition using a least fixed point pattern μ, following the strategy for defining operations inductively presented by Chen and Rosu for matching μ logic [Chen and Rosu 2019]. The first axiom defines reduction for applications: it states that the instances of $red(app(x, y))$ are obtained by reducing in the first argument, or in the second argument, or applying the β-rule at the root if the first argument is a λ-abstraction, using the capture-avoiding substitution symbol $subst$ previously defined. These are the only instances of red.

Type checking can also be characterized in NML as shown below, using sorts Ty for types and $Map[Var, Ty]$ for finite maps from variables to types (we abbreviate $c: Map[Var, Ty]$ as $c: Ctx$). The symbol $wf \in \Sigma_{Ctx, Ty; Exp}$, denoting well-formed expressions of type Ty in the typing context Ctx, satisfies the following axiom:

$$wf(c, t) \quad = \quad \exists a: Var.var(a) \wedge t = c[a]$$
$$\qquad\qquad \vee \quad \exists u: Ty.app(wf(c, fn(u, t)), wf(c, u))$$
$$\qquad\qquad \vee \quad \exists t_1, t_2: Ty.t = fn(t_1, t_2) \wedge$$
$$\qquad\qquad\qquad\qquad\qquad lam(\sqcap a.[a]wf(c[a := t_1], t_2))$$

Here we write $c[a]$ for the (partial) operation that looks up a's binding in c and write $c[a := t]$ for the (partial) operation that extends a finite map c with a binding for a variable not already present in its domain. Notice that $wf(c, t)$ is a pattern which matches just those lambda-terms that are well-formed in context c and have type t: The first line in the definition of $wf(c, t)$ indicates that a variable $var(a)$ is a well-formed term of type t in c if $c[a] =$

t; the second line specifies the usual typing rule for applications: the first argument must have arrow type $fn(u, t)$ in the context c, while the second argument must have type u in c; the third and fourth lines specify the typing rule for λ expressions: a well-formed λ-abstraction of type $fn(t_1, t_2)$ in c is an instance of the pattern $lam(\sqcap a.[a]wf(c[a := t_1], t_2))$. For example, $lam([a]var(a)) \in wf([], fn(t, t))$ holds for any type t.

Finite maps satisfy standard axioms, such as

$$c[a := t][a] = t \qquad a \neq b \Rightarrow c[a := t][b] = c[b]$$
$$x \in c[a] \wedge y \in c[a] \Rightarrow x = y$$

and we abuse notation slightly by writing $c \subseteq c'$ for $\forall a.c[a] \subseteq c'[a]$.

Subject Reduction. The standard property of subject reduction can be stated as follows:

$$red(wf(c, t)) \subseteq wf(c, t)$$

that is, all the reducts of a well-formed term of type t in context c (i.e., the instances of $red(wf(c, t))$) are well-formed terms of type t in context c. In other words, given a well-formed term of type t in context c, reducing it yields another well-formed term of the same type. This property can be proved using the axioms for red and wf and the following induction principle:

$$(\forall t: Ty, a: Var, c: Ctx.(var(a) \wedge t = c[a]) \subseteq P(c, t)) \quad \Rightarrow$$
$$(\forall t_1, t_2: Ty, c: Ctx.app(P(c, fn(t_1, t_2)), P(c, t_1)) \subseteq P(c, t_2)) \quad \Rightarrow$$
$$(\forall t_1, t_2: Ty, c: Ctx.lam(\sqcap a: Var.[a]P(c[a := t_1], t_2))$$
$$\subseteq P(c, fn(t_1, t_2))) \quad \Rightarrow$$
$$\forall c: Ctx, t: Ty.wf(c, t) \subseteq P(c, t)$$

together with a lemma stating that well-typed substitutions preserve types:

$$\forall a, t, t', c, c'. \quad c' \subseteq c$$
$$\Rightarrow subst(wf(c[a := t'], t), a, wf(c', t')) \subseteq wf(c, t)$$

First we show how to prove this lemma using the above-mentioned induction principle and the axioms for $subst$ and wf.

We start by using the definition of wf and the semantics of symbols, which extends pointwise to sets of arguments, allowing us to move the $subst$ symbol inside $\exists$ and $\vee$ patterns:

$$subst(wf(c[a := t'], t), a, wf(c', t))$$
$$= \quad subst(\exists a': Var.var(a') \wedge t = (c[a := t'])[a']$$
$$\vee \exists u: Ty.app(wf(c[a := t'], fn(u, t)), wf(c[a := t'], u))$$
$$\vee \exists t_1, t_2: Ty.t = fn(t_1, t_2) \wedge$$
$$lam(\sqcap b.[b]wf(c[a := t'][b := t_1], t_2)), a, wf(c', t'))$$
$$= \quad \exists a': Var.subst(var(a') \wedge t = (c[a := t'])[a'], a, wf(c', t'))$$
$$\vee \exists u: Ty.subst(app(wf(c[a := t'], fn(u, t)), wf(c, u)),$$
$$\qquad\qquad\qquad\qquad\qquad\qquad a, wf(c', t'))$$
$$\vee \exists t_1, t_2: Ty.subst(t = fn(t_1, t_2) \wedge$$
$$lam(\sqcap b.[b]wf(c[a := t'][b := t_1], t_2)), a, wf(c't'))$$

Note that $(c[a := t'])[a] = t'$ and to prove the lemma it is sufficient to prove that each of the disjuncts is included in $wf(c, t)$.

For the first disjunct, we consider whether the variable a' is equal to a or not. If so, then $t = c[a := t'][a] = t'$ and by ($Subst1$) (see Example 2.1) $subst(var(a), a, wf(c', t')) \subseteq wf(c', t') \subseteq wf(c, t)$ as desired since $c' \subseteq c$ and $t = t'$. Otherwise, $(c[a := t'])[a'] = c[a']$ and $subst(var(a'), a, wf(c', t')) = var(a')$ and by definition of wf we have $\exists a'.var(a') \wedge t = c[a'] \subseteq wf(c, t)$.

In the second disjunct, using $(Subst3)$ to move $subst$ inside app, the induction hypothesis, and the definition of wf, we obtain the required inclusion.

Finally, in the third disjunct again we can move V out (Prop. 3.15(4)) and then apply axiom $(Subst4)$ to move $subst$ inside lam, completing the proof by using the induction hypothesis again when $t = fn(t_1, t_2)$ (otherwise the disjunct is empty and the inclusion holds trivially).

To prove Subject Reduction, again we start by using the definition of wf:

$$
\begin{aligned}
& red(wf(c,t)) \\
= \; & red(\exists a : Var.var(a) \wedge t = c[a] \\
& \vee \exists u : Ty.app(wf(c, fn(u,t)), wf(c,u)) \\
& \vee \exists t_1, t_2 : Ty.t = fn(t_1, t_2) \wedge lam(\mathsf{V}\mathsf{a}.[\mathsf{a}]wf(c[\mathsf{a} := t_1], t_2))) \\
= \; & \exists a : Var.red(var(a) \wedge t = c[a] \\
& \vee \exists u : Ty.red(app(wf(c, fn(u,t)), wf(c,u)) \\
& \vee \exists t_1, t_2 : Ty.red(t = fn(t_1, t_2) \wedge \\
& \qquad\qquad\qquad\qquad lam(\mathsf{V}\mathsf{a}.[\mathsf{a}]wf(c[\mathsf{a} := t_1], t_2)))
\end{aligned}
$$

Note that $red(var(a)) = \bot$ so the first disjunct is empty.

In the second disjunct we can use the first axiom for red. Notice that by Prop. 3.17, the following is equivalent to the original axiom involving concretion: $red(app(x,y)) = app(red(x),y) \vee app(x, red(y)) \vee (\mathsf{V}\mathsf{a}.\exists z.x = lam([\mathsf{a}]z) \wedge subst(z, \mathsf{a}, y))$.

