Dependent Types for Nominal Terms with Atom Substitutions

Elliot Fairweather[1], Maribel Fernández[1], Nora Szasz[2], and Alvaro Tasistro[2]

1 **King's College London, UK**
2 **Universidad ORT Uruguay, Uruguay**

Abstract

Nominal terms are an extended first-order language for specifying and verifying properties of syntax with binding. Founded upon the semantics of nominal sets, the success of nominal terms with regard to systems of equational reasoning is already well established. This work first extends the untyped language of nominal terms with a notion of non-capturing atom substitution for object-level names and then proposes a dependent type system for this extended language. Both these contributions are intended to serve as a prelude to a future nominal logical framework based upon nominal equational reasoning and thus an extended example is given to demonstrate that this system is capable of encoding various other formal systems of interest.

1998 ACM Subject Classification F.4.1 Mathematical Logic, lambda calculus and related systems, D.3.3 Language Constructs and Features, data types and structures, frameworks

Keywords and phrases α-equivalence, nominal term, substitution, dependent type

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.180

1 Introduction

There exist many formal systems described by a syntax that makes use of name binding constructs. Nominal terms [35, 15, 14], are, by now, a well-established approach to the specification and verification of properties of such languages and systems. Based upon the name abstraction semantics of nominal sets [23], nominal terms use the properties of permutations of object-level names or 'atoms' to provide an explicit formalisation for both the use of side conditions on names and for the axiomatisation of alpha-equivalence between object-level terms. Following the development of efficient algorithms for matching [4] and unification [5, 25], nominal terms have been applied in rewriting [15, 16, 12] and unoriented equational reasoning [22, 10].

In these works, the capture-avoiding substitution used in many systems of interest has, thus far, needed to be encoded by explicit rewrite rules or axioms for the syntax in question. The first contribution of the present work addresses this issue by extending the language of nominal terms with a notion of capture-avoiding atom substitution at the object-level. Definitions of freshness, alpha-equivalence and matching together with informal proofs of decidability are provided for the new syntax.

The second contribution of this paper is the definition of a proposed dependent type system for the extended language. The language of the type system presented in this paper is user-defined; users define an interdependent signature of type- and term-constructors of interest and give their type declarations. It is the responsibility of the user to maintain the adequacy of their encoding and thus to declare types in accordance with the system to be

13th International Conference on Typed Lambda Calculi and Applications (TLCA'15).
Editor: Thorsten Altenkirch; pp. 180–195

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

formalised. In keeping with earlier work on nominal equational reasoning and in contrast with previous work on nominal type theories [8, 9, 32], the system given here remains 'lambda-free'.

Both these contributions are presented as foundations for the future development of a logical framework based upon nominal equational reasoning. A logical framework is a formal system that provides the facility to define a number of other formal systems, such as programming languages, mathematical structures and logics, by abstraction of their common features [24, 2, 27, 30, 29]. Many logical frameworks are developed as a type system; the minimum expressive power desirable of such a type system is that types be allowed to depend upon terms.

The type system presented here is one of such expressive power and to demonstrate this an extended example is given in the form of a specification for first-order logic for which adequacy is proven. Equality axioms are not yet considered in this prototype 'framework' but in the future it is expected that the system be expanded to include a user-defined nominal equational theory, specified as a set of equality axioms or rewriting rules in the style of [20, 15, 16].

2 Syntax

Consider countably infinite, pairwise disjoint sets of **atoms**, $a, b, c, \ldots \in \mathbb{A}$, **variables**, $X, Y, Z, \ldots \in \mathbb{X}$, **term-formers**, $\mathsf{f}, \mathsf{g}, \ldots \in \mathbb{F}$, and **type-formers**, $\mathsf{C}, \mathsf{C}', \ldots \in \mathbb{C}$. The syntax of **permutations**, π, **atom substitutions**, ϑ, **pseudo-terms**, s, t, and **pseudo-types**, σ, τ, is defined by mutual induction and generated by the grammar in Definition 1.

