Nominal Matching Logic with Fixpoints

Mircea Sebe
University of Illinois at Urbana-Champaign
Champaign, USA
osebe2@illinois.edu

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

James Cheney
University of Edinburgh
Edinburgh, United Kingdom
jcheney@inf.ed.ac.uk

Abstract

Matching logic is the foundation of the K semantic environment for the specification of programming languages and automated generation of evaluators and verification tools. NLML is a formalization of nominal logic, which facilitates specification and reasoning about languages with binders, as a matching logic theory. Many properties of interest are inductive, and to prove them an induction principle modulo alpha-equality is required. In this paper we show that an alpha-structural Induction Principle for any nominal binding signature can be derived in an extension of NLML with set variables and fixpoint operators. We illustrate the use of the principle to prove properties of the λ-calculus, the computation model underlying functional programming languages. The techniques generalize to other languages with binders. The proofs have been written in and generated using Metamath Zero.

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

Keywords: Binding operator, Matching Logic, Nominal Logic, Lambda-Calculus, Verification, Induction, Metamath Zero

ACM Reference Format:
Mircea Sebe, Maribel Fernández, and James Cheney. 2025. Nominal Matching Logic with Fixpoints. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '25), January 20–21, 2025, Denver, CO, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3703595.3705872

1 Introduction

Matching Logic [11] provides a powerful framework for the study of programming language semantics. In this paper we consider extensions of Matching Logic with nominal features and fixpoints, and study proof systems for the extended logic. Prior work proposed adding nominal features to Matching Logic in two different ways [5]: NLML is obtained by including the axioms of Nominal Logic [8] as a *theory* in Matching Logic, whereas NML extends the syntax and semantics of Matching Logic to incorporate names, abstraction and the И quantifier as primitive notions. NLML and NML provide complementary advantages and disadvantages for the specification of languages with binders. The second approach has the advantage of allowing users to write more concise specifications taking advantage of the additional built-in operations, whereas the first has the advantage of being directly definable in Matching Logic without any changes to existing infrastructure. In both cases, proofs by structural induction on syntax with binders can be written in a natural way, which follows closely the structure of informal proofs.

The prior work introducing nominal extensions of Matching Logic presented induction principles for the λ-calculus as an example, but these have to be assumed as axioms in a first-order setting. In this paper, we aim to prove that the induction principles are correct, derivable from fixed point primitives, and can be integrated in verification environments. For this, first we consider an extension of Matching Logic with set variables and fixpoints [3] and show this is compatible with the nominal extensions. (As is becoming standard in the Matching Logic community, we consider the term "Matching Logic" to implicitly include set variables and fixpoints by default; we reserve the term "first-order" to refer to the baseline first-order version of ML, or of NLML.)

We focus in this paper on NLML. By adding fixpoints to NLML and developing a proof system for the extended logic, we obtain a more expressive logic where we are able to derive the induction principles required to reason on languages with binders. Interestingly, NLML extended with fixpoints supports a different flavour of reasoning compared to tools such as Nominal Isabelle, as the induction principle we get using the fixpoint operator μ does not have built-in freshness constraints in the cases for binders like the strong induction principles that are derived for nominal datatypes in prior work [14, 15]. Instead, we use standard fixed point induction rules, and instead make use of general properties of nominal logic such as freshness, equivariance, and name abstraction in appropriate places.

We first review basic concepts of nominal logic and matching logic and their semantics before recalling first-order NLML, obtained by adding nominal logic as a theory in matching logic (Sec. 2). We then discuss extensions with fixpoints for NLML and corresponding proof systems and illustrate the applications via standard lambda-calculus examples (Sec. 3). Finally we present details of a formalization of properties of substitution using fixpoint induction in NLML using Metamath Zero (Sec. 4). Sec. 5 discusses related work and Sec. 6 concludes. The most relevant parts of the formalization are included in the appendix.

2 Background

In this section we briefly recall the syntax and semantics of matching logic [3], nominal logic [8] and their combination: NLML [5]. In all 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. Here $Pred$ is a special sort symbol used for predicates. 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; we use the special sort $Pred$ and write $p \in \Sigma_{\tau_1,\ldots,\tau_n;Pred}$ to indicate that p is a predicate whose arguments have sorts $\tau_1, \ldots, \tau_n$.

2.1 Matching-Mu Logic (ML)

Matching-Mu Logic [3] is an extension of Matching Logic [11] with set variables and fixpoint operators. Formulas, called *patterns*, are built using variables, symbols from a signature, first-order logic connectives and quantifiers. Patterns are interpreted as sets, generalizing both terms and first-order logic formulas.

We consider a matching-mu logic signature $\Sigma = (S, \mathcal{V}ar, \Sigma)$, which we assume contains operations for product sorts and $Pred$. The set $\mathcal{V}ar$ includes two kinds of variables: element variables x and set variables X. Patterns are generated by the following grammar:

$$\begin{array}{rcl} \phi_\tau & ::= & x\colon \tau \mid \phi_\tau \wedge \psi_\tau \mid \neg\phi_\tau \mid \exists x\colon \tau'.\phi_\tau \\ & & \mid \sigma(\phi_{\tau_1}, \ldots, \phi_{\tau_n}) \mid X\colon \tau \mid \mu X\colon \tau.\phi_\tau \end{array}$$

where $x, X \in \mathcal{V}ar_\tau$ (x denotes an element variable and X a set variable), $\sigma \in \Sigma_{\tau_1,\ldots,\tau_n;\tau}$ and in a μ pattern, ϕ_τ is positive in $X\colon \tau$ (i.e., every free occurrence of $X\colon \tau$ is under an even number of negations). The meaning of a pattern of sort τ is a set of elements of sort τ that the pattern matches. 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, $\top_\tau \triangleq \exists x\colon \tau.x\colon \tau$ (since carriers are non-empty sets) and $\bot_\tau \triangleq \neg\top_\tau$. 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-Mu 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 that maps element variables to elements of the domain and set variables to subsets of the domain, respecting sorts: $\rho(x) \in M_\tau$ for each $x \in \mathcal{V}ar_\tau$ and $\rho(X) \in \mathcal{P}(M_\tau)$ for each $X \in \mathcal{V}ar_\tau$. We abbreviate it as $\rho : \mathcal{V}ar \to M$.

Given $\Sigma = (S, \mathcal{V}ar, \Sigma)$, a Matching-Mu Logic Σ-model M and valuation ρ, the extension $\| - \|_\rho$ to patterns is defined by:

- $\|x\|_\rho = \{\rho(x)\}$ for all $x \in \mathcal{V}ar$,
- $\|X\|_\rho = \rho(X)$ for all $X \in \mathcal{V}ar$,
- $\|\phi_1 \wedge \phi_2\|_\rho = \|\phi_1\|_\rho \cap \|\phi_2\|_\rho$,
- $\|\neg\phi_\tau\|_\rho = M_\tau - \|\phi_\tau\|_\rho$,
- $\|\exists x\colon \tau'.\phi_\tau\|_\rho = \bigcup_{a \in M_{\tau'}} \|\phi_\tau\|_{\rho[a/x]}$,
- $\|\sigma(\phi_{\tau_1}, \ldots, \phi_{\tau_n})\|_\rho = \overline{\sigma_M}(\|\phi_{\tau_1}\|_\rho, \ldots, \|\phi_{\tau_n}\|_\rho)$, where $\sigma \in \Sigma_{\tau_1,\ldots,\tau_n;\tau}$ and $\overline{\sigma_M}$ denotes the pointwise extension of σ_M:

$$\overline{\sigma_M}(V_1, \ldots, V_n) = \bigcup \{\sigma_M(v_1, \ldots, v_n) \mid v_1 \in V_1, \ldots, v_n \in V_n\}$$

- $\|\mu X\colon \tau.\phi_\tau\|_\rho = \mathbf{lfp}\ F$, where $F(S) = \|\phi_\tau\|_{\rho, X \mapsto S}$, that is, $\mu X\colon \tau.\phi_\tau$ denotes the least fixpoint of ϕ_τ.

Note that F is monotone since ϕ is positive in X, which guarantees the existence of the least fixpoint.

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 [3] 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 $\exists x : \tau.x \in^\dagger \phi$). Alternatively, the coercion operator can be defined using a primitive definedness symbol. The notation $\sigma\colon \tau \to \tau'$ indicates that σ is a *function*, in which case an appropriate axiom ($Function_\sigma$) for σ is assumed to be included in the theory stating that $\forall x.\exists y.\sigma(x) = y$. Since symbols σ can be partial or multivalued in matching logic, this axiom states that a definite value for $\sigma(x)$ exists and is unique.

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 ψ is functional, i.e., it evaluates to a singleton set, then $(\forall x.\phi) \Rightarrow \phi[\psi/x]$ is valid (see [3]). Substitution of patterns for set variables $\phi[\psi/X]$ is similar and standard, implicitly renaming μ-bound variables to avoid capture.

Hilbert-style proof systems for matching logic are available [3, 11]. We recall the system $\mathcal{H}_\mu$ from [3] in Figure 1.

We note the fact that Matching Logic at its core does not have a built-in notion of equality (semantic equality, not syntactic). Equality can however be defined inside a theory that is usually assumed to be part of all the other theories by default, unless stated otherwise. This theory contains a single symbol called the definedness symbol denoted by the ceiling operator and a single axiom stating $\lceil x \rceil$. How equality, subset and the other usual operations involving equality are derived from that is not discussed here but can be viewed in [2]. As a result of this fact, we freely use equality within our Matching Logic theories as well as equivalence with the added note that if they occur at the top of a pattern, equality and equivalence are logically interchangeable. The choice of which one we chose to use at the top of a pattern is stylistic and based on what we think would make the presentation easier to digest.