In the third disjunct we can use the second axiom for red after moving the V out (Prop. 3.15(4)). Hence,

$$
\begin{aligned}
& red(wf(c,t)) \\
= \; & \exists u : Ty.app(red(wf(c, fn(u,t))), wf(c,u)) \\
& \qquad \vee app(wf(c, fn(u,t)), red(wf(c,u))) \\
& \qquad \vee (\mathsf{V}\mathsf{a}.\exists z.wf(c, fn(u,t)) \supseteq lam([\mathsf{a}]z) \\
& \qquad\qquad \wedge subst(z, \mathsf{a}, wf(c,u))) \\
& \vee \exists t_1, t_2 : Ty.t = fn(t_1, t_2) \\
& \qquad \wedge \mathsf{V}\mathsf{a}.lam([\mathsf{a}]red(wf(c[\mathsf{a} := t_1], t_2)))
\end{aligned}
$$

We can push the existential quantifiers above under disjunction to get four disjuncts. It remains to prove that each disjunct above is included in $wf(c,t)$. The first two and the last inclusions follow directly by induction. The third follows from the lemma above, noticing that

$$
wf(c, fn(u,t)) \supseteq lam([\mathsf{a}]z) \Rightarrow z \subseteq wf(c[\mathsf{a} := u], t)
$$

by definition of wf.

Progress. Another standard property in a syntactic proof of type soundness states that a well-formed closed term that is not weakly-reducible is a value, where in this case $value = lam(\top)$ since we consider only a pure lambda-calculus. This is equivalent to saying that a well-formed closed term is either a value or can be reduced, so can be written as follows:

$$
wf([], t) \subseteq value \vee reducible
$$

where $reducible$ is defined as $\exists x.x \wedge \exists y.y \in red(x)$, that is, $reducible$ matches those terms that can reduce in some way. We prove some auxiliary properties of $reducible$:

$$
app(reducible, \top) \subseteq reducible \qquad app(lam(\top), \top) \subseteq reducible
$$

To show $app(reducible, \top) \subseteq reducible$ suppose we have some matches $app(reducible, \top)$, that is, it is an application $app(x,y)$ whose first argument is reducible, i.e. there is some $x' \in red(x)$.

Hence $app(x', y)$ matches $red(app(x,y))$ so $app(x,y)$ is reducible. To show $app(lam(\top), \top) \subseteq reducible$, again suppose $app(lam(x), y)$ matches $app(lam(\top), \top)$. Then

$$
\exists z.lam(x) = lam(z) \wedge \mathsf{V}\mathsf{a}.subst(z@\mathsf{a}, \mathsf{a}, y) \subseteq red(app(lam(x), y))
$$

so we just need to show the contained pattern is inhabited. Clearly choosing $x = z$ satisfies the first conjunct and it remains to show that the substitution pattern is defined. This can be proved by induction on the first argument to $subst$.

We now outline a proof sketch for the progress property, which we reformulate slightly to $c = [] \Rightarrow wf(c,t) \subseteq value \vee reducible$. Assume $c = []$. Proceed by induction on $wf(c,t)$ to prove $wf(c,t) \Rightarrow value \vee reducible$.

- Case: $\exists a.var(a) \wedge t = c[a] \subseteq value \vee reducible$: $c = []$ so $c[a]$ is empty and so $\exists a.var(a) \wedge t = c[a] = \bot \subseteq value \vee reducible$.
- Case: $app(value \vee reducible, value \vee reducible) \subseteq value \vee reducible$. We consider two cases. If the first argument is reducible then, by definition of $reducible$:
$app(reducible, value \vee reducible) \subseteq app(reducible, \top)$
$\subseteq value \vee reducible$.
Otherwise, the first argument is a value and since $value = lam(\top)$ we have

$$
\begin{aligned}
app(lam(\top), value \vee reducible) \quad & \subseteq \quad app(lam(\top), \top) \\
& \subseteq \quad value \vee reducible
\end{aligned}
$$

again by definition. (Here the reasoning is simpler because in the pure simply typed lambda calculus there is only one form of value; for a larger language with some other value forms we would need a separate lemma ensuring that values of function type are lambdas.)
- Case: $lam(\mathsf{V}\mathsf{a}.[\mathsf{a}]value \vee reducible) \subseteq value \vee reducible$. This follows immediately since $lam(\mathsf{V}\mathsf{a}.[\mathsf{a}]value \vee reducible) \subseteq lam(\top) = value$.

Although in many cases occurrences of V are adjacent to abstraction (which could justify having syntactic sugar for this), the examples in this section, in particular the induction principles, illustrate the advantage of keeping V independent of abstraction.

5 COMPARISON WITH BINDING IN APPLICATIVE MATCHING LOGIC

Binders, as exemplified by λ in the λ-calculus, can be encoded in matching logic by using an existential pattern, a special symbol intension and a sort containing representatives of all possible function graphs, as shown by Chen and Rosu [2020]. The existential pattern represents a function as a set of pairs (argument, value) and intension packs this set as an object in a power sort. For example, to represent the λ-term $\lambda x.e$, we apply intension to $\exists x : Var.\langle x, e \rangle$, which produces an element of the power sort $2^{Var \times Exp}$, and introduce a symbol lam $\in \Sigma_{2^{Var \times Exp};Exp}$ that decodes it into the intended interpretation of $\lambda x.e$. Abbreviating intension $\exists x : Var.\langle x, e \rangle$ as $[x : Var]e$, we obtain a notation like that for abstraction in nominal logic, but note that whereas the abstraction $[x]e$ in nominal logic denotes an α-equivalence class of pairs, in matching logic it denotes the combination of an existential pattern (matching a

union of pairs) with an intension operator that packs the union into an object — essentially a second-order construction.

Given an expression $\mathsf{lam}([x]e)$ and a fresh variable y, in nominal logic the α-equivalence $\mathsf{lam}([x]e) = \mathsf{lam}([y](x\ y) \cdot e)$ holds by construction (given the axiomatisation of swapping and abstraction), whereas in matching logic only the α-equivalence between the existential patterns $\exists x\colon Var.\langle x, e\rangle$ and $\exists y\colon Var.\langle x, e\{x \mapsto y\}\rangle$ holds by construction. We can then prove the alpha-equivalence $[x\colon Var]e = [y\colon Var](e\{x \mapsto y\})$ by relying on the semantics of existential patterns, and the semantics of the intension symbol.

Once we have a mechanism to define syntax with binders, we can define the β-reduction relation. In both nominal and matching logic, we can introduce axioms to specify this relation. For example, Chen and Rosu [2020] give the following (β) axiom schema:

$$\forall x_1\colon Var. \cdots \forall x_n\colon Var.\ \mathsf{app}(\mathsf{lam}[x]e, e') = e[e'/x]$$

where e, e' are patterns encoding two λ-calculus expressions, $x_1, \ldots, x_n$ are the free variables of $(\lambda x.e)\ e'$, and $e[e'/x]$ denotes the meta-level substitution of matching logic. In other words, we have one axiom for each possible β-redex, for example:

$$\forall xyz.(\mathsf{lam}[w]\mathsf{app}(\mathsf{app}(w, w), \mathsf{app}(y, z)) = \mathsf{app}(\mathsf{app}(y, z), \mathsf{app}(y, z))$$

In contrast, in nominal logic (or NLML) one can write this as a single (conditional) equational law

$$\forall e, e', a.\ a\#e' \Rightarrow \mathsf{app}(\mathsf{lam}([a]e), e') = \mathsf{subst}(e, a, e')$$

where subst is a function symbol denoting explicit substitution (defined via a set of equations, see Section 2.1). Alternatively using quantifier alternation we can specify the β-axiom as follows:

$$\forall e'.\mathcal{N}a.\forall e.\mathsf{app}(\mathsf{lam}([a]e)), e') = \mathsf{subst}(e, a, e')\ ,$$

or, in NML, even as an equational axiom using $\mathcal{N}$ and @:

$$\forall e'\colon Exp.\forall e\colon [Var]Exp.\mathsf{app}(\mathsf{lam}(e), e') = \mathcal{N}a.\mathsf{subst}(e@a, a, e')\ .$$

6 ADDITIONAL RELATED WORK

Matching logic [Roşu 2017] is the foundation of the $\mathbb{K}$ semantic framework, which has been used to specify programming languages (e.g., Java and C), virtual machines (e.g. KEVM [Hildenbrandt et al. 2018]), amongst others. Variants of matching logic have been defined to include recursive pattern definitions (see Matching μ-logic [Chen and Rosu 2019]) and facilitate automated reasoning [Chen et al. 2020]. To specify programming languages that include binding operators, matching logic uses an encoding based on the internalisation of the graph of a function from bound name to expressions containing bound names [Chen and Rosu 2020]. This is sufficient to formalise the semantics of binding operators, but it is unclear how to reason by induction on the defined syntax.