▶ **Definition 1** (Syntax).

$$\begin{array}{llllll} \pi & ::= & \mathsf{id} \mid \pi * (a\ b) & s,\ t & ::= & a \mid [a\colon \sigma]\, t \mid \mathsf{f}\, t \mid (t_1,\ \ldots,\ t_n) \mid\ \vartheta \vert \pi \cdot X \\ \vartheta & ::= & \mathsf{id} \mid [a \mapsto t] * \vartheta & \sigma,\ \tau & ::= & [a\colon \sigma]\, \tau \mid \mathsf{C}\, t \mid (\tau_1 \times\ \ldots \times \tau_n) \end{array}$$

A **permutation** is a bijection on the set of atoms, $\mathbb{A}$, represented as a list of swappings, such that $\pi(a) \neq a$ for finitely many atoms, $a \in \mathbb{A}$. $\pi(a)$ is easily computed using swappings (we omit the inductive definition). An **atom substitution** is a mapping from atoms to pseudo-terms, equal to the identity mapping but for finitely many arguments. This mapping is represented as a list of pairs, ϑ, of the form, $[a_i \mapsto t_i]$ such that the atoms, a_i, are *pairwise distinct*. This list is interpreted as a set of *simultaneous* bindings and not as a sequence, and thus the value of an atom substitution is determined directly from the syntactic representation. The final id and 'list cons' operators in the syntax for both permutations and atom substitutions are commonly omitted. Atom substitutions act upon pseudo-terms and pseudo-types by instantiating atoms and are 'capture-avoiding'. Atom substitutions suspend upon variables and are applied *after* a suspended permutation.

The constructions for pseudo-terms are called respectively **atom terms**, **abstractions**, **function applications**, **tuples** (where $n \geq 0$) and **moderated variables** and those for pseudo-types, **abstraction types**, **constructed types** and **product types** ($n \geq 0$). Abbreviate $\mathsf{f}\,()$ as f, and $\mathsf{C}\,()$ as C. Let $M, N, \ldots$ range over elements of the union of the set of pseudo-terms and set of pseudo-types. There is only one kind of well-formed types: type.

▶ **Definition 2** (Permutation Action).

$$\pi \cdot \mathsf{id} \triangleq \mathsf{id} \qquad \pi \cdot ([a \mapsto t] * \vartheta) \triangleq [\pi(a) \mapsto \pi \cdot t] * (\pi \cdot \vartheta)$$

$$\pi \cdot a \triangleq \pi(a) \qquad \pi \cdot (\vartheta\big|\pi' \cdot X) \triangleq \pi \cdot \vartheta\big|(\pi \mathbin{@} \pi') \cdot X$$
$$\pi \cdot [a\colon \sigma]\, t \triangleq [\pi(a)\colon \pi \cdot \sigma]\, (\pi \cdot t) \qquad \pi \cdot [a\colon \sigma]\, \tau \triangleq [\pi \cdot a\colon \pi \cdot \sigma]\, (\pi \cdot \tau)$$
$$\pi \cdot \mathsf{f}\, t \triangleq \mathsf{f}\, (\pi \cdot t) \qquad \pi \cdot \mathtt{C}\, t \triangleq \mathtt{C}\, (\pi \cdot t)$$
$$\pi \cdot (t_1,\ \ldots,\ t_n) \triangleq (\pi \cdot t_1,\ \ldots,\ \pi \cdot t_n) \qquad \pi \cdot (\tau_1 \times \ldots \times \tau_n) \triangleq (\pi \cdot \tau_1 \times \ldots \times \pi \cdot \tau_n)$$

Call $a \# M$ a **freshness constraint**. Let $\Delta,\ \nabla$, range over sets of freshness constraints of the form $a \# X$; call such sets **freshness contexts**. Write $\Delta \vdash a \# M$ when a derivation exists using the rules given in Definition 3 below; in rule $(atm)^{\#}$ we assume $a \neq b$.

▶ **Definition 3** (Freshness Relation).

$$\frac{}{\Delta \vdash a \# b}\,(atm)^{\#} \qquad \frac{\mathsf{img}^{\#}(\Delta,\ a,\ \vartheta\big|\pi \cdot X) \quad a \in \mathsf{dom}(\vartheta)}{\Delta \vdash a \# \vartheta\big|\pi \cdot X}\,(var : aa)^{\#}$$

$$\frac{\mathsf{img}^{\#}(\Delta,\ a,\ \vartheta\big|\pi \cdot X) \quad a \notin \mathsf{dom}(\vartheta) \quad \pi^{-1}(a) \# X \in \Delta}{\Delta \vdash a \# \vartheta\big|\pi \cdot X}\,(var : ab)^{\#}$$

$$\frac{\Delta \vdash a \# \tau}{\Delta \vdash a \# [a\colon \tau]\, s}\,(abs : aa)^{\#} \qquad \frac{\Delta \vdash a \# s \quad \Delta \vdash a \# \tau}{\Delta \vdash a \# [b\colon \tau]\, s}\,(abs : ab)^{\#}$$