2.2 Nominal Logic

Nominal logic [9] is a sorted first-order theory, where some sorts α are distinguished 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 α for simplicity. We will present nominal logic as a matching logic theory, an approach referred to as NLML in prior work.

Abstractions in $[\alpha]\tau$ correspond to elements x of τ with a distinguished *bound* name a of sort α, typically written $[a]x$. 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 ($\#$) : $\alpha\times\tau \to Pred$ relating any name sort and any sort. Finally, nominal logic includes an extra quantifier, $\text{И}a.\phi$, which is pronounced "for fresh a, ϕ holds."

Pitts [9] gives a set of axioms and axiom schemas that describe the behavior of these constructs, and previous work [5] showed these can be adapted to matching logic as shown in Figure 2. The S axioms describe the behavior of swapping; note that by applying the swapping $(a\ a')$ to both sides of $S3$ and using S2 we can obtain $(a\ a')\cdot a' = a$. Axioms F and A characterize freshness and abstraction respectively. The E axiom ensures equivariance, that is, that swapping preserves the behavior of function and atomic predicate symbols. This axiom subsumes the four equivariance axioms that are needed when defining nominal logic as a first-order logic theory, since here σ can be a function or predicate symbol. However, E does not ensure that $R(\bar{x}) \Rightarrow R((a\ b)\cdot\bar{x})$, so we also include axiom P to ensure 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}). Finally axiom scheme Q characterizes the И-quantifier. Additionally, to ensure that function symbols are interpreted as (total, deterministic) functions in matching logic we also add instances of the function axioms:

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

where f and c are the function symbols and constants of the nominal logic signature. In particular, the built-in symbols $(- -)\cdot -$, $[-]-$, and $- \# -$ are assumed to be functional.

The abstraction construct can be used to specify binding operators, as shown in the example below for the λ-calculus.

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

$$\begin{array}{rl} var: & Var \to Exp \\ app: & Exp \times Exp \to Exp \\ lam: & [Var]Exp \to Exp \end{array}$$

A term of the form $\lambda x.e$ is represented as $lam([x]\mathbf{e})$ where $\mathbf{e}$ is the representation of e. Axiom $(A1)$ (see Figure 2) ensures the α-equivalence relation between λ-terms holds by construction. The capture-avoiding substitution operation $subst: Exp \times Var \times Exp \to Exp$ can be axiomatized as follows:

$$\begin{array}{rcll} subst(var(x), x, z) & = & z & (Subst1) \\ x \neq y \Rightarrow subst(var(x), y, z) & = & var(x) & (Subst2) \\ subst(app(x_1, x_2), y, z) & = & & \\ & & app(subst(x_1, y, z), subst(x_2, y, z)) & (Subst3) \\ a \# y, z \Rightarrow subst(lam([a]x), y, z) & = & & \\ & & lam([a]subst(x, y, z)) & (Subst4) \end{array}$$

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

$\mathcal{H}_\mu$	$\mathcal{H}$	(Proposition$_1$)	$\varphi_1 \to (\varphi_2 \to \varphi_1)$
		(Proposition$_2$)	$(\varphi_1 \to (\varphi_2 \to \varphi_3)) \to (\varphi_1 \to \varphi_2) \to (\varphi_1 \to \varphi_3)$
		(Proposition$_3$)	$(\neg\varphi_1 \to \neg\varphi_2) \to (\varphi_2 \to \varphi_1)$
		(Modus Ponens)	$\dfrac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2}$
		($\exists$-Quantifier)	$\varphi[y/x] \to \exists x.\varphi$
		($\exists$-Generalization)	$\dfrac{\varphi_1 \to \varphi_2}{(\exists x.\varphi_1) \to \varphi_2}$ if $x \notin FV(\varphi_2)$
		(Propagation$_\bot$)	$C_\sigma[\bot] \to \bot$
		(Propagation$_\vee$)	$C_\sigma[\varphi_1 \vee \varphi_2] \to C_\sigma[\varphi_1] \vee C_\sigma[\varphi_2]$
		(Propagation$_\exists$)	$C_\sigma[\exists x.\varphi] \to \exists x.C_\sigma[\varphi]$ if $x \notin FV(C_\sigma[\exists x.\varphi])$
		(Framing)	$\dfrac{\varphi_1 \to \varphi_2}{C_\sigma[\varphi_1] \to C_\sigma[\varphi_2]}$
		(Existence)	$\exists x.\, x$
		(Singleton Variable)	$\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])$ where C_1 and C_2 are nested symbol contexts.
		(Set Variable Substitution)	$\dfrac{\varphi}{\varphi[\psi/X]}$
		(Pre-Fixpoint)	$\varphi[\mu X.\varphi/X] \to \mu X.\varphi$
		(Knaster-Tarski)	$\dfrac{\varphi[\psi/X] \to \psi}{\mu X.\varphi \to \psi}$

Figure 1. Sound and complete proof system $\mathcal{H}$ of matching logic (above the double line) and the proof system $\mathcal{H}_\mu$ of Matching-Mu logic. Symbol contexts $C_\sigma = \sigma(\phi_1, \ldots, \Box, \ldots, \phi_n)$ are one-hole contexts where the hole is an argument to a symbol, and nested symbol contexts are compositions of symbol contexts.

$$\begin{array}{rcll}
(a\ a) \cdot x & = & x & (S1) \\
(a\ a') \cdot (a\ a') \cdot x & = & x & (S2) \\
(a\ a') \cdot a & = & a' & (S3) \\
(a\ b) \cdot \sigma(\bar{x}) & = & \sigma((a\ b) \cdot \bar{x}) & (E) \\
\forall x \colon Pred.\, x & = & \top_{Pred} & (P) \\
a \# x \wedge a' \# x & \Rightarrow & (a\ a') \cdot x = x & (F1) \\
a \# a' & \Leftrightarrow & a \neq a' & (F2) \\
\forall a : \alpha, a' : \alpha'.\ a \# a' & & (\alpha \neq \alpha') & (F3) \\
\forall \bar{x}.\exists a.\ a \# \bar{x} & & & (F4) \\
\forall \bar{x}. \quad (\text{И} a.\phi & \Leftrightarrow & \exists a.a \# \bar{x} \wedge \phi) & \\
 & & (FV(\text{И} a.\phi) \subseteq \bar{x}) & (Q) \\
[a]x = [a']x' & \Leftrightarrow & (a = a' \wedge x = x') & \\
 & & \vee (a \# x' \wedge (a\ a') \cdot x = x') & (A1) \\
\forall x : [\alpha]\tau.\ \exists a : \alpha, y : \tau.\ x & = & [a]y & (A2)
\end{array}$$

Figure 2. Matching Logic axiomatization of Nominal Logic

they can be proved from the above axioms and an induction principle for λ-terms (see e.g. [14]).

Alternative presentations of the axioms, using a concretion operator (which is partial function) and a fresh operator (which is multivalued), are also available, see [5] for details.

Nominal sets provide semantics to nominal logic (cf. [10]) and can be used to define matching logic models of nominal logic [5]. We recall the main definitions:

Let G be the symmetric group $Sym(\mathbb{A})$ of finite permutations 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 *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* is a G-set in which all elements have a finite support. This implies each element has a unique, least finite support which is denoted $supp(x) = \bigcap\{S \subseteq \mathbb{A} \mid \forall \pi.supp(\pi) \cap S = \emptyset \Rightarrow \pi \cdot x = x\}$.

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:

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

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.

Using the above constructions, the presentation of nominal logic in matching logic, given by the axioms in Figure 2 together with axioms Fun_c and Fun_f, is correct. More precisely, any closed nominal logic formula (i.e., any closed first-order formula ϕ using the signature of nominal logic) can be translated to a pattern $\phi' : Pred$ preserving validity (see [5] for details).

3 NLML: Nominal Logic as a Matching-Mu Logic Theory

NLML as presented in [5] is a specification of nominal logic as a theory in matching logic. Here we consider instead nominal logic as an instance of matching-mu logic. We will find it convenient to reformulate the axioms slightly, principally by replacing single-valued variables with multi-valued patterns when possible and by defining a *support pattern* $supp(\phi)$ that matches all atoms in the support of an element of ϕ, which we then use to define freshness.

There are several considerations we will address in turn:

1. Syntax: The only significant difference when moving to matching-mu logic is the addition of set variables and μ. We also include a *supp* symbol in the signature, which we use to define a *fresh-for* pattern, as well as a *comma* symbol that is used for specifying simultaneous freshness constraints (more on that later).
2. Axioms: We add axioms asserting that the built-in function symbols from nominal logic are functions, and generalise the axioms $S1, S2, F1, F4, A1$ to use multi-valued patterns.
3. Semantics: We need to confirm that the semantic behavior of the additional features is well-behaved from the point of view of nominal set models.

Once the above three issues are addressed, we can proceed to use NLML as an ordinary matching-mu logic theory, with confidence that the behavior of nominal constructs is consistent with the first-order treatment; more importantly, we can provide a solid foundation for definitions of languages with binding using μ, such as the lambda-calculus, directly instead of requiring additional ad-hoc axiom schemes for inversion and induction as in prior work.