Another approach to encode binders in matching logic is to use numerical codes to represent bound variables, as in de Bruijn encodings of the λ-calculus. This approach works well from the computational point of view, but reasoning on de Bruijn's syntax is generally considered less intuitive than on syntax with names. On the other hand, reasoning about "raw" syntax without any support for alpha-equivalence is also painful because operations that intuitively seem like functions, such as substitution, need to be treated with care. Because intuitive reasoning about names and binding is

challenging, over several decades many other techniques for representing and reasoning about binding have been investigated, including higher-order abstract syntax [Pfenning and Elliott 1988], and locally-nameless encodings [Aydemir et al. 2008], to name just a few. Could they be combined with matching logic instead? For de Bruijn encodings the answer is certainly yes, but this would do little to overcome the gap between on-paper and formal reasoning. For other approaches, particularly higher-order abstract syntax, where the typed lambda-calculus is used as a metalanguage and its binding structure used to encode bindings in object languages, the answer is less clear: induction and recursion over higher-order encodings has been a long-standing challenge [Despeyroux et al. 1995; Despeyroux and Hirschowitz 1994; Despeyroux et al. 2001; Röckl et al. 2001] with satisfactory solutions such as Beluga [Pientka 2008, 2010] and Abella [Baelde et al. 2014b] only appearing fairly recently. Though this is not an insurmountable obstacle, supporting induction or recursion over HOAS has necessitated developing significant new logical or semantic foundations, for example *contextual modal type theory* for Beluga [Nanevski et al. 2008] and logics for reasoning about generic judgments and nominal abstraction for Abella [Gacek et al. 2011; McDowell and Miller 2002; Miller and Tiu 2005; Tiu 2007]. Both of these go well beyond vanilla first-order logic so the prospect of combining them with matching logic seems much more speculative than for nominal logic. Other techniques, such as locally nameless encodings [Aydemir et al. 2008], appear to have no obvious obstacle to being adapted for use with matching logic, but there is no systematic logical foundation for locally nameless encodings analogous to nominal logic, instead one can view them as a recipe for employing de Bruijn indices for bound names while using ordinary names for free names such that induction and recursion principles can be proved.

The focus of this paper is on the question of how to adapt matching logic to accommodate binding. Naturally, there are many techniques for encoding binding in theorem provers and logical frameworks that are not subject to this constraint, and we do not make any claim here that NML (or NLML) is better than the state of the art among such systems generally. For this reason we have not attempted to give a complete picture of this field, but refer to recent comparisons such as [Felty et al. 2018a] and [Felty et al. 2018b]. As attempts to reason about languages with binding in matching logic are in their infancy, with the first proposal appearing only two years ago, it seems premature to try to predict whether the end results will be competitive with established approaches such as Abella [Baelde et al. 2014a], locally nameless in Coq [Aydemir et al. 2008], Nominal Isabelle [Urban 2008], or Beluga [Pientka 2010], which are already used by nonspecialists for formalizing their own research. Each of these systems offers significant advantages that are not directly addressed by our work, for example support for automated reasoning, inductive proofs, and (for HOAS-based approaches) built-in capture-avoiding substitution.

Likewise, although implementation in concrete tools such as $\mathbb{K}$ is not in focus in this paper, for completeness we mention some other comparable specification and verification tools in this space such as Ott [Sewell et al. 2010], PLT Redex [Felleisen et al. 2009], Bedwyr [Baelde et al. 2007], and αProlog/αCheck [Cheney and Momigliano 2017; Cheney and Urban 2008b]. Ott is a tool primarily aimed at

easing the work involved in managing large semantics specifications, and provides a convenient concrete syntax for terms and rules that is parsed to abstract syntax that can then be transformed to LaTeX for presentation or publication purposes, or to specifications suitable for various theorem provers, such as Isabelle/HOL or Coq. Ott provides some support for binding and can generate specifications that use popular techniques such as locally-nameless [Aydemir et al. 2008], but focuses on specifying and representing syntax and rules, not constructing or checking proofs, and Ott does not have an underlying logic in which properties of Ott specifications could be specified or proved. Likewise, PLT Redex [Felleisen et al. 2009] is a domain-specific language built on Racket that supports specifying languages with conventional abstract syntax, evaluation rules and typing rules. It is again primarily focused on making it easy to define and experiment with specifications, including visualization of how expressions evaluate, but its support for binding is somewhat limited: binding operators are recognized to some extent and capture-avoiding substitution is provided as a built-in operation. Bedwyr [Baelde et al. 2007] is a system based on the logic of generic judgments [Tiu 2007] for specifying and model-checking properties of formal calculi including binding. Finally, αProlog is a typed logic programming language that provides built-in support for name binding using nominal logic. αProlog is not a full-fledged logical framework but can be used to model and check properties of specifications, using an exhaustive form of property-based testing [Cheney and Momigliano 2017]. Adapting ideas from (nominal) matching logic to logic programming may be an interesting future direction.

The variants of matching logic defined in this paper include names and name management primitives following the nominal approach, which supports inductive reasoning over syntax with binding. We propose three alternative ways of incorporating nominal primitives in matching logic: in the first one (section 3.1), we exploit the fact that nominal logic has been defined as a theory of first-order logic, which is definable in matching logic, in the second (section 3.1.5), we use the features of patterns to define nominal primitives, and in the third (section 3.2), we extend the syntax and the semantics of matching logic to include built-in nominal primitives. Each of these approaches permits to encode binders directly by using the abstraction construct of nominal logic, avoiding the use of functional encodings and inheriting the reasoning principles of nominal logic.

The treatment of concretion and locally scoped fresh names in NML is somewhat similar to the $\lambda\alpha\nu$-calculus of Pitts [2011]. The latter is a lambda-calculus extended with a binding abstraction operator $\alpha a.e$, concretion $e@a$, and locally scoped names $\nu a.e$; expressions are given a total semantics by interpreting over nominal sets enriched with an operation called *restriction*, which models the behavior of ν. Thus, in contrast to NML, concretion is total and satisfies a beta-equivalence law $(\alpha a.e)@b = \nu a.(a\,b)\cdot e$, but free occurrences of a in the left hand side (obtained by swapping with occurrences of b in e) will be replaced by a special name anon $= \nu a.a$. Our treatment of concretion and local fresh name scoping using Ⴎ leverages matching logic's ability to represent partial functions.

7 CONCLUSIONS

We have illustrated the expressive power of NML by using it to specify the β-reduction and typing relations in the λ-calculus as well as principles of induction to reason over λ-terms. In future we will study extensions of NML with fixed point operators, and will consider proof systems and computational properties (e.g., algorithms to check equivalence of patterns in NML). This is important to build theorem provers and programming language verification tools based on this logic. Regarding implementations of NLML and NML: the first can be directly implemented in 𝕂 since it is simply a theory in matching logic; the second requires an extension of 𝕂, which we are discussing with the designers of 𝕂. We expect the efficient techniques developed for nominal terms with ground names can be adapted to work with NML patterns.

REFERENCES

Brian E. Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering Formal Metatheory. In *POPL*. 3–15.

David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. 2014a. Abella: A System for Reasoning about Relational Specifications. *J. Formalized Reasoning* 7, 2 (2014), 1–89. https://doi.org/10.6092/issn.1972-5787/4650

David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. 2014b. Abella: A System for Reasoning about Relational Specifications. *Journal of Formalized Reasoning* 7, 2 (Jan. 2014), 1–89. https://doi.org/10.6092/issn.1972-5787/4650

David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. 2007. The Bedwyr System for Model Checking over Syntactic Expressions. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4603)*, Frank Pfenning (Ed.). Springer, 391–397. https://doi.org/10.1007/978-3-540-73595-3_28

Henk P. Barendregt. 1984. *The Lambda Calculus: its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics, Vol. 103. North-Holland.