$$\frac{\Delta \vdash a \# s_1 \ \ldots\ \Delta \vdash a \# s_n}{\Delta \vdash a \# (s_1,\ \ldots,\ s_n)}\,(tpl)^{\#} \qquad \frac{\Delta \vdash a \# s}{\Delta \vdash a \# \mathsf{f}\, s}\,(app)^{\#}$$

$$\frac{\Delta \vdash a \# \sigma}{\Delta \vdash a \# [a\colon \sigma]\, \tau}\,(abt : aa)^{\#} \qquad \frac{\Delta \vdash a \# \tau \quad \Delta \vdash a \# \sigma}{\Delta \vdash a \# [b\colon \sigma]\, \tau}\,(abt : ab)^{\#}$$

$$\frac{\Delta \vdash a \# \tau_1 \ \ldots\ \Delta \vdash a \# \tau_n}{\Delta \vdash a \# (\tau_1 \times \ldots \times \tau_n)}\,(prd)^{\#} \qquad \frac{\Delta \vdash a \# t}{\Delta \vdash a \# \mathtt{C}\, t}\,(cns)^{\#}$$

The main differences with respect to the freshness relation for nominal terms are the introduction of new rules for types, $(abt : aa)^{\#}$, $(abt : ab)^{\#}$, $(prd)^{\#}$ and $(cns)^{\#}$, and the rules for moderated variables, $(var : aa)^{\#}$ and $(var : ab)^{\#}$, which take into account suspended atom substitutions as well as suspended permutations. The notation, $\mathsf{img}^{\#}(\Delta,\ a,\ \vartheta\big|\pi \cdot X)$, in these rules, defines the conditions necessary for freshness with regard to the suspended atom substitution and is an abbreviation of the following finite set of hypotheses.

$$\left\{ \left(\Delta \vdash a \# \vartheta(\pi(b))\right) \vee \left(b \# X \in \Delta\right) \mid b \in \mathbb{A},\ \pi(b) \in \mathsf{dom}(\vartheta) \right\}$$

This ensures that the substitution will not introduce the atom a when it is applied to an instance of X. However this disjunction of conditions results in the possibility of multiple derivations $\Delta_i \vdash a \# t$ for a given freshness constraint $a \# t$. If one considers the suspended atom substitution, ϑ, to be id, the conditions upon ϑ are satisfied vacuously and the two rules clearly reduce to that for nominal terms.

Call $M \approx_{\alpha} N$ an **alpha-equality constraint** and write $\Delta \vdash M \approx_{\alpha} N$ when a derivation exists using the rules given in Definition 4 below. Note that because alpha-equivalence is defined using freshness, again multiple possible derivations may exist for a given constraint.

▶ **Definition 4** (Alpha-equivalence Relation).

$$\frac{}{\Delta \vdash a \approx_\alpha a}\,(atm)^\alpha \qquad \frac{\forall a \in \mathsf{ds}(\vartheta_1|\pi_1,\ \vartheta_2|\pi_2).\ a \# X \in \Delta}{\Delta \vdash \vartheta_1|\pi_1 \cdot X \approx_\alpha \vartheta_2|\pi_2 \cdot X}\,(var)^\alpha$$

$$\frac{\Delta \vdash s_1 \approx_\alpha t_1\ \ldots\ \Delta \vdash s_n \approx_\alpha t_n}{\Delta \vdash (s_1,\ \ldots,\ s_n) \approx_\alpha (t_1,\ \ldots,\ t_n)}\,(tpl)^\alpha \qquad \frac{\Delta \vdash \sigma \approx_\alpha \tau\ \ \Delta \vdash s \approx_\alpha t}{\Delta \vdash [a\colon \sigma]\, s \approx_\alpha [a\colon \tau]\, t}\,(abs : aa)^\alpha$$

$$\frac{\Delta \vdash s \approx_\alpha t}{\Delta \vdash \mathsf{f}\, s \approx_\alpha \mathsf{f}\, t}\,(app)^\alpha \qquad \frac{\Delta \vdash \sigma \approx_\alpha \tau\ \ \Delta \vdash a \# t\ \ \Delta \vdash s \approx_\alpha (a\ b) \cdot t}{\Delta \vdash [a\colon \sigma]\, s \approx_\alpha [b\colon \tau]\, t}\,(abs : ab)^\alpha$$

$$\frac{\Delta \vdash \sigma_1 \approx_\alpha \tau_1\ \ldots\ \Delta \vdash \sigma_n \approx_\alpha \tau_n}{\Delta \vdash (\sigma_1 \times \ldots \times \sigma_n) \approx_\alpha (\tau_1 \times \ldots \times \tau_n)}\,(prd)^\alpha \qquad \frac{\Delta \vdash \sigma_1 \approx_\alpha \tau_1\ \ \Delta \vdash \sigma_2 \approx_\alpha \tau_2}{\Delta \vdash [a\colon \sigma_1]\, \sigma_2 \approx_\alpha [a\colon \tau_1]\, \tau_2}\,(abt : aa)^\alpha$$