3.1 Syntax

We consider matching-mu logic with the syntax specified in Section 2.2, which includes symbols for swapping, abstraction, etc. and include in the signature also the symbol $supp_\tau : \Sigma_{\tau;\alpha}$ which for each sort, maps elements of that sort to the names present in their supports. Note that the support can be empty or multivalued. We define *fresh_for* using *supp* as follows:

$$fresh\_for(\phi, \psi) = \neg(\phi \subseteq supp(\psi))$$

3.2 Axioms

We assume that the built-in symbols such as swapping and abstraction, and other symbols unless explicitly stated, behave as functions according to the axioms

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

As noted in previous work the axiom Q for the freshness quantifier of nominal logic can be used to remove all occurrences of the freshness quantifier from a formula or theory. This is helpful in our setting because we wish to work in a conventional matching logic setting without having to modify implementations to add additional quantifiers.

Thus, we assume from now on that if $\reflectbox{N}$ is used it is only as a shorthand for an instance of the right-hand side of axiom Q.

The axioms $S1, S2, F1, F4, A1$ are generalised to use multi-valued patterns whenever possible, as follows:

$$\begin{array}{rcll}
(a\ a) \cdot \phi & = & \phi & (S1') \\
(a\ a') \cdot (a\ a') \cdot \phi & = & \phi & (S2') \\
a \# \phi \wedge a' \# \phi & \Rightarrow & (a\ a') \cdot \phi = \phi & (F1') \\
\exists a.\ a \# \phi & & \text{for finitely valued } \phi & (F4') \\
[a]\phi = [a']\psi & \Leftrightarrow & (a = a' \wedge \phi = \psi) & \\
 & & \vee (a \# \psi \wedge (a\ a') \cdot \phi = \psi) & (A1')
\end{array}$$

In axiom $F4'$ we have to restrict ϕ to be finitely-valued to ensure there exists a fresh atom (otherwise ϕ could match the whole set of atoms, leading to an inconsistency). It is worth noting that this formulation is a little handwavey as there is no easy way to express in Matching Logic the fact that a pattern is matched by a finite number of values (as is the case in most logics, one needs the notion of bijections and natural numbers in order to express finiteness). To get around this issue, we actually require ϕ to be single-valued but we include a binary symbol that is axiomatized appropriately so as to "bundle together" multiple values into one, similar to a categorical product. We call this operator Comma and explore it in more detail in Section 4.

These axioms are equivalent to the original ones (Figure 2).

3.3 Models

As in previous work [5], sorts are interpreted as nominal sets and name sorts correspond to sets of atoms. The main differences from the previous specification of nominal logic in matching logic are the interpretation of set variables (in the power set of the domain) and the interpretation of μ.

We need to check that the interpretation of μ is indeed a nominal set. This follows from Proposition 7.14 in [10]: the least fixpoint operator preserves the support, assuming an equivariant, monotone function.

The correctness of the representation of nominal logic in matching logic using the axioms in Figure 2 was shown in [5]. The proofs are also valid in matching-mu logic (which subsumes matching logic).

Proposition 3.1 ([5]). *There exists a natural translation from closed nominal logic formulas to first-order 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.*

The translation is natural in the sense that each term in the formula is translated to its correspondent in NLML.

In prior work [5], a matching logic symbol $fresh_{\tau;\alpha}$ was considered instead which matches names fresh for values of some sort τ. Thus, for a variable x of sort τ denoting a single value, the pattern $\#\ (x)$ matches all names a of sort α that are fresh for x. The semantics of matching logic patterns then dictates that for a pattern ϕ, the pattern $fresh(\phi)$ will match all names that are fresh for *some* value matching ϕ; so for example, perhaps counterintuitively, we have $fresh(a \vee b) = \top_\alpha$, will match all names because every name is fresh for some other name. Here, we take a different approach by taking as primitive the *support* operation *supp*, which for a single-valued variable x matches names that are in the support of x. Then, $supp(\phi)$ matches names that are in the support of some value matching ϕ. This can again have counterintuitive consequences for patterns that match infinitely many values, but behaves a little better for patterns that match only finitely many values, in which case the meaning of the support pattern is exactly the union of the supports of the values. We can define the freshness relation straightforwardly in terms of *supp* as usual: $a\ \#\ x \equiv a \notin supp(x)$.

3.4 Induction Principle

In Matching Logic we can naturally derive induction principles for simple algebraic data types defined via the μ operator. More exactly, assume we want to formalize an ADT defined by a number of productions which may be inductive:

$$\begin{aligned} ADT = {} & constr_1(T^1_1, \dots T^1_{n_1}) \\ & \mid constr_2(T^2_1, \dots T^2_{n_2}) \\ & \dots \\ & \mid constr_m(T^m_1, \dots T^m_{n_m}) \end{aligned}$$

where any of the T's may be replaced by ADT.

In Matching Logic we would define the data type above using a μ pattern that matches all objects constructed via these productions (made up of a combination of the m constructors above):

$$\mu X.constr_1(T^{1\dagger}_1, \dots T^{1\ \dagger}_{n_1}) \vee \dots \vee constr_m(T^{m\dagger}_1, \dots T^{m\ \dagger}_{n_m})$$

where $ADT^\dagger = X$ and each $T^{a\dagger}_b = T^a_b$ if T^a_b is not ADT. For example, this is (part of) how natural numbers are axiomatized in [3]:

$$Nat = \mu X.zero \vee succ(X)$$

for previously defined sort Nat and constructors $zero : Nat$ and $succ : Nat \to Nat$. This axiom corresponds to the standard notion of "no junk".

If we just add a statement of this form to our theory, we can then immediately derive an induction principle for our ADT, in the form of a Matching Logic theorem that looks like this:

$$\begin{aligned} & (constr_1(T^{1\ddagger}_1, \dots T^{1\ \ddagger}_{n_1}) \Rightarrow \phi) \Rightarrow \\ & (constr_2(T^{2\ddagger}_1, \dots T^{2\ \ddagger}_{n_2}) \Rightarrow \phi) \Rightarrow \\ & \dots \\ & (constr_m(T^{m\ddagger}_1, \dots T^{m\ \ddagger}_{n_m}) \Rightarrow \phi) \Rightarrow \\ & (ADT \Rightarrow \phi) \end{aligned}$$

where $ADT^\ddagger = \phi$ and each $T^{a\ddagger}_b = T^a_b$ if T^a_b is not ADT.

In the case of natural numbers, the induction principle will look like:

$$(zero \Rightarrow \phi) \Rightarrow (succ(\phi) \Rightarrow \phi) \Rightarrow (Nat \Rightarrow \phi)$$

In layman's terms this is saying that if we can prove that a pattern is closed under the given productions, then the whole data type is contained within the pattern.

Similar techniques can be used to derive α-structural induction principles for languages with binding operators in NLML. For our formalisation of lambda calculus, we first add a standard "no junk" axiom:

$$Exp = \mu X.var(Var) \vee app(X, X) \vee lam([Var]X)$$

corresponding to the three productions of lambda calculus:

$$Exp = var(Var) \quad \mid \quad app(Exp, Exp) \quad \mid \quad lam([Var]Exp).$$

This axiom says that Exp is the least fixpoint of the construction that includes all variables built from variable names in α, and is closed under applications and lambda-abstractions of names in α.

Then, using the axiom above we can derive the following **standard induction principle**:

$$\begin{aligned} & (var(\top_{Var}) \Rightarrow \phi) \Rightarrow \\ & (app(\phi, \phi) \Rightarrow \phi) \Rightarrow \\ & (lam([\top_{Var}]\phi) \Rightarrow \phi) \Rightarrow \\ & (\top_{Exp} \Rightarrow \phi) \end{aligned}$$

In contrast with the approach taken in [5], which used the less expressive first-order matching logic, here we are able to axiomatize Exp directly using μ and then the induction principle is derivable.

Note that the induction principle as stated above is different from the one presented in prior work, which was schematic over a predicate $P \in \Sigma_{Exp,\tau_1,\dots,\tau_n;Pred}$ parameterised by an expression and some additional variables $\bar{y}$ representing context with respect to which bound variables need to be fresh:

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

This induction principle, which had to be taken as an axiom in [5], corresponds to the strong nominal induction principles provided by Nominal Isabelle [14, 15], specifically in the third case for lambda-abstractions where we are permitted to assume the bound name a is fresh for all of the variables $\bar{y}$.

We will now argue the case that the schematic induction principle above is not as useful when working in a Matching Logic context.

First note that the induction principle defined in previous work is expressed in terms of this arbitrary predicate P that can be applied to lambda terms. In Matching Logic, constructing predicate expressions parametric in element variables is awkward at best. One of the easiest ways to construct something resembling P is to introduce a new symbol for each predicate that we are interested in and to axiomatize the behavior of this symbol when applied to an element variable of the appropriate type. Instead of doing this however, we can opt for a much more idiomatic way to express predicates in Matching Logic, and that is by considering the predicate on some sort S to be represented by a pattern ϕ with $\phi \subseteq S$. The idea is that any element in S that satisfies the predicate will be matched by ϕ and any element matched by ϕ will satisfy the predicate. So ϕ here is thought of as the pattern matching all elements that satisfy the predicate[1]. If we try to restate the former induction principle under this interpretation, we note that we cannot easily integrate the freshness constraint $\bar{y}$ into the predicate. The closest variant we can get is the following:

$$\begin{aligned}(\forall a, b : Var.\ (ab)\phi = \phi) \Rightarrow\\ (\forall a : Var.\ var(a) \in \phi) \Rightarrow\\ (\forall t_1, t_2 : Exp.\ t_1 \in \phi \land t_2 \in \phi \Rightarrow app(t_1, t_2) \in \phi) \Rightarrow\\ (\forall a : Var, t : Exp.\ a \# \rho \Rightarrow (t \in \phi) \Rightarrow lam([a]t) \in \phi) \Rightarrow\\ \forall t : Exp.\ t \in \phi\end{aligned}$$

where ρ represents a pattern (that may be chosen by the user of the theorem) matching the variables and terms that we want to be fresh for.