Xiaohong Chen and Grigore Rosu. 2019. Matching μ-Logic. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*. 1–13. https://doi.org/10.1109/LICS.2019.8785675

Xiaohong Chen and Grigore Rosu. 2020. A general approach to define binders using matching logic. *Proc. ACM Program. Lang.* 4, ICFP (2020), 88:1–88:32. https://doi.org/10.1145/3408970

Xiaohong Chen, Minh-Thai Trinh, Nishant Rodrigues, Lucas Peña, and Grigore Roşu. 2020. Towards a Unified Proof Framework for Automated Fixpoint Reasoning Using Matching Logic. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 161 (Nov. 2020), 29 pages. https://doi.org/10.1145/3428229

James Cheney. 2005. A Simpler Proof Theory for Nominal Logic. In *FoSSaCS (LNCS, Vol. 3441)*. Springer, 379–394.

James Cheney. 2006. Completeness and Herbrand theorems for nominal logic. *J. Symb. Log.* 71 (2006), 299–320. Issue 1.

James Cheney. 2016. A simple sequent calculus for nominal logic. *Journal of Logic and Computation* 26, 2 (2016), 699–726.

James Cheney and Alberto Momigliano. 2017. αCheck: A mechanized metatheory model checker. *TPLP* 17, 3 (May 2017), 311–352.

James Cheney and Christian Urban. 2004. Alpha-Prolog: A Logic Programming Language with Names, Binding and Alpha-Equivalence. In *Proceedings of the 20th International Conference on Logic Programming (ICLP 2004) (LNCS, 3132)*, Bart Demoen and Vladimir Lifschitz (Eds.). Springer, 269–283.

James Cheney and Christian Urban. 2008a. Nominal Logic Programming. *ACM TOPLAS* 30, 5 (August 2008), 26.

James Cheney and Christian Urban. 2008b. Nominal logic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 5 (2008), 1–47. https://doi.org/10.1145/1387673.1387675

Joëlle Despeyroux, Amy P. Felty, and André Hirschowitz. 1995. Higher-Order Abstract Syntax in COQ. In *TLCA'95 (LNCS, Vol. 3461)*. 124–138.

Joëlle Despeyroux and André Hirschowitz. 1994. Higher-Order Abstract Syntax with Induction in COQ. In *LPAR'94 (LNCS, Vol. 822)*. Springer, 159–173.

Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. 2001. Primitive Recursion for Higher-Order Abstract Syntax. *Theor. Comp. Sci.* 266, 1-2 (September 2001), 1–57.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex.* MIT Press. http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=11885

Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. 2018a. Benchmarks for reasoning with syntax trees containing binders and contexts of assumptions. *Math. Struct. Comput. Sci.* 28, 9 (2018), 1507–1540. https://doi.org/10.1017/S0960129517000093

Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. 2018b. Benchmarks for reasoning with syntax trees containing binders and contexts of assumptions. *Math. Struct. Comput. Sci.* 28, 9 (2018), 1507–1540. https://doi.org/10.1017/S0960129517000093

Maribel Fernández and Murdoch J. Gabbay. 2007. Nominal Rewriting. *Inf. Comput.* 205, 6 (June 2007), 917–965.

Murdoch J. Gabbay. 2007. A General Mathematics of Names. *Information and Computation* 205, 7 (July 2007), 982–1011.

Murdoch J. Gabbay and Andrew M. Pitts. 2001. A New Approach to Abstract Syntax with Variable Binding. *Formal Aspects of Computing* 13, 3–5 (July 2001), 341–363.

Andrew Gacek, Dale Miller, and Gopalan Nadathur. 2011. Nominal abstraction. *Inf. Comput.* 209, 1 (2011), 48–73.

Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *CSF 2018*. IEEE, 204–217. https://doi.org/10.1109/CSF.2018.00022

Raymond McDowell and Dale Miller. 2002. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. Comput. Log.* 3, 1 (2002), 80–136. https://doi.org/10.1145/504077.504080

Dale Miller and Alwen Tiu. 2005. A proof theory for generic judgments. *ACM Trans. Comput. Log.* 6, 4 (2005), 749–783. https://doi.org/10.1145/1094622.1094628

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual Modal Type Theory. *TOCL* 9, 3 (2008), 23:1–23:49.

Frank Pfenning and Conal Elliott. 1988. Higher-order Abstract Syntax. In *PLDI*. ACM Press, 199–208.

Brigitte Pientka. 2008. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. *SIGPLAN Notices* 43, 1 (2008), 371–382.

Brigitte Pientka. 2010. Beluga: Programming with Dependent Types, Contextual Data, and Contexts. In *FLOPS*. 1–12. https://doi.org/10.1007/978-3-642-12251-4_1

Andrew M. Pitts. 2003. Nominal Logic, A First Order Theory of Names and Binding. *Inf. Comput.* 186, 2 (2003), 165–193.

Andrew M. Pitts. 2006. Alpha-structural Recursion and Induction. *J. ACM* 53, 3 (2006), 459–506.

Andrew M. Pitts. 2011. Structural recursion with locally scoped names. *J. Funct. Program.* 21, 3 (2011), 235–286. https://doi.org/10.1017/S0956796811000116

Andrew M. Pitts. 2013. *Nominal Sets: Names and Symmetry in Computer Science.* Cambridge University Press, USA.

Christine Röckl, Daniel Hirschkoff, and Stefan Berghofer. 2001. Higher-Order Abstract Syntax with Induction in Isabelle/HOL: Formalizing the pi-Calculus and Mechanizing the Theory of Contexts. In *FoSSaCS*. Springer, 364–378.

Grigore Roşu. 2017. Matching logic. *Logical Methods in Computer Science* 13, 4 (December 2017), 1–61. https://doi.org/abs/1705.06312

Grigore Rosu. 2017. 𝕂: A Semantic Framework for Programming Languages and Formal Analysis Tools. In *Dependable Software Systems Engineering*. 186–206. https://doi.org/10.3233/978-1-61499-810-5-186

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strnisa. 2010. Ott: Effective tool support for the working semanticist. *J. Funct. Program.* 20, 1 (2010), 71–122. https://doi.org/10.1017/S0956796809990293

M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. 2003. FreshML: Programmming with Binders Made Simple. In *ICFP*. ACM Press, 263–274.

Alwen Tiu. 2007. A Logic for Reasoning about Generic Judgments. *ENTCS* 174, 5 (2007), 3–18.

Christian Urban. 2008. Nominal Techniques in Isabelle/HOL. *J. Autom. Reason.* 40, 4 (May 2008), 327–356.

Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. 2004. Nominal Unification. *Theor. Comp. Sci.* 323, 1–3 (2004), 473–497.

A PROOFS

PROOF OF PROP. 3.1. It is easy to see that axioms (E3) and (E5) are particular cases of axiom (EV): function symbols and predicates are symbols in the signature Σ_{NLML}.

Axiom (E1) is also a particular case of axiom (EV) since swapping is represented as a symbol in the signature Σ_{NLML} (it is a particular σ of arity 3: two names and a pattern in which we are swapping the names).

Axiom (E2) does not exactly translate to (EV), but using axiom (P) we can show that $a \# x = (b\ b') \cdot (a \# x)$ and then using (EV) the desired result follows. Axiom (E4) is handled similarly. □

PROOF OF PROP. 3.2. The translation function is defined inductively in the obvious way using the mapping between symbols in the nominal logic and NLML signatures described above, resulting in a pattern of sort *Pred* since this is the type of the result for predicate symbols and formula constructors.

Part 1 follows from the fact that FOL can be represented as a theory in matching logic using the *Pred* sort introduced above (see Chen and Rosu [2019] for the definition of the FOL theory in matching logic), and the axioms of nominal logic can be represented as a theory in matching logic too, as a consequence of Proposition 3.1.