$$\frac{\Delta \vdash s \approx_\alpha t}{\Delta \vdash \mathsf{C}\, s \approx_\alpha \mathsf{C}\, t}\,(cns)^\alpha \qquad \frac{\Delta \vdash \sigma_1 \approx_\alpha \tau_1\ \ \Delta \vdash a \# \tau_2\ \ \Delta \vdash \sigma_2 \approx_\alpha (a\ b) \cdot \tau_2}{\Delta \vdash [a\colon \sigma_1]\, \sigma_2 \approx_\alpha [b\colon \tau_1]\, \tau_2}\,(abt : ab)^\alpha$$

This presentation of alpha-equivalence is defined by induction on the size of the pair, (M, N), and is both syntax-directed and decidable when considered as a recursive predicate. It is a generalisation of the notion of alpha-equivalence on nominal terms.

The only case that is not straightforward is again that of a moderated variable, $(var)^\alpha$. Here it is important to remember that both permutations and atom substitutions are finite mappings and that the image of a suspended atom substitution is given as a sub-term of the syntax of the moderated variable. Thus, the disagreement set of two suspensions, $\vartheta_1|\pi_1$ and $\vartheta_2|\pi_2$, written $\mathsf{ds}(\vartheta_1|\pi_1,\ \vartheta_2|\pi_2)$, is also finite and may be defined as $\{a \mid \vartheta_1(\pi_1(a)) \not\approx_\alpha \vartheta_2(\pi_2(a)),\ a \in \mathbb{A}\}$, which although recursive, is of decreasing size. The reflexivity, symmetry and transitivity of the alpha-equivalence relation have been proved by adapting the proofs given in [15], themselves simplified from those in [35].

The action of an atom substitution, ϑ, upon a pseudo-term or pseudo-type, M, written, $M\,\vartheta$, is defined by induction in the presence of a freshness context, Δ, in Definition 5. For the sake of clarity of presentation this freshness context is not explicitly written throughout the definition. Let ϑ^{-a} denote the atom substitution ϑ restricted to the domain, $\mathsf{dom}(\vartheta) \setminus \{a\}$. The composition of two atom substitutions, written $\vartheta_1 \circ \vartheta_2$, is defined as the atom substitution equivalent to applying ϑ_1 followed by ϑ_2. The syntactic construction of such a composition built from two substitutions represented as sets of bindings can be defined by adapting the algorithm described in [3, 2.1]. Note that this operation itself uses the action of an atom substitution upon pseudo-term and so must be defined simultaneously with Definition 5 and is also parameterised by the freshness context, Δ.

▶ **Definition 5** (Action of Atom Substitution).

$$(\vartheta'|\pi \cdot X)\,\vartheta \triangleq (\vartheta' \circ \vartheta)|\pi \cdot X \qquad a\,\vartheta \triangleq \vartheta(a);\ a \in \mathsf{dom}(\vartheta) \qquad a\,\vartheta \triangleq a;\ a \notin \mathsf{dom}(\vartheta)$$

$$([a\colon \sigma]\, s)\,\vartheta \triangleq [c\colon \sigma\,\vartheta]\,((a\ c) \cdot s)\,\vartheta^{-c};\ \Delta \vdash c \# s,\ c \# \mathsf{img}(\vartheta)$$

$$(\mathsf{f}\, s)\,\vartheta \triangleq \mathsf{f}\,(s\,\vartheta) \qquad (t_1,\ \ldots,\ t_n)\,\vartheta \triangleq (t_1\,\vartheta,\ \ldots,\ t_n\,\vartheta)$$

$$([a\colon \sigma]\, \tau)\,\vartheta \triangleq [c\colon \sigma\,\vartheta]\,((a\ c) \cdot \tau)\,\vartheta^{-c};\ \Delta \vdash c \# \tau,\ c \# \mathsf{img}(\vartheta)$$

$$(\mathsf{C}\, s)\,\vartheta \triangleq \mathsf{C}\,(s\,\vartheta) \qquad (\tau_1 \times \ldots \times \tau_n)\,\vartheta \triangleq (\tau_1\,\vartheta \times \ldots \times \tau_n\,\vartheta)$$

The capture-avoidance of unabstracted atoms is ensured by the fact that when an atom substitution acts upon an abstraction or abstraction type, a suitable alpha-equivalent representative is first chosen with respect to the freshness context, Δ. In practice, this

presentation will result in the creation of freshness constraints for atoms, newly-generated with respect to the system as a whole, and is similar to the approach taken in [16]. A suitable 'even fresher' atom always exists, and it is one's right to add constraints for that atom to the freshness context, a fact which is taken advantage of below in Definition 9. Any implementation of this definition as a recursive function must accommodate a suitable mechanism for the generation of such names; this is most easily achieved by the threading of global state throughout the function or by the use of a global choice function that returns the next available name. Atom substitutions work uniformly on alpha-equivalence classes of pseudo-terms and pseudo-types.