[1]One thing to note here is that we use the word "predicate" to refer to first-order predicates, parametric in one or more variables. Matching Logic also has the notion of a "predicate" although that is a different concept - it refers to patterns that are either matched by everything or nothing. This is not what we are referring to.

The keen eyed reader may notice that this formulation is almost equivalent to the standard induction principle stated earlier. The only differences are the addition of the first antecedent requiring ϕ to be equivariant (closed under swaps) and the $a \# \rho$ constraint in the fourth antecedent. This is indeed the first iteration of the induction principle that we proved ("induction_principle" in the formalization). Due to this difference however, the proof was more difficult than the one for the standard induction principle. On top of that, we later realised that for our purposes, this formulation was not any stronger than the standard induction principle. To see exactly why this "baked-in" freshness guarantee was not useful for the user of the principle, consider what ϕ would normally look like when instantiated. For the substitution lemma (which we will discuss shortly), the resulting ϕ looks like this:

$$\begin{aligned}satisfying\_exps \triangleq \exists x : Exp.x \land\\ \forall a, b : Var.\forall p_1, p_2 : Exp.a \# p_2, b \Rightarrow\\ x[p_1/a][p_2/b] = x[p_2/b][p_1[p_2/b]/a]\end{aligned}$$

Notice how the variables that we need a to be fresh for are bound inside the definition of *satisfying_exps*. Thus we have no way to construct a meaningful ρ for the induction principle.

After realising this shortcoming of this induction principle, we decided to instead change our approach and use the standard induction principle for μ ("simple_induction_principle" in the formalization).

In the rest of this section we show how to use the techniques described above to prove properties of the lambda-calculus. Our case study for demonstrating the power and effectiveness of our approach is the derivation of the well-known substitution lemma in lambda calculus, namely:

$$p[p_1/a][p_2/b] = p[p_2/b][p_1[p_2/b]/a] \text{ if } a \# p_2, b$$

To this end, we had to perform a proof by induction on p by using the induction principle. More precisely we had to prove that $\top_{Exp}$ = *satisfying_exps* which is where the induction principle comes into play. While, by using the standard induction principle, we no longer need to provide the proof of equivariance of *satisfying_exps* directly, we still need this in order to prove the third antecedent of the standard induction principle:

$$(lam([\top_{Var}]satisfying\_exps)) \subseteq satisfying\_exps$$

On the high level, the proof consisted of choosing an arbitrary term matching the LHS of the subset relation, say $[a]p$ such that $a : Var$ and $p \in satisfying\_exps$. Using axiom F4 of Nominal Logic, we can find a variable b that is fresh for every term and variable seen so far. Using axiom A1, we can show that $[a]p = [b]((ab)p)$. Using the equivariance of *satisfying_exps*, we were able to show that $p \in satisfying\_exps \Rightarrow (bc)p \in satisfying\_exps$. Therefore, our original term $[a]p$ equals some other term $[b]p'$ for some

$p' \in$ *satisfying_exps*. Using the fact that b is chosen to be fresh for everything, the rest of the proof is straightforward.

The other two antecedents regarding the *var* and *app* symbols are straightforward, using the axioms of lambda substitution (Subst1)–(Subst4) (whose formalizations are shown in detail in the next section).

Once we successfully proved all three antecedents of the standard induction principle, and knowing also that *satisfying_exps* $\subseteq \top_{Exp}$ from its definition, we were able to use the principle to prove that $\top_{Exp} =$ *satisfying_exps*. Using this fact, the rest of the proof is routine.

4 Mechanization

The mechanization part of the project consists of formalizing the theory presented above in a proof assistant. For this we have used Metamath Zero[2] (see [1]). The entire code of our formalisation (including the base library, but excluding Metamath Zero itself) can be found in the supplementary material part of the submission, as a zip file. We recommend that the reader follows along while looking at these files if they are interested in all the technical details of our work. But for convenience we also added most of the code (minus the proofs) to appendix (A).

4.1 Metamath Zero

Metamath Zero is a new proof assistant, inspired from Metamath [7] and designed with the same philosophy of stripping down the trust base to the minimum necessary. The framework includes a custom binary format (mmb) for encoding proofs and a text based format for encoding specifications that the proofs are checked against (mm0). It also includes a number of tools designed to aid in proof engineering, such as a higher level proof language (mm1) that compiles down to a pair of spec (mm0) and proof (mmb) files as well as a VS Code extension.

We chose to use Metamath Zero for the following reasons: We had access to begin with to an extensive MM0 Matching Logic library [12] as well as to expertise regarding its design. Additionally we like MM0 for its simplicity and its minimal trust base (quite possibly the smallest in terms of lines of code compared to the standard distributions of all other mainstream proof assistants, including Metamath) as well as its incredible performance when checking proofs.

Our work builds upon the aforementioned library by introducing a new Matching Logic theory formalizing Nominal Logic. Patterns in this theory correspond to sets of finitely supported elements of Nominal Sets as per [10].

We then used this theory to formalize untyped lambda calculus and to derive an induction principle for lambda terms, as described in the earlier section. Finally, we used this induction principle to prove a standard result about substitutions in lambda calculus.

[2] https://github.com/digama0/mm0

4.2 Applicative Matching Logic

One important aspect to note about the Matching Logic library is that it is based on Applicative Matching Logic [4], a variant of Matching Logic without sorts or function signatures. The reason why we chose this variant is first because it cuts down on complexity, reducing the logic to its bare minimum, and second because it makes the formalization more flexible as it is not affected by Metamath Zero's limit on the number of definable sorts. Furthermore, as shown in earlier work on this subject, sorts and function signatures can be defined at the object level on top of Applicative Matching Logic (abbreviated further on as AML) as a theory, so we are also reducing the trust base.

Below is the definition and grammar of AML:

Definition 4.1. An AML *signature* (EVar, SVar, Σ) has a set EVar of *element variables* $x, y, \ldots$, a set SVar of *set variables* $X, Y, \ldots$, and a set Σ of *(constant) symbols* $\sigma, f, g, \ldots$. We often omit EVar and SVar and use Σ to denote the signature. The set of AML *patterns*, denoted Pattern, is defined as:

$$\varphi ::= x \mid X \mid \sigma \mid \varphi_1\varphi_2 \mid \bot \mid \varphi_1 \Rightarrow \varphi_2 \mid \exists x\,.\,\varphi \mid \mu X\,.\,\varphi$$

where in $\mu X\,.\,\varphi$ we require that φ is positive in X, i.e., X is not nested an odd number of times on the left of an implication $\varphi_1 \Rightarrow \varphi_2$. Pattern $\varphi_1\varphi_2$ is called *application* and assumed associative to the left. The scope of binders $\exists$ and μ goes farthest to the right. The notions of free variables $FV(\varphi)$, α-renaming, and capture-avoiding substitution ($\varphi[\psi/x]$ and $\varphi[\psi/X]$) are defined in the usual way.

We briefly outline the way in which we axiomatize sorts. Each sort that we want to consider will have an associated sort symbol in the signature which we axiomatize to be functional (matched by a singleton). Additionally, all signatures will have a special symbol denoted *dom* that represents the domain of a given sort when applied to its symbol (e.g. for an appropriate axiomatization of integers, *dom Int* will match all integers).

Once we have this, it is straightforward to axiomatize a symbol's signature. Take, for example, a symbol $f : A \longrightarrow B$. This will correspond to the following axiom:

$$\forall x. x \in dom\ A \Rightarrow f\ x \in dom\ B$$

The case when f has a longer signature is analogous.

4.3 Formalization

We start by explaining some of the particularities of the Metamath Zero syntax. Each axiom or theorem has a name followed by a list of metavariables that it is parameterised by (see the examples later in this section and also the Appendix). The metavariables in curly braces can only be instantiated by distinct variable names while the ones in parentheses can be instantiated by any pattern. Sometimes we may also have a list of variables after the sort definition, like here: (phi : $Pattern\ a\ b\ c\ d$). This simply means that the given names are allowed to occur in the instantiation of *phi*, while if they are absent, then *phi* may not contain those. We aim to have as few such restrictions as possible so we usually allow all mentioned variables in all patterns, except for Matching Logic sorts.

Following the declaration of metavariables we have a possibly empty list of hypotheses (which in the case of theorems have names also) followed by a conclusion. Each hypothesis and the conclusion is wrapped in dollar signs which signify math mode. When using an axiom or theorem in a proof, all hypothesises must be proved in order to obtain a proof of the conclusion.

For example note how we formalise axiom (S3):

```
axiom S3 {a b: EVar} (alpha: Pattern a b):
  $ is_atom_sort alpha $ >
  $ s_forall alpha a (
    s_forall alpha b (
    ((swap (eVar a) (eVar b) (eVar a)) == (eVar b)))) $;
```

The notation *is_atom_sort* simply requires that the provided pattern is indeed an atom sort, that is, that it is part of the sort of all atom sorts. The notation *is_nominal_sort* is analogous.