Part 2 follows from completeness in first-order logic and nominal logic [Cheney 2006] using $\neg\phi$: the formula $\neg\phi$ is provable, and by part 1, $\neg\phi' = \top$ holds in matching logic, and therefore $\phi' = \bot$ holds also. □

PROOF OF PROP. 3.3. Using (∗) we can write (A1′) as follows

$$(\exists y.y \wedge [b]y = [a]x) = ((a\ b) \cdot x \wedge b \#^\dagger [a]x)$$

Note that $b \# [a]x = (b = a \vee b \# x)$, so the above pattern is equal to:

$$((\exists y.y \wedge [b]y = [a]x) = ((a\ b) \cdot x \wedge a =^\dagger b)) \vee ((\exists y.y \wedge [b]y = [a]x) = ((a\ b) \cdot x \wedge b \#^\dagger x))$$

By (A1) in the first disjunct we must have $x = y$ so the formula simplifies to $\top$ using (S1). In the second disjunct, using (A1) we must have $y = (a\ b) \cdot x \wedge b \#^\dagger x$, so the formula again simplifies to $\top$ by instantiating y.

Using (∗) we can write (A2′) as follows:

$$x = \exists a.a \#^\dagger x \wedge [a](\exists y.y \wedge [a]y = x)$$

Using (A2) we know we can choose b, z such that $x = [b]z$. We thus need to show that for any such choice of b, z we have:

$$[b]z = \exists a.a \#^\dagger [b]z \wedge [a](\exists y.y \wedge [a]y = [b]z)$$

We can again consider two cases: $a = b$ or $a \# z$. In the first case the formula simplifies as follows:

$$[b]z = [b](\exists y.y \wedge [b]y = [b]z)$$

and by (A1) we have $y = z$ so the whole formula simplifies to $\top$ on instantiating $y = z$. In the second case the formula simplifies to:

$$[b]z = \exists a.a \#^\dagger z \wedge [a](\exists y.y \wedge [a]y = [b]z)$$

where using (A1) we can replace $[a]y = [b]z$ with $y = (a\ b) \cdot z \wedge a \# z$. Thus the formula simplifies further to

$$[b]z = \exists a.a \#^\dagger z \wedge [a](a\ b) \cdot z$$

To complete the proof we note that an a fresh for z can always be chosen, discharging the freshness assumption and again using (A1) we can conclude that $[b]z = [a](a\ b) \cdot z$ since $b \# (a\ b) \cdot z$ follows from $a \# z$ by equivariance.

To show (A1) we need to show:

$$[a]x = [b]y \iff (a = b \wedge x = y) \vee (a \# y \wedge x = (a\ b) \cdot y)$$

In the forward direction, suppose a, b, x, y given with $[a]x = [b]y$. Let $c \# a, b, x, y$ be a fresh name; then $([a]x)@c = ([b]y)@c$. Using (A1′) this equation is equivalent to $((a\ c) \cdot x \wedge c \# [a]x) = ((b\ c) \cdot y \wedge c \# [b]y)$. Since c is fresh for everything in sight the freshness constraints simplify away and we have $(a\ c) \cdot x = (b\ c) \cdot y$. We now consider two cases. If $a = b$ then $(a\ c) \cdot x = (a\ c) \cdot y$ implies $x = y$. Otherwise, notice that $a \# [a]x = [b]y$ and if $a \neq b$ the only way this can be the case is if $a \# y$. This, together with properties of swapping, means we can conclude $[c]((a\ c) \cdot x) = [c](a\ c) \cdot (a\ b) \cdot y = [c](c\ b) \cdot (a\ c) \cdot y = [(c\ b) \cdot b](c\ b) \cdot y = (c\ b) \cdot [b]y$, and since $a \# y$ we know $c \# [b]y$ so $(c\ b) \cdot [b]y = [b]y$. Otherwise we may assume $c \neq a$ and $c \# x$ and already know $b \neq a$ so both b and c are fresh for $[c](a\ c) \cdot x$ so we have $[c]((a\ c) \cdot x) = (b\ c) \cdot [c](a\ c) \cdot x = [b](b\ c) \cdot (a\ c) \cdot x = [b](a\ b) \cdot x = [b](a\ b) \cdot (a\ b) \cdot y = [b]y$. Since c was arbitrary, we can conclude that $\exists c.c \# [a]x \wedge [c](a\ c) \cdot x \subseteq [b]y$. The converse direction, showing that $\exists c.c \# [b]y \wedge [c](b\ c) \cdot y \subseteq [a]x$, is symmetric noting that $a \# x \wedge x = (a\ b) \cdot y$ is equivalent to $b \# x \wedge y = (a\ b) \cdot x$.

To show (A2) suppose x is a given abstraction and use (A2′) to choose a fresh name $a \# x$ such that $x = [a]x@a$; then (A2) holds with the existential quantifiers witnessed by a and $y = x@a$. □

PROOF OF PROP. 3.4. Each primed axiom is equivalent to the corresponding nominal logic axiom. For example $(F1')$ just says that swapping any two names matching $fresh(x)$ (i.e., both fresh for x) does not affect x. $(F2')$ just says that the names of sort α fresh for $a : \alpha$ are all those not matching a, while $(F3')$ says that the names of a different sort α' fresh for $a : \alpha$ are all names. The final axiom $(F4')$ just says $fresh(x)$ is nonempty for any x, i.e., a fresh name can always be chosen. Note that since we assume product sorts are part of any signature, τ may be a product of finitely many sorts, enabling us to emulate instances of axiom scheme $(F4)$ with multiple variables. □

PROOF OF PROP. 3.11. For part (1), suppose ρ is a suitable valuation. Then

$$
\begin{aligned}
\|[a](\phi_1 \wedge \phi_2)\|_\rho &= \{[a']v \mid v \in \|\phi_1 \wedge \phi_2\|_\rho\} \\
&= \{[a']v \mid v \in \|\phi_1\|_\rho \cap \|\phi_2\|_\rho\} \\
&= \{[a']v \mid v \in \|\phi_1\|_\rho\} \cap \{[a']v \mid v \in \|\phi_2\|_\rho\} \\
&= \|[a]\phi_1\|_\rho \cap \|[a]\phi_2\|_\rho \\
&= \|[a]\phi_1 \wedge [a]\phi_2\|_\rho
\end{aligned}
$$

and since ρ is arbitrary we can conclude $[a](\phi_1 \wedge \phi_2) \iff [a]\phi_1 \wedge [a]\phi_2$.

For part (2), similarly suppose suitable ρ is given and suppose $a : \alpha$ and $\phi : \tau$. Let $a' = \rho(a)$. Then

$$
\begin{aligned}
\|[a](\neg\phi)\|_\rho &= \{[a']v \mid v \in \|\neg\phi\|_\rho\} \\
&= \{[a']v \mid v \in M_\tau - \|\phi\|_\rho\} \\
&= \{[a']v \mid v \in M_\tau\} - \{[a']v \mid v \in \|\phi\|_\rho\} \\
&= \|[a]\top\|_\rho - \|[a]\phi\|_\rho \\
&= \|[a]\top \wedge \neg[a]\phi\|_\rho
\end{aligned}
$$

Since ρ was arbitrary we can conclude $[a](\neg\phi) \iff [a]\top \wedge \neg[a]\phi$.