The predicate *is_sorted_func* is used to represent the fact that the given pattern is of a given sort and also functional (matched by a single element). More precisely,

$$is\_sorted\_func(S, \phi) \triangleq \exists x : S.x = \phi \ .$$

Note that while the statements and definitions that we use may seem verbose, those can be made more succinct by using mm1 macros or by adjusting the binding power of operators to reduce use of brackets, but this is outside the scope of our current work.

Before presenting the embedding of our axioms and theorems, we will go over some of the noteworthy design decisions that we had to make along the way:

Equivariance axioms. The Nominal Logic axiom scheme E states that every symbol is equivariant. While we did start off by attempting to state a universal axiom about all possible symbols, we found that to be potentially dangerous and prone to misunderstandings. There are a number of reasons why we don’t want to be so restrictive. Specifically, we chose to design our library so that it may soundly interoperate with other Matching Logic theories. We also wanted to allow one to add symbols to our theory that are needed for technical reasons but that may not be necessarily equivariant, like a symbol that can check membership of a an atom inside a given set. As a result of these considerations we decided to drop axiom E and to instead construct specific equivariance axioms for each individual symbol that we add. This resulted in axioms such as the following:

```
axiom EV_abstraction {a b: EVar} (alpha tau: Pattern)
  (phi psi: Pattern a b):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ s_forall alpha a (s_forall alpha b (
    (is_of_sort phi alpha) ->
    (is_of_sort psi tau) ->
    ((swap (eVar a) (eVar b) (abstraction phi psi)) ==
    abstraction
      (swap (eVar a) (eVar b) phi)
      (swap (eVar a) (eVar b) psi)))) $;

axiom EV_swap {a b c d: EVar} (alpha tau: Pattern)
  (phi: Pattern a b c d):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ s_forall alpha a (s_forall alpha b (
    s_forall alpha c (s_forall alpha d (
    (is_of_sort phi alpha) ->
    (is_of_sort psi tau) ->
    ((swap (eVar a) (eVar b)
      (swap (eVar c) (eVar d) phi)) ==
    swap
      (swap (eVar a) (eVar b) (eVar c))
      (swap (eVar a) (eVar b) (eVar d))
      (swap (eVar a) (eVar b) phi)))))) $;

axiom EV_supp {a b: EVar} (alpha tau: Pattern)
  (phi: Pattern a b):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ s_forall alpha a (s_forall alpha b (
    (is_of_sort phi tau) ->
    ((swap (eVar a) (eVar b) (supp phi)) ==
    supp (swap (eVar a) (eVar b) phi)))) $;
```

Multi-valued ML patterns. While Nominal Logic was initially designed as a theory in First-Order Logic, we found it interesting and more idiomatic to leverage the expressiveness of Matching Logic when restating the Nominal Logic axioms. As such we decided to express the axioms in terms of Matching Logic pattern metavariables, which may be multi-valued in the interpretation. This is not due to a limitation in Matching Logic or in Metamath Zero. We could have instead been more faithful to the original axiomatization and used universally quantified element variables (which are single valued) instead of the pattern metavariables but this wouldn’t have been as idiomatic for Matching Logic. Instead we allowed some of the variables to now be multi-valued, making sure that this is a conservative change. This is more in-line with what makes Matching Logic shine in the first place, working by default with sets of objects, all matched by a single pattern. This change feels very natural and makes the statements look a lot closer to their formulation in standard Nominal Logic. We see the fact that we can do this with no compromises as one of the contributions of our work.

Support. As mentioned in earlier sections, we decided to not use a primitive notion of freshness, but instead to define freshness in terms of the support of a pattern, represented by the symbol *supp*. This is because *supp* behaves very nicely as a matching logic symbol, which is in turn because of the point-wise semantics of symbol application in Matching Logic. If we defined $fresh$ as a primitive symbol giving us all the fresh atoms of a term, then $fresh$ applied to a multi-valued pattern would have an unexpected and awkward meaning: $fresh(phi)$ would *not* match all the atoms fresh for all terms that match *phi*, instead it would match all the atoms fresh for at least one term matching *phi*. In particular, assuming a and b are disjoint, $fresh(a \vee b)$ would match all atoms since any atom is fresh for either a or b. If we however instead consider *supp* as the primitive symbol, we note that it does behave as expected: $supp(phi \vee psi)$ will match all the atoms in the union of the supports of *phi* and *psi*, and in particular $supp(a \vee b) = a \vee b$. This allows us to define the *fresh_for* predicate as follows: $fresh\_for(phi, psi) \triangleq phi \not\subseteq supp(psi)$. This has the added advantage that it allows us to use a multi-valued pattern for the atoms in question and it still retains the semantics that we expect: that all the atoms matching *phi* are fresh for all the terms matching *psi*.

Axiom F4 and the Comma operator. Stating that a pattern has a finite interpretation is surprisingly difficult in Matching Logic. We can use certain tricks to state that a pattern is matched by exactly N elements for a given N but we can't easily state in Matching Logic itself that a pattern is matched by N elements for some unknown natural number N. This forces us to be very careful when stating axiom F4, which allows us to find a fresh atom for any finite list of terms. If we naively expressed axiom F4 as $\exists x.fresh\_for(x, phi)$ for some metavariable *phi* then nothing would be stopping us from instantiating *phi* by the sort of all atoms $\top_\alpha$, which would obviously be unsound. We therefore choose to restrict our formalization of axiom F4 to require that *phi* is matched by a single element:

```
axiom F4 {a: EVar} (alpha tau phi: Pattern):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ (is_sorted_func (dom tau) phi) ->
    (s_exists alpha a (fresh_for (eVar a) phi)) $;
```

While this solves the unsoundness issue, it is too restrictive for our needs. To counter that, we introduce a new symbol: *comma*, also written , , . The purpose of this symbol is to bundle together multiple terms into a single one, such that we may use it in axiom F4:

```
term comma_sym: Symbol;
def comma (phi psi: Pattern): Pattern =
  $ (sym comma_sym) @@ phi @@ psi $;
infixl comma: $,,$ prec 35;
term comma_sort_sym: Symbol;
def comma_sort: Pattern = $ sym comma_sort_sym $;

axiom comma_sort_nominal: $ is_nominal_sort comma_sort $;
axiom fresh_comma {a: EVar} (alpha tau1 tau2: Pattern)
  (phi1 phi2: Pattern a):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau1 $ >
  $ is_nominal_sort tau2 $ >
  $ s_forall alpha a (
    (is_of_sort phi1 tau1) ->
    (is_of_sort phi2 tau2) ->
    (fresh_for (eVar a) (phi1 ,, phi2)) ->
    (fresh_for (eVar a) phi1) /\
    (fresh_for (eVar a) phi2)) $;
```

As we can see, the *fresh_comma* axiom allows us to unpack these bundled terms. We also note that since all terms must have finite support, combining two terms together in this manner also gives us something with finite support, so this is not breaking soundness.

Functions. Since we are working in an Applicative Matching Logic based system (due to its minimality), sorts and functions must be defined at the object level. What this means is that, besides the equivariance axioms, we must also add for each symbol an axiom stating that it is a function with a certain signature. What each of these axioms state is that if the symbol in question is applied to singleton patterns of the correct sort, the result will also be a singleton of the output sort. Below are a few examples of these axioms:

```
axiom function_swap:
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ ,(is_function '(sym swap_sym) '[alpha alpha tau]
    'tau) $;
axiom function_swap_atom:
  $ is_atom_sort alpha $ >
  $ ,(is_function '(sym swap_sym) '[alpha alpha alpha]
    'alpha) $;
axiom function_abstraction:
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ ,(is_function '(sym abstraction_sym) '[alpha tau]
    '(sort_abstraction alpha tau)) $;
axiom multifunction_supp:
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ ,(is_multi_function '(sym supp_sym) '[tau]
    'alpha) $;
```

Nominal Logic. We are now ready to present an overview of the structure of the main file defining Nominal Logic:

```
term atoms_sym: Symbol;
def atoms: Pattern = $ sym atoms_sym $;
axiom sort_atoms: $ is_sort atoms $;

term nominal_sorts_sym: Symbol;
def nominal_sorts: Pattern = $ sym nominal_sorts_sym $;
axiom sort_nominal_sorts: $ is_sort nominal_sorts $;

axiom atoms_nominal_sorts: $ dom atoms C=
dom nominal_sorts $;

def is_atom_sort (alpha: Pattern): Pattern =
  $ (is_func alpha) /\ (alpha C= dom atoms) $;
def is_nominal_sort (tau: Pattern): Pattern =
  $ (is_func tau) /\ (tau C= dom nominal_sorts) $;
def is_atom (a alpha: Pattern): Pattern =
  $ is_sorted_func (dom alpha) a $;
```

Most of the above is boilerplate code required to set up the infrastructure for working with sorts as well to state that atom sorts are also nominal sorts and to introduce some useful definitions that we briefly discussed in earlier sections.

Besides this and the symbol axioms discussed above, we only have the Nominal Logic axiomatization that we are now ready to present:

```
axiom S1 (alpha tau a phi: Pattern):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ (is_atom a alpha) ->
    (is_of_sort phi tau) ->
    ((swap a a phi) == phi) $;
axiom S2 {a b: EVar} (alpha tau phi: Pattern a b):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ s_forall alpha a (
    s_forall alpha b (
    (is_of_sort phi tau) ->
    ((swap (eVar a) (eVar b)
      (swap (eVar a) (eVar b) phi)) == phi))) $;
axiom S3 {a b: EVar} (alpha: Pattern a b):
  $ is_atom_sort alpha $ >
  $ s_forall alpha a (
    s_forall alpha b (
    ((swap (eVar a) (eVar b) (eVar a)) == (eVar b)))) $;
axiom F1 {a b: EVar} (alpha tau phi: Pattern a b):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ s_forall alpha a (
    s_forall alpha b (
    (is_of_sort phi tau) ->
    ((fresh_for (eVar a) phi) /\
    (fresh_for (eVar b) phi) ->
    ((swap (eVar a) (eVar b) phi) == phi)))) $;
axiom F2 {a b: EVar} (alpha: Pattern a b):
  $ is_atom_sort alpha $ >
  $ s_forall alpha a (
    s_forall alpha b (
    (((eVar a) != (eVar b)) <->
    (fresh_for (eVar a) (eVar b))))) $;
axiom F3 {a b: EVar} (alpha1 alpha2: Pattern a b):
  $ is_atom_sort alpha1 $ >
  $ is_atom_sort alpha2 $ >
  $ s_forall alpha a (
    s_forall alpha b (
    (alpha1 != alpha2) -> ((eVar a) != (eVar b)))) $;
axiom F4 {a: EVar} (alpha tau phi: Pattern):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ (is_sorted_func (dom tau) phi) ->
    (s_exists alpha a (fresh_for (eVar a) phi)) $;
axiom A1 {a b: EVar} (alpha tau phi rho: Pattern a b):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ s_forall alpha a (
    s_forall alpha b (
    (is_of_sort phi tau) ->
    (is_of_sort rho tau) ->
    (((abstraction (eVar a) phi) ==
    (abstraction (eVar b) rho)) <->
    (((eVar a) == (eVar b)) /\ (phi == rho)) \/
    ((fresh_for (eVar a) rho) /\
    ((swap (eVar a) (eVar b) phi) == rho))))) $;
axiom A2 (alpha tau: Pattern) {x a y: EVar}:
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ s_forall (sort_abstraction alpha tau) x
    (s_exists alpha a (s_exists tau y
    (eVar x == abstraction (eVar a) (eVar y)))) $;
```

As can be noted, the statements look very similar to the ones presented in Section 2.

In what follows we will also present the axiomatization of lambda calculus as well as the statements of our two main theorems: the induction principle and the substitution theorem.

```
term Var_sym: Symbol;
def Var: Pattern = $ sym Var_sym $;
def Vars: Pattern = $ dom Var $;
def is_var (p: Pattern): Pattern =
  $ is_of_sort p Var $;
```

```
axiom Var_atom: $ is_atom_sort Var $;

term Exp_sym: Symbol;
def Exp: Pattern = $ sym Exp_sym $;
def Exps: Pattern = $ dom Exp $;
def is_exp (p: Pattern): Pattern =
  $ is_of_sort p Exp $;
axiom Exp_sort: $ is_nominal_sort Exp $;
```

Here we introduce the two sorts that we will be working with, Var and Exp, and we axiomatize that they are atom and nominal sorts respectively.

We then introduce the four symbols that we will be working with, three constructors and one function:

$$\begin{aligned} var &: Var \to Exp \\ app &: Exp \times Exp \to Exp \\ lam &: sort\_abstraction(Var, Exp) \to Exp \\ subst &: Var \times Exp \times Exp \to Exp \end{aligned}$$

Each of which comes with a function axiom and an equivariance axiom, as described above. Here is what these definitions and axioms look like for *lam* (we skip the rest for brevity):

```
term lc_lam_sym: Symbol;
def lc_lam (phi: Pattern): Pattern =
  $ (sym lc_lam_sym) @@ phi $;
def lc_lam_a (a p: Pattern): Pattern =
  $ lc_lam (abstraction a p) $;
axiom function_lc_lam:
  $ ,(is_function '(sym lc_lam_sym)
    '[(sort_abstraction Var Exp)] 'Exp) $;
axiom EV_lc_lam {a b: EVar} (phi: Pattern a b):
  $ s_forall Var a (s_forall Var b (
    (is_of_sort phi (sort_abstraction Var Exp)) ->
    ((swap (eVar a) (eVar b) (lc_lam phi)) ==
    lc_lam (swap (eVar a) (eVar b) phi)))) $;
```

We then have the standard no junk axiom, as described earlier:

```
axiom no_junk {X: SVar}:
  $ Exps == mu X ( (lc_var Vars)
              \/ (lc_app (sVar X) (sVar X))
              \/ (lc_lam (abstraction Vars (sVar X))))
  $;
```

Below is the standard induction principle. It is slightly modified from the one presented earlier in order to better fit our proofs but can be very straightforwardly derived from it.

```
theorem induction_principle (exp_pred: Pattern):
  $ (is_exp exp_pred) ->
    ((lc_var Vars) C= exp_pred) ->
    ((lc_app exp_pred exp_pred) C= exp_pred) ->
    ((lc_lam (abstraction Vars exp_pred))
      C= exp_pred) ->
    (Exps == exp_pred) $
```

Following this, we introduce the axiomatization of the substitution operation:

```
axiom subst_fresh {a: EVar} (phi plug: Pattern a):
  $ s_forall Var a (
    (is_exp phi) ->
    (is_exp plug) ->
    (fresh_for (eVar a) phi) ->
    ((subst (eVar a) phi plug) == phi)) $;
axiom subst_same_var {a: EVar} (plug: Pattern a):
  $ s_forall Var a (
    (is_exp plug) ->
    ((subst (eVar a) (lc_var (eVar a)) plug) ==
      plug)) $;
axiom subst_diff_var {a b: EVar} (plug: Pattern a b):
  $ s_forall Var a (
    s_forall Var b (
    ((eVar a) != (eVar b)) ->
    (is_exp plug) ->
    ((subst (eVar b) (lc_var (eVar a)) plug) ==
      (lc_var (eVar a))))) $;
axiom subst_var {a b: EVar} (plug: Pattern a b):
  $ s_forall Var a (
    s_forall Var b (
    ((eVar a) == (eVar b)) ->
    (is_exp plug) ->
    ((subst (eVar b) (lc_var (eVar a)) plug) ==
      plug))) $;
axiom subst_app {a: EVar} (phi1 phi2 plug: Pattern a):
  $ s_forall Var a (
    (is_exp plug) ->
    (is_exp phi1) ->
    (is_exp phi2) ->
    ((subst (eVar a) (lc_app phi1 phi2) plug) ==
      (lc_app
        (subst (eVar a) phi1 plug)
        (subst (eVar a) phi2 plug)))) $;
axiom subst_lam {a b: EVar} (plug phi: Pattern a b):
  $ s_forall Var a (
    s_forall Var b (
    ((eVar a) != (eVar b)) ->
    (fresh_for (eVar a) plug) ->
    (is_exp plug) ->
    (is_exp phi) ->
    ((subst (eVar b) (lc_lam (abstraction
      (eVar a)
      phi)) plug) ==
      (lc_lam (abstraction
        (eVar a)
        (subst (eVar b) phi plug)))))) $;
```

As we can see, we do not need to have an axiom for the case when we substitute for a in $\lambda a. \ldots$ and this is because we can always alpha rename the bound variable to one different from a. And indeed these are all the axioms we need to prove the substitution theorem:

```
theorem subst_induction (a b phi plug1 plug2: Pattern)
  (diff_atoms_ab: $ a != b $)
  (a_var: $ is_sorted_func Vars a $)
  (b_var: $ is_sorted_func Vars b $)
  (phi_exp: $ is_exp phi $)
  (plug1_exp: $ is_exp plug1 $)
  (plug2_exp: $ is_sorted_func Exps plug2 $)
  (a_fresh: $ fresh_for a plug2 $):
  $ (subst b (subst a phi plug1) plug2) ==
    (subst a (subst b phi plug2)
             (subst b plug1 plug2)) $
```

The proof goes as described in the previous section.

5 Related Work

The closest related work is [5], where nominal logic is axiomatized as a theory in matching logic without mu. As explained in Section 3, thanks to the use of μ here we are able to specify the "no junk" axioms that are needed to derive induction principles, so we could state the induction principle for the lambda calculus as a theorem, whereas in previous work it had to be taken as an axiom.

In terms of the derivation of induction principles in Matching Logic, the most recent and relevant works are [3] and [12]. In this paper we actually make the observation that the techniques used in both of these works can indeed be generalised to arbitrary algebraic data types, provided that they are not coinductive and have at least one non-inductive production.

The possibility to provide so-called strong induction principles that build-in Barendregt-style conventions regarding renaming and freshness of bound names was seen from an early stage as an important advantage of nominal techniques [6, 10]. Urban et al. [15] formalised this connection in the context of Nominal Isabelle, an extension of Isabelle/HOL to include support for nominal techniques including atoms, nominal datatypes, defining functions and inductive rules over nominal datatypes, reasoning about equivariance, and strong induction principles. In Nominal Isabelle, nominal sets are internal constructions; rather than modifying the foundational theory, nominal sets and related concepts are defined in terms of ordinary higher-order logic, with additional proof automation and keywords supported by the Isabelle infrastructure. This has the advantage of ensuring compatibility with Isabelle/HOL's large collection of libraries, but means that certain expected properties of nominal sets and datatypes do not come for free and have to be proved anew for each new type, recursive function or inductive definition. The strong induction principles, while providing convenient freshness constraints in cases involving binders, can require some extra effort to discharge proof obligations and use correctly. In contrast, our approach does not bake freshness assumptions into the induction principle, and just uses the standard fixpoint induction rule, but then some care is required to reason about cases involving binding.