Finally for part (3) consider again a suitable ρ and let $a' = rho(a)$ and $b' = \rho(b)$. Recall that since a and b are distinct names we must also have $a' \neq b'$ since ρ must be injective on names.

$$
\begin{aligned}
\|([a]\phi)@b\|_\rho &= \{v \mid [b']v \in \|[a]\phi\|_\rho \\
&= \{v \mid [b']v \in \{[a']w \mid w \in \|\phi\|_\rho\} \\
&= \{v \mid w \in \|\phi\|_\rho, [b']v = [a']w\} \\
&= \{v \mid w \in \|\phi\|_\rho, b' \# w \wedge v = (a'\ b') \cdot w\} \\
&= \{v \in M_{[v]\tau} \mid w \in \|\phi\|_\rho, b' \# [a']w\} \cap \{(a'\ b') \cdot v \mid v \in \|\phi\|_\rho \\
&= \{v \in M_{[v]\tau} \mid \star \in \{\star \mid w \in \|\phi\|_\rho \mid b' \# [a']w\}\} \cap \|(a\ b) \cdot \phi\|_\rho \\
&= \{v \in M_{[v]\tau} \mid \star \in \|b \# [a]\phi\|_\rho\} \cap \|(a\ b) \cdot \phi\|_\rho \\
&= \|b \# [a]\phi\|_\rho \cap \|(a\ b) \cdot \phi\|_\rho \\
&= \|(b \# [a]\phi) \wedge (a\ b) \cdot \phi\|_\rho
\end{aligned}
$$

Since ρ was arbitrary, this completes the proof that $([a]\phi)@b \iff (b \# [a]\phi) \wedge (a\ b) \cdot \phi$. □

PROOF OF THM. 3.12. By induction on the definition of patterns.

- The cases for names and variables are trivial: by Definition 3.8, if e is a name or a variable then $(a\ a') \cdot \|e\|_\rho = (a\ a') \cdot \{\rho(e)\} = \{(a\ a') \cdot \rho(e)\} = \|e\|_{(a\ a') \cdot \rho}$.
- If $\phi_\tau = \sigma(\phi_{\tau_1}, \ldots, \phi_{\tau_n})$ the result follows by induction and the fact that the interpretation of σ is equivariant (see Def. 3.6).
- If $\phi_\tau = \phi_1 \wedge \phi_2$ or $\phi_\tau = \neg\phi'$ the result follows directly by induction. For example

$$
\begin{aligned}
(a\ a') \cdot \|\phi \wedge \phi'\|_\rho &= (a\ a') \cdot (\|\phi\|_\rho \cap \|\phi'\|_\rho) \\
&= (a\ a') \cdot \|\phi\|_\rho \cap (a\ a') \cdot \|\phi'\|_\rho \\
&= \|\phi\|_{(a\ a') \cdot \rho} \cap \|\phi'\|_{(a\ a') \cdot \rho} \\
&= \|\phi \wedge \phi'\|_{(a\ a') \cdot \rho} .
\end{aligned}
$$

- For existential patterns $\exists x : \tau.\phi$ we reason as follows.

$$
\begin{aligned}
(a\ a') \cdot \|\exists x : \tau.\phi\|_\rho &= (a\ a') \cdot \bigcup_{v \in M_\tau} \|\phi\|_{\rho[v/x]} = \bigcup_{v \in M_\tau} (a\ a') \cdot \|\phi\|_{\rho[v/x]} \\
&= \bigcup_{v \in M_\tau} \|\phi\|_{(a\ a') \cdot \rho[v/x]} \\
&= \bigcup_{(a\ a) \cdot v \in (a\ a') \cdot M_\tau} \|\phi\|_{((a\ a') \cdot \rho)[(a\ a') \cdot v/x]} \\
&= \bigcup_{w \in M_\tau} \|\phi\|_{((a\ a') \cdot \rho)[w/x]} \\
&= \|\exists x : \tau.\phi\|_{(a\ a') \cdot \rho}
\end{aligned}
$$

where in addition to the induction hypothesis we use the fact that a nominal set M_τ is closed under swappings so that $M_\tau = (a\ b) \cdot M_\tau$.

- For the case of $\text{И}a : \alpha.\phi$ we reason as follows.

$$
\begin{aligned}
(a\,a') \cdot \|\text{И}a : \alpha.\phi\|_\rho \; &= \; (a\,a') \cdot \bigcup_{b\in\mathbb{A}_\alpha-supp(\rho)}\{v \in \|\phi\|_{\rho[b/a]} \mid b \notin supp(v)\} \\
&= \; \bigcup_{b\in\mathbb{A}_\alpha-supp(\rho)}\{(a\,a') \cdot v \in (a\,a') \cdot \|\phi\|_{\rho[b/a]} \mid b \notin supp(v)\} \\
&= \; \bigcup_{b\in\mathbb{A}_\alpha-supp(\rho)}\{(a\,a') \cdot v \in \|\phi\|_{(a\,a')\cdot\rho[b/a]} \mid b \notin supp(v)\} \\
&= \; \bigcup_{(a\,a')\cdot b\in(a\,a')\cdot(\mathbb{A}_\alpha-supp(\rho))}\{(a\,a') \cdot v \in \|\phi\|_{((a\,a')\cdot\rho)[(a\,a')\cdot b/a]} \mid b \notin supp(v)\} \\
&= \; \bigcup_{b'\in\mathbb{A}_\alpha-supp((a\,a')\cdot\rho)}\{w \in \|\phi\|_{((a\,a')\cdot\rho)[b'/a]} \mid b' \notin supp(w)\} \\
&= \; \|\text{И}a : \alpha.\phi\|_{(a\,a')\cdot\rho}
\end{aligned}
$$

Here we make use of the fact that $M_\alpha = \mathbb{A}_\alpha$ is a nominal set hence closed under swappings, as well as the fact that set difference and the support operation are equivariant.

$\square$

Proof of Prop. 3.14. The first part is a direct consequence of the definition of $\|\text{И}a.\phi\|_\rho$ (Definition 3.8): if a is not free in ϕ then $\|\phi\|_\rho = \|\phi\|_{\rho[a/a]}$ for any a.

The second part also follows from Definition 3.8: Both patterns are interpreted by

$$
\bigcup_{\substack{a \in \mathbb{A}_\alpha - supp(\rho) \\ b \in \mathbb{A}_{\alpha'} - supp(\rho) \\ a \neq b}} \{v \in \|\phi_\tau\|_{\rho[a/a][b/b]} \mid a \notin supp(v), b \notin supp(v)\}
$$

since the conditions on a, b ensure $(a\,b) \cdot v = v$ and $(a\,b) \cdot \rho[a/a][b/b] = \rho[a/b][b/a]$.

$\square$

Proof of Prop. 3.15. (1) We can show that both patterns have the same semantics using Theorem 3.12 (Equivariance).

By Definition 3.8 (semantics of И and $\wedge$ patterns):

$$
\|\text{И}a\colon \alpha.\phi_\tau \wedge \psi_\tau\|_\rho = \bigcup_{a\in\mathbb{A}_\alpha-supp(\rho)} \{v \in \|\phi_\tau\|_{\rho[a/a]} \cap \|\psi_\tau\|_{\rho[a/a]} \mid a \notin supp(v)\}.
$$

The latter is equal to $\bigcap_{a\in\mathbb{A}_\alpha-supp(\rho)}\{v \in \|\phi_\tau\|_{\rho[a/a]} \cap \|\psi_\tau\|_{\rho[a/a]} \mid a \notin supp(v)\}$. To see that the union can be turned into an intersection, it is sufficient to consider two different a, a' in $\mathbb{A}_\alpha - supp(\rho)$ such that $a, a' \notin supp(v)$ and observe that $v \in \|\phi_\tau\|_{\rho[a/a]}$ if and only if $v = (a\,a') \cdot v \in \|\phi_\tau\|_{(a\,a')\cdot(\rho[a/a])} = \|\phi_\tau\|_{\rho[a'/a]}$ by Theorem 3.12 (and similarly for ψ_τ). Therefore

$$
\begin{aligned}
\|\text{И}a\colon \alpha.\phi_\tau \wedge \psi_\tau\|_\rho \; = \; &\bigcap_{a\in\mathbb{A}_\alpha-supp(\rho)} \{v \in \|\phi_\tau\|_{\rho[a/a]} \mid a \notin supp(v)\} \cap \\
&\bigcap_{a\in\mathbb{A}_\alpha-supp(\rho)} \{v \in \|\psi_\tau\|_{\rho[a/a]} \mid a \notin supp(v)\}.
\end{aligned}
$$

Using the same reasoning again, we can write:

$$
\begin{aligned}
\|\text{И}a\colon \alpha.\phi_\tau \wedge \psi_\tau\|_\rho \; = \; &\bigcup_{a\in\mathbb{A}_\alpha-supp(\rho)} \{v \in \|\phi_\tau\|_{\rho[a/a]} \mid a \notin supp(v)\} \cap \\
&\bigcup_{a\in\mathbb{A}_\alpha-supp(\rho)} \{v \in \|\psi_\tau\|_{\rho[a/a]} \mid a \notin supp(v)\} \\
= \; &\|\rho\|_{\text{И}a\colon \alpha.\phi_\tau\wedge\text{И}a\colon \alpha.\psi_\tau}.
\end{aligned}
$$

Notice that by Prop. 3.14, $\text{И}a.(\phi \wedge \psi) \iff ((\text{И}a.\phi) \wedge \psi)$ if a is not free in ψ.