6 Conclusions

We proposed a specification of nominal logic as a theory in matching-mu logic, and showed how it can be used to define algebraic data types that include binding operators and to reason about their properties. Taking advantage of μ patterns and the abstraction construct from nominal logic, we can specify "no junk" axioms that enable the automatic derivation of induction principles that take into account the α-equality relation generated by the binders. In contrast, induction principles are not derivable in prior specifications of nominal logic in matching logic without μ. We showed how to use these principles to build proofs by induction taking advantage of the expressive power of matching logic, that allows us to model properties as patterns. To illustrate the techniques we defined the type of lambda-terms and derived an induction principle, which we used to prove the substitution lemma of the lambda-calculus.

The techniques presented in this paper have been formalized in Metamath Zero, using the existing Matching Logic library, which we extended with axioms to specify nominal logic and the lambda-calculus. Although the simplicity and minimal trust base of Metamath make it an excellent choice for this kind of formalization, the techniques could be adapted to be used in other proof assistants provided a matching logic library is available.

A Code

In this section we highlight some relevant snippets of the code of the formalization, written in Metamath Zero.

The formalization builds on top of a former Matching Logic MM0 library. We will omit details related to this core library, choosing instead to focus here on the new contributions. Below is the complete formalisation of Nominal Logic, including the usual axioms of Nominal Logic translated in our formalization as Matching Logic statements, as discussed in Sections 3 and 4:

```
term atoms_sym: Symbol;
def atoms: Pattern = $ sym atoms_sym $;
axiom sort_atoms: $ is_sort atoms $;

term nominal_sorts_sym: Symbol;
```

```
def nominal_sorts: Pattern =
  $ sym nominal_sorts_sym $;
axiom sort_nominal_sorts:
  $ is_sort nominal_sorts $;

axiom atoms_nominal_sorts:
  $ dom atoms C= dom nominal_sorts $;

term sort_abstraction_sym: Symbol;
def sort_abstraction (abs body: Pattern): Pattern =
  $ (sym sort_abstraction_sym) @@ abs @@ body $;
axiom function_sort_abstraction:
  $ ,(is_function '(sym sort_abstraction_sym)
    '[atoms nominal_sorts] 'nominal_sorts) $;

def is_atom_sort (alpha: Pattern): Pattern =
  $ (is_func alpha) /\ (alpha C= dom atoms) $;
def is_nominal_sort (tau: Pattern): Pattern =
  $ (is_func tau) /\ (tau C= dom nominal_sorts) $;
def is_atom (a alpha: Pattern): Pattern =
  $ is_sorted_func (dom alpha) a $;

axiom nominal_sorts_are_sorts:
  $ (is_nominal_sort phi) -> (is_sort phi) $;

term swap_sym: Symbol;
def swap (a b phi: Pattern): Pattern =
  $ (sym swap_sym) @@ a @@ b @@ phi $;
term abstraction_sym: Symbol;
def abstraction (phi rho: Pattern): Pattern =
  $ (sym abstraction_sym) @@ phi @@ rho $;
term supp_sym: Symbol;
def supp (phi: Pattern): Pattern =
  $ (sym supp_sym) @@ phi $;
def fresh_for (phi psi: Pattern): Pattern =
  $ ~ (phi C= supp psi) $;
term comma_sym: Symbol;
def comma (phi psi: Pattern): Pattern =
  $ (sym comma_sym) @@ phi @@ psi $;
infixl comma: $,$ prec 35;
term comma_sort_sym: Symbol;
def comma_sort: Pattern =
  $ sym comma_sort_sym $;

axiom comma_sort_nominal:
  $ is_nominal_sort comma_sort $;

def EV_pattern {.a .b: EVar}
  (alpha phi: Pattern): Pattern =
$ s_forall alpha a (s_forall alpha b (
  (swap (eVar a) (eVar b) phi) == phi)) $;

axiom function_swap:
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ ,(is_function '(sym swap_sym)
    '[alpha alpha tau] 'tau) $;
axiom function_swap_atom:
  $ is_atom_sort alpha $ >
  $ ,(is_function '(sym swap_sym)
    '[alpha alpha alpha] 'alpha) $;
axiom function_abstraction:
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ ,(is_function '(sym abstraction_sym)
    '[alpha tau] '(sort_abstraction alpha tau)) $;
axiom multifunction_supp:
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ ,(is_multi_function '(sym supp_sym)
    '[tau] 'alpha) $;
axiom function_comma:
  $ is_nominal_sort tau1 $ >
  $ is_nominal_sort tau2 $ >
  $ ,(is_function '(sym comma_sym)
    '[tau1 tau2] 'comma_sort) $;

axiom EV_abstraction {a b: EVar}
  (alpha tau: Pattern) (phi psi: Pattern a b):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ s_forall alpha a (s_forall alpha b (
    (is_of_sort phi alpha) ->
    (is_of_sort psi tau) ->
    ((swap (eVar a) (eVar b)
      (abstraction phi psi)) ==
      abstraction
        (swap (eVar a) (eVar b) phi)
        (swap (eVar a) (eVar b) psi)))) $;

axiom EV_swap {a b c d: EVar} (alpha tau: Pattern)
  (phi: Pattern a b c d):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ s_forall alpha a (s_forall alpha b (
    s_forall alpha c (s_forall alpha d (
    (is_of_sort phi alpha) ->
    (is_of_sort psi tau) ->
    ((swap (eVar a) (eVar b)
      (swap (eVar c) (eVar d) phi)) ==
      swap
        (swap (eVar a) (eVar b) (eVar c))
        (swap (eVar a) (eVar b) (eVar d))
        (swap (eVar a) (eVar b) phi)))))) $;

axiom EV_supp {a b: EVar} (alpha tau: Pattern)
```

```
  (phi: Pattern a b):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ s_forall alpha a (s_forall alpha b (
    (is_of_sort phi tau) ->
    ((swap (eVar a) (eVar b) (supp phi)) ==
    supp (swap (eVar a) (eVar b) phi)))) $;

axiom EV_comma {a b: EVar} (alpha tau: Pattern)
  (phi1 phi2: Pattern a b):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ s_forall alpha a (s_forall alpha b (
    (is_of_sort phi1 tau) ->
    (is_of_sort phi2 tau) ->
    ((swap (eVar a) (eVar b) (phi1 ,, phi2)) ==
    ((swap (eVar a) (eVar b) phi1) ,,
     (swap (eVar a) (eVar b) phi2))))) $;

axiom fresh_comma {a: EVar} (alpha tau1 tau2: Pattern)
  (phi1 phi2: Pattern a):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau1 $ >
  $ is_nominal_sort tau2 $ >
  $ s_forall alpha a (
    (is_of_sort phi1 tau1) ->
    (is_of_sort phi2 tau2) ->
    (fresh_for (eVar a) (phi1 ,, phi2)) ->
    (fresh_for (eVar a) phi1) /\
    (fresh_for (eVar a) phi2)) $;

axiom S1 (alpha tau a phi: Pattern):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ (is_atom a alpha) ->
    (is_of_sort phi tau) ->
    ((swap a a phi) == phi) $;
axiom S2 {a b: EVar}
  (alpha tau phi: Pattern a b):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ s_forall alpha a (
    s_forall alpha b (
    (is_of_sort phi tau) ->
    ((swap (eVar a) (eVar b)
      (swap (eVar a) (eVar b) phi)) == phi))) $;
axiom S3 {a b: EVar} (alpha: Pattern a b):
  $ is_atom_sort alpha $ >
  $ s_forall alpha a (
    s_forall alpha b (
    ((swap (eVar a) (eVar b) (eVar a)) ==
    (eVar b)))) $;
axiom F1 {a b: EVar} (alpha tau phi: Pattern a b):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ s_forall alpha a (
    s_forall alpha b (
    (is_of_sort phi tau) ->
    ((fresh_for (eVar a) phi) /\
    (fresh_for (eVar b) phi) ->
    ((swap (eVar a) (eVar b) phi) == phi)))) $;
axiom F2 {a b: EVar} (alpha: Pattern a b):
  $ is_atom_sort alpha $ >
  $ s_forall alpha a (
    s_forall alpha b (
    (((eVar a) != (eVar b)) <->
    (fresh_for (eVar a) (eVar b))))) $;
axiom F3 {a b: EVar} (alpha1 alpha2: Pattern a b):
  $ is_atom_sort alpha1 $ >
  $ is_atom_sort alpha2 $ >
  $ s_forall alpha a (
    s_forall alpha b (
    (alpha1 != alpha2) ->
    ((eVar a) != (eVar b)))) $;
axiom F4 {a: EVar} (alpha tau phi: Pattern):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ (is_sorted_func (dom tau) phi) ->
    (s_exists alpha a (fresh_for (eVar a) phi)) $;
axiom A1 {a b: EVar} (alpha tau phi rho: Pattern a b):
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ s_forall alpha a (
    s_forall alpha b (
    (is_of_sort phi tau) ->
    (is_of_sort rho tau) ->
    (((abstraction (eVar a) phi) ==
    (abstraction (eVar b) rho)) <->
    (((eVar a) == (eVar b)) /\ (phi == rho)) \/
    ((fresh_for (eVar a) rho) /\
    ((swap (eVar a) (eVar b) phi) == rho))))) $;
axiom A2 (alpha tau: Pattern) {x a y: EVar}:
  $ is_atom_sort alpha $ >
  $ is_nominal_sort tau $ >
  $ s_forall (sort_abstraction alpha tau) x
    (s_exists alpha a (s_exists tau y
    (eVar x == abstraction (eVar a) (eVar y)))) $;

def coercion (a phi: Pattern) {.y: EVar}: Pattern =
  $ exists y (eVar y /\
    ((abstraction a (eVar y)) == phi)) $;
```

Below we have the axiomatization of lambda calculus:

```
term Var_sym: Symbol;
def Var: Pattern = $ sym Var_sym $;
def Vars: Pattern = $ dom Var $;
def is_var (p: Pattern): Pattern =
  $ is_of_sort p Var $;
axiom Var_atom: $ is_atom_sort Var $;

term Exp_sym: Symbol;
def Exp: Pattern = $ sym Exp_sym $;
def Exps: Pattern = $ dom Exp $;
def is_exp (p: Pattern): Pattern =
  $ is_of_sort p Exp $;
axiom Exp_sort: $ is_nominal_sort Exp $;

term lc_var_sym: Symbol;
def lc_var (phi: Pattern): Pattern =
  $ (sym lc_var_sym) @@ phi $;
axiom function_lc_var:
  $ ,(is_function '(sym lc_var_sym) '[Var] 'Exp) $;
axiom EV_lc_var {a b: EVar} (c: Pattern a b):
  $ s_forall Var a (s_forall Var b (
    (is_of_sort c Var) ->
    ((swap (eVar a) (eVar b) (lc_var c)) ==
    lc_var (swap (eVar a) (eVar b) c)))) $;

term lc_app_sym: Symbol;
def lc_app (phi rho: Pattern): Pattern =
  $ (sym lc_app_sym) @@ phi @@ rho $;
axiom function_lc_app:
   $ ,(is_function '(sym lc_var_app) '[Exp Exp] 'Exp) $;
axiom EV_lc_app {a b: EVar} (phi psi: Pattern a b):
  $ s_forall Var a (s_forall Var b (
    (is_of_sort phi Exp) ->
    (is_of_sort psi Exp) ->
    ((swap (eVar a) (eVar b) (lc_app phi psi)) ==
    lc_app
      (swap (eVar a) (eVar b) phi)
      (swap (eVar a) (eVar b) psi)))) $;

term lc_lam_sym: Symbol;
def lc_lam (phi: Pattern): Pattern =
  $ (sym lc_lam_sym) @@ phi $;
def lc_lam_a (a p: Pattern): Pattern =
  $ lc_lam (abstraction a p) $;
axiom function_lc_lam:
  $ ,(is_function '(sym lc_lam_sym)
    '[(sort_abstraction Var Exp)] 'Exp) $;
axiom EV_lc_lam {a b: EVar} (phi: Pattern a b):
  $ s_forall Var a (s_forall Var b (
    (is_of_sort phi (sort_abstraction Var Exp)) ->
    ((swap (eVar a) (eVar b) (lc_lam phi)) ==
    lc_lam (swap (eVar a) (eVar b) phi)))) $;

axiom no_junk {X: SVar}:
  $ Exps == mu X ( (lc_var Vars)
                 \/ (lc_app (sVar X) (sVar X))
                 \/ (lc_lam
                   (abstraction Vars (sVar X)))) $;
axiom function_subst:
  $ ,(is_function '(sym subst_sym)
    '[Var Exp Exp] 'Exp) $;
axiom EV_subst {a b: EVar} (c phi plug: Pattern a b):
  $ s_forall Var a (s_forall Var b (
    (is_var c) ->
    (is_exp phi) ->
    (is_exp plug) ->
    ((swap (eVar a) (eVar b) (subst c phi plug)) ==
    subst
      (swap (eVar a) (eVar b) c)
      (swap (eVar a) (eVar b) phi)
      (swap (eVar a) (eVar b) plug)))) $;
axiom subst_fresh {a: EVar} (phi plug: Pattern a):
  $ s_forall Var a (
    (is_exp phi) ->
    (is_exp plug) ->
    (fresh_for (eVar a) phi) ->
    ((subst (eVar a) phi plug) == phi)) $;
axiom subst_same_var {a: EVar} (plug: Pattern a):
  $ s_forall Var a (
    (is_exp plug) ->
    ((subst (eVar a) (lc_var (eVar a)) plug) ==
    plug)) $;
axiom subst_diff_var {a b: EVar} (plug: Pattern a b):
  $ s_forall Var a (
    s_forall Var b (
    ((eVar a) != (eVar b)) ->
    (is_exp plug) ->
    ((subst (eVar b) (lc_var (eVar a)) plug) ==
    (lc_var (eVar a))))) $;
axiom subst_var {a b: EVar} (plug: Pattern a b):
  $ s_forall Var a (
    s_forall Var b (
    ((eVar a) == (eVar b)) ->
    (is_exp plug) ->
    ((subst (eVar b) (lc_var (eVar a)) plug) ==
    plug))) $;
axiom subst_app {a: EVar} (phi1 phi2 plug: Pattern a):
  $ s_forall Var a (
    (is_exp plug) ->
    (is_exp phi1) ->
    (is_exp phi2) ->
    ((subst (eVar a) (lc_app phi1 phi2) plug) ==
    (lc_app
      (subst (eVar a) phi1 plug)
      (subst (eVar a) phi2 plug)))) $;
axiom subst_lam {a b: EVar} (plug phi: Pattern a b):
  $ s_forall Var a (
    s_forall Var b (
    ((eVar a) != (eVar b)) ->
    (fresh_for (eVar a) plug) ->
    (is_exp plug) ->
    (is_exp phi) ->
    ((subst
      (eVar b)
      (lc_lam (abstraction (eVar a) phi))
      plug) ==
      (lc_lam (abstraction
        (eVar a)
        (subst (eVar b) phi plug)))))) $;
```

Below is also a selection of proved statements related to this lambda calculus formalisation:

Induction principle:

```
theorem simple_induction_principle (exp_pred: Pattern):
  $ (is_exp exp_pred) ->
    ((lc_var Vars) C= exp_pred) ->
    ((lc_app exp_pred exp_pred) C= exp_pred) ->
    ((lc_lam (abstraction Vars exp_pred)) C=
      exp_pred) ->
    (Exps == exp_pred) $
```

Substitution lemma, proved using the induction principle above:

```
theorem subst_induction (a b phi plug1 plug2: Pattern)
  (diff_atoms_ab: $ a != b $)
  (a_var: $ is_sorted_func Vars a $)
  (b_var: $ is_sorted_func Vars b $)
  (phi_exp: $ is_exp phi $)
  (plug1_exp: $ is_exp plug1 $)
  (plug2_exp: $ is_sorted_func Exps plug2 $)
  (a_fresh: $ fresh_for a plug2 $):
  $ (subst b (subst a phi plug1) plug2) ==
    (subst a (subst b phi plug2)
             (subst b plug1 plug2)) $
```

Data-Availability Statement

The artifact of this paper can be found at [13].

Acknowledgements

We would like to thank the anonymous reviewers for their valuable and insightful feedback and also Mario Carneiro for his availability for technical consultations regarding Metamath Zero.

This work was supported by a Royal Society International Exchange grant (number IES\R2\212106).

References

[1] Mario Carneiro. 2020. Metamath Zero: Designing a theorem prover prover. In *Intelligent Computer Mathematics: 13th International Conference, CICM 2020, Bertinoro, Italy, July 26–31, 2020, Proceedings 13.* Springer, 71–88.

[2] Xiaohong Chen, Dorel Lucanu, and Grigore Roşu. 2021. Matching logic explained. *Journal of Logical and Algebraic Methods in Programming* 120 (2021), 100638. https://doi.org/10.1016/j.jlamp.2021.100638

[3] 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

[4] 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

[5] James Cheney and Maribel Fernández. 2022. Nominal Matching Logic. In *PPDP 2022: 24th International Symposium on Principles and Practice of Declarative Programming, Tbilisi, Georgia, September 20 - 22, 2022.* ACM, 5:1–5:15. https://doi.org/10.1145/3551357.3551375

[6] 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.

[7] Norman D. Megill. 2019. *Metamath: A Computer Language for Mathematical Proofs.* Lulu Press, Morrisville, North Carolina. http://us.metamath.org/downloads/metamath.pdf.

[8] Andrew M. Pitts. 2001. Nominal Logic: A First Order Theory of Names and Binding. In *TACS (LNCS, Vol. 2215).* Springer, 219–242.

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

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

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

[12] Nishant Rodrigues, Mircea Octavian Sebe, Xiaohong Chen, and Grigore Roşu. 2024. A Logical Treatment of Finite Automata. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer Nature Switzerland, Cham, 350–369.

[13] Octavian Mircea Sebe. [n. d.]. *NLML formalization.* https://doi.org/10.1145/3580443

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

[15] Christian Urban, Stefan Berghofer, and Michael Norrish. 2007. Barendregt's Variable Convention in Rule Inductions. 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, 35–50. https://doi.org/10.1007/978-3-540-73595-3_4

Received 2024-09-17; accepted 2024-11-19