(2) This case also relies on the equivariance of the semantics (Theorem 3.12):

$$
\begin{aligned}
\|\neg\text{И}a\colon \alpha.\phi_\tau\|_\rho \; &= \; M_\tau - \|\text{И}a\colon \alpha.\phi_\tau\|_\rho \\
&= \; M_\tau - \bigcup_{a\in\mathbb{A}_\alpha-supp(\rho)} \{v \in \|\phi_\tau\|_{\rho[a/a]} \mid a \notin supp(v)\}
\end{aligned}
$$

$$
\begin{aligned}
\|\text{И}a\colon \alpha.\neg\phi_\tau\|_\rho \; &= \; \bigcup_{a\in\mathbb{A}_\alpha-supp(\rho)} \{v \in \|\neg\phi_\tau\|_{\rho[a/a]} \mid a \notin supp(v)\} \\
&= \; \bigcup_{a\in\mathbb{A}_\alpha-supp(\rho)} \{v \in M_\tau - \|\phi_\tau\|_{\rho[a/a]} \mid a \notin supp(v)\} \\
&= \; M_\tau - \bigcap_{a\in\mathbb{A}_\alpha-supp(\rho)} \{v \in \|\phi_\tau\|_{\rho[a/a]} \mid a \notin supp(v)\}
\end{aligned}
$$

The union and intersection above are equivalent, as shown in the previous case: If we consider two different elements a and a' in $\mathbb{A}_\alpha - supp(\rho)$ such that $a, a' \notin supp(v)$ then $v \in \|\phi_\tau\|_{\rho[a/\mathrm{a}]}$ if and only if $v = (a\,a') \cdot v \in \|\phi_\tau\|_{(a\,a') \cdot \rho[a/\mathrm{a}]} = \|\phi_\tau\|_{\rho[a'/\mathrm{a}]}$ (Theorem 3.12).

(3) By Definition 3.8,

$$\|[\mathrm{a}]\text{И}\mathrm{b}\colon \alpha.\phi_\tau\|_\rho = [\rho(\mathrm{a})] \bigcup_{b \in \mathbb{A}_\alpha - supp(\rho)} \{v \in \|\phi_\tau\|_{\rho[b/\mathrm{b}]} \mid b \notin supp(v)\}$$

$$= \bigcup_{b \in \mathbb{A}_\alpha - supp(\rho)} \{[\rho(\mathrm{a})]v \mid v \in \|\phi_\tau\|_{\rho[b/\mathrm{b}]}, b \notin supp(v)\}$$

Also by Definition 3.8,

$$\|\text{И}\mathrm{b}\colon \alpha.[\mathrm{a}]\phi_\tau\|_\rho = \bigcup_{b \in \mathbb{A}_\alpha - supp(\rho)} \{w \in \|[\mathrm{a}]\phi_\tau\|_{\rho[b/\mathrm{b}]} \mid b \notin supp(w)\}$$

$$= \bigcup_{b \in \mathbb{A}_\alpha - supp(\rho)} \{[\rho(\mathrm{a})]v \mid v \in \|\phi_\tau\|_{\rho[b/\mathrm{b}]}, b \notin supp(v)\}$$

(4) By Definition 3.8, $\|\langle \text{И}\mathrm{a}.\phi, \psi \rangle\|_\rho = \langle \bigcup_{a \in \mathbb{A}_\alpha - supp(\rho)} \{v \in \|\phi\|_{\rho[a/\mathrm{a}]} \mid a \notin supp(v)\}, \|\psi\|_\rho \rangle$.
Since a does not occur free in ψ, $\|\psi\|_{\rho[a/\mathrm{a}]} = \|\psi\|_\rho$ for any a. Hence,

$$\|\langle \text{И}\mathrm{a}.\phi, \psi \rangle\|_\rho = \bigcup_{a \in \mathbb{A}_\alpha - supp(\rho)} \{\langle v, w \rangle \mid v \in \|\phi\|_{\rho[a/\mathrm{a}]}, w \in \|\psi\|_{\rho[a/\mathrm{a}]}, a \notin supp(v)\} \;.$$

Also by Definition 3.8,

$$\|\text{И}\mathrm{a}.\langle \phi, \psi \rangle\|_\rho = \bigcup_{a \in \mathbb{A}_\alpha - supp(\rho)} \{\langle v, w \rangle \mid v \in \|\phi\|_{\rho[a/\mathrm{a}]}, w \in \|\psi\|_{\rho[a/\mathrm{a}]}, a \notin supp(v, w)\} \;.$$

The only difference between the two sets is the fact that in the second one we require $a \notin supp(w)$. Therefore every element in the second set is also in the first. We need to show that every element in the first set is also in the second: Assume $\langle v, w \rangle$ is in the first set and was obtained using a. If $a \notin supp(v, w)$ then $\langle v, w \rangle$ is also in the second set and we are done. If $a \in supp(w)$ there exists a' such that $a' \notin supp(w, v, \rho)$. Since a is not free in ψ, $\|\psi\|_{\rho[a'/\mathrm{a}]} = \|\psi\|_{\rho[a/\mathrm{a}]}$, hence $w \in \|\psi\|_{\rho[a'/\mathrm{a}]}$. By equivariance, $(a\,a') \cdot v \in \|\phi\|_{(a\,a') \cdot \rho[a/\mathrm{a}]} = \|\phi\|_{\rho[a'/\mathrm{a}]}$. Hence, $\langle (a\,a') \cdot v, w \rangle$ is in the second set and this is $\langle v, w \rangle$ since $a, a' \notin supp(v)$.
In the more general case:
By Definition 3.8,

$$\|\sigma(\text{И}\mathrm{a}.\phi_1, \ldots, \text{И}\mathrm{a}.\phi_n)\|_\rho = \sigma_M(\|\text{И}\mathrm{a}.\phi_1\|_\rho, \ldots, \|\text{И}\mathrm{a}.\phi_n\|_\rho)$$

$$= \sigma_M\left(\bigcup_{a \in \mathbb{A}_\alpha - supp(\rho)} \{v \in \|\phi_1\|_{\rho[a/\mathrm{a}]} \mid a \notin supp(v)\}, \ldots, \bigcup_{a \in \mathbb{A}_\alpha - supp(\rho)} \{v \in \|\phi_n\|_{\rho[a/\mathrm{a}]}) \mid a \notin supp(v)\} \right)$$

and

$$\|\text{И}\mathrm{a}.\sigma(\phi_1, \ldots, \phi_n)\|_\rho = \bigcup_{a \in \mathbb{A}_\alpha - supp(\rho)} \{v \in \sigma_M(\|\phi_1\|_{\rho[a/\mathrm{a}]}, \ldots, \|\phi_n\|_{\rho[a/\mathrm{a}]}) \mid a \notin supp(v)\} \;.$$

Again we need to show that both sets are the same, and in one direction the result is direct (if an atom is not in the support of $v_1, \ldots, v_n$, it cannot be in the support of $\sigma_M(v_1, \ldots, v_n)$ since σ_M is equivariant). In the other direction, the result follows from the assumption $\forall x_1, \ldots, x_n, \mathrm{a}, \mathrm{a}\#\sigma(x_1, ..., x_n) \Rightarrow \mathrm{a}\#x_1 \land \ldots \land \mathrm{a}\#x_n$: if a is not in the support of $\sigma_M(v_1, \ldots, v_n)$ then it is not in the support of $v_1, \ldots, v_n$.

$\square$

PROOF OF PROP. 3.16. To show that two patterns ϕ_τ and ψ_τ are equivalent, we need to show they have the same instances in M_τ for any given ρ.

To prove the first equivalence, assume $v \in M_\tau$ matches $\text{И}\mathrm{a}\colon \alpha.\phi_\tau$, that is, there is some $a \in \mathbb{A}_\alpha - supp(\rho)$ such that $v \in \|\phi_\tau\|_{\rho[a/\mathrm{a}]}$ and $a \notin supp(v)$. Hence, there exists $a \in \mathbb{A}_\alpha$ such that a is fresh for $\rho(\bar{\mathrm{b}}, \bar{x}), v$ and $v \in \|\phi_\tau\|_{\rho[a/\mathrm{a}]}$. Therefore, by definition of ρ (existential and conjunction cases), we deduce that v matches $\exists z_\mathrm{a}\colon \alpha.((\exists y\colon \tau.y \land z_\mathrm{a}\#(\bar{\mathrm{b}}, \bar{x}, y)) \land \phi_\tau\{\mathrm{a} \mapsto z_\mathrm{a}\}$.

Similarly, if $v \in M_\tau$ matches $\exists z_\mathrm{a}\colon \alpha.((\exists y\colon \tau.y \land z_\mathrm{a}\#(\bar{\mathrm{b}}, \bar{x}, y)) \land \phi_\tau\{\mathrm{a} \mapsto z_\mathrm{a}\}$, then there is a value $a \in \mathbb{A}_\alpha$ such that v matches both conjuncts, that is, a is fresh for $\|\bar{\mathrm{b}}, \bar{x}\|_\rho$, v and v is in $\|\phi_\tau\{\mathrm{a} \mapsto z_\mathrm{a}\}\|_{\rho[a/z_\mathrm{a}]}$. If a is fresh for $supp(\rho)$ then we are done, as $v \in \|\text{И}\mathrm{a}\colon \alpha.\phi_\tau\|_\rho$. If a is not fresh for $supp(\rho)$ (i.e., it occurs in the image of some variable or name that does not occur free in ϕ_τ), there exists a' such that $a' \in \mathbb{A} - supp(\rho)$ and a' is also fresh for v (hence $(a\,a') \cdot v = v$). By the equivariance theorem, $\|\phi_\tau\{\mathrm{a} \mapsto z_\mathrm{a}\}\|_{(a\,a') \cdot (\rho[a/z_\mathrm{a}])} = (a\,a') \cdot v = v$. Since $\|\phi_\tau\{\mathrm{a} \mapsto z_\mathrm{a}\}\|_{(a\,a') \cdot (\rho[a/z_\mathrm{a}])} = \|\phi_\tau\|_{\rho[a'/z_\mathrm{a}]}$ (because $a' \notin supp(\rho)$ and a is not in the support of free names or variables in ϕ_τ), we deduce $v \in \rho(\text{И}\mathrm{a}\colon \alpha.\phi_\tau)$ as required.

In the second equivalence, $\Rightarrow$ is the pattern implication operation, defined as $\phi \Rightarrow \psi \Leftrightarrow \neg(\phi) \vee \psi$. The semantics of $\phi \Rightarrow \psi$ is $M_\tau - (\|\phi\|_\rho - \|\psi\|_\rho)$, i.e., the set of values that either don't match ϕ or do match ψ, i.e. the set of values v such that if v matches ϕ then v matches ψ also. The pattern $\forall x : \tau'.\phi_\tau$ is syntactic sugar for $\neg \exists x : \tau'.\neg\phi_\tau(x)$, interpreted as the complement of the union for each $v \in M_{s'}$ of values in M_τ that do not match $\|\phi\|_{\rho[v/x]}$.

Assume $v \in M_\tau$ matches $\mathsf{V}a \colon \alpha.\phi_\tau(\mathsf{a}, \bar{\mathsf{b}}, \bar{x})$. Then there is a value $a \in \mathbb{A}_\alpha$ such that a is fresh for $\|\bar{x}\|_\rho, \|\bar{\mathsf{b}}\|_\rho, v$ and v is in $\|(\|_{\rho[a/z_\mathsf{a}]}\phi_\tau(z_\mathsf{a}, \bar{\mathsf{b}}, \bar{x}))$.

Consider now a different value $a' \in \mathbb{A}_\alpha$, we want to prove that v still satisfies the implication pattern. If v does not satisfy the existential for a' (i.e., a' is not fresh for $\rho(\bar{x}), \rho(\bar{\mathsf{b}}), v$) then this is trivially true. Otherwise, a' is fresh for $\rho(\bar{x}), \rho(\bar{\mathsf{b}}), v$ and by equivariance (Theorem 3.12) v must also match $\|(\|_{\rho[a'/z_\mathsf{a}]}\phi_\tau(z_\mathsf{a}, \bar{\mathsf{b}}, \bar{x}))$.

In the other direction, assume $v \in M_\tau$ matches $\forall z_\mathsf{a} \colon \alpha.((\exists y \colon \tau.y \wedge z_\mathsf{a}\#(\bar{x}, \bar{\mathsf{b}}, y)) \Rightarrow \phi_\tau(z_\mathsf{a}, \bar{\mathsf{b}}, \bar{x}))$. The nominal set semantics ensures that there exists some $a \in \mathbb{A}_\alpha$ that is fresh for $\rho(\bar{x}), \rho(\bar{\mathsf{b}}), v$, that is, v matches the left-hand side of the implication pattern, therefore v must match $\|(\|_{\rho[a/z_\mathsf{a}]}\phi_\tau(z_\mathsf{a}, \bar{\mathsf{b}}, \bar{x}))$. Hence, v matches $\exists z_\mathsf{a} \colon \alpha.((\exists y \colon \tau.y \wedge z_\mathsf{a}\#(\bar{x}, \bar{\mathsf{b}}, y)) \wedge \phi_\tau(z_\mathsf{a}, \bar{\mathsf{b}}, \bar{x}))$ and we are done by part 1.

We have shown that both patterns have the same instances, which completes the proof. □

PROOF OF PROP. 3.17. Using properties of V:

$$
\begin{aligned}
(\mathsf{V}a.[a]\phi)@b &= \mathsf{V}a.([a]\phi)@b && \text{by Prop. 3.15(4)} \\
&= \mathsf{V}a.(b \,\#\, [a]\phi \wedge (a\;b) \cdot \phi) && \text{by Prop. 3.11(3)} \\
&= (\mathsf{V}a.b \,\#\, [a]\phi) \wedge (\mathsf{V}a.(a\;b) \cdot \phi) && \text{by Prop. 3.15(1)} \\
&= (\mathsf{V}a.(b = a \vee b \,\#\, \phi)) \wedge (\mathsf{V}a.(a\;b) \cdot \phi) && \\
&= (\mathsf{V}a.b = a \vee \mathsf{V}a.b \,\#\, \phi) \wedge (\mathsf{V}a.(a\;b) \cdot \phi) && \text{by Prop. 3.15(1)} \\
&= (\mathsf{V}a.b \,\#\, \phi) \wedge (\mathsf{V}a.(a\;b) \cdot \phi) && \text{by Prop. 3.14 (1)} \\
&= \mathsf{V}a.(a\;b) \cdot \phi &&
\end{aligned}
$$

The last step follows from the semantics of freshness and V: Given a valuation ρ, if $\|b\|_\rho$ is not fresh in an instance $v \in \|[a]\phi\|_{\rho[a/a]}$ for some new a (i.e., the first conjunct is false), then $\|b\|_\rho \in supp(v)$ for all new a. Therefore $\|b\|_\rho$ is in the support of $\|\phi\|_{\rho[a/a]}$ for all new a, and hence $\mathsf{V}a.(a\;b) \cdot \phi$ has no instances. Otherwise, the first conjunct is true (interpreted as the whole carrier set) and the conjunction is therefore equivalent to the second conjunct. □