▶ **Definition 6** (Variable Substitutions)**.** A **variable substitution** is a mapping from variables to pseudo-terms, equal to the identity mapping but for finitely many arguments, and written as a set of bindings $[X_1 \mapsto s_1] \ldots [X_n \mapsto s_n]$, such that the variables, $X_1, \ldots, X_n$, are pairwise distinct.

The action of variable substitutions upon atom substitutions, pseudo-terms and pseudo-types is given in Definition 7. A variable substitution, θ, acts upon an atom substitution, ϑ, by instantiating the variables occurring in the pseudo-terms of the image of ϑ, and is written $\vartheta\,\theta$. Note that the instantiation of a variable requires the application of an atom substitution to a pseudo-term and thus the action of variable substitutions is also parameterised by a freshness context, which again is left implicit in the definition below.

▶ **Definition 7** (Variable Substitution Action)**.**

$$\mathsf{id}\,\theta \triangleq \mathsf{id} \qquad ([a \mapsto t] * \vartheta)\,\theta \triangleq [a \mapsto t\,\theta] * (\vartheta\,\theta)$$

$$a\,\theta \triangleq a$$

$$(\vartheta \big| \pi \cdot X)\,\theta \triangleq (\pi \cdot \theta(X))\,(\vartheta\,\theta);\ X \in \mathsf{dom}(\theta) \qquad (\vartheta \big| \pi \cdot X)\,\theta \triangleq (\vartheta\,\theta) \big| \pi \cdot X;\ X \notin \mathsf{dom}(\theta)$$

$$([a\colon \sigma]\,t)\,\theta \triangleq [a\colon \sigma\,\theta]\,(t\,\theta) \qquad (\mathsf{f}\,t)\,\theta \triangleq \mathsf{f}\,(t\,\theta) \qquad (t_1, \ldots, t_n)\,\theta \triangleq (t_1\,\theta, \ldots, t_n\,\theta)$$

$$([a\colon \sigma]\,\tau)\,\theta \triangleq [a\colon \sigma\,\theta]\,(\tau\,\theta) \qquad (\mathtt{C}\,t)\,\theta \triangleq \mathtt{C}\,(t\,\theta) \qquad (\tau_1 \times \ldots \times \tau_n)\,\theta \triangleq (\tau_1\,\theta \times \ldots \times \tau_n\,\theta)$$

The type system introduced in Section 3 requires a formalisation of matching to check that term-formers and type-formers are used in a way that is consistent with their respective type declarations. The concepts of constraint problems and matching are now therefore extended to nominal terms with atom substitutions.

Let C range over freshness and alpha-equality constraints; a **constraint problem**, $\mathcal{C}$, is an arbitrary set of such constraints. Extend the above notations for the derivability of constraints element-wise to constraint problems; thus, write $\Delta \vdash \{C_1, \ldots, C_n\}$ for $\Delta \vdash C_1$, $\ldots$, $\Delta \vdash C_n$. Substitution action extends naturally to constraints and constraint problems.

▶ **Definition 8** (Matching Problem)**.** Given a constraint problem, $\mathcal{C}$, $\{\ldots, a_j \# Q_j, \ldots, M_i \approx_\alpha N_i, \ldots\}$, a corresponding **matching problem** is defined if $(\bigcup_i \mathsf{vars}(M_i)) \cap (\bigcup_i \mathsf{vars}(N_i)) = \varnothing$ and is written $\{\ldots, a_j \# Q_j, \ldots, M_i \stackrel{?}{\approx}_\alpha N_i, \ldots\}$.

A solution to such a problem, if one exists, is a pair, (Δ, θ), of a freshness context, Δ, and a variable substitution, θ, such that $\mathsf{dom}(\theta) \subseteq \bigcup \mathsf{vars}(M_i)$ and $\Delta \vdash \mathcal{C}\,\theta$.

Informally, this says that a matching problem is a constraint problem in which one adds the restriction that the variables in the left-hand sides of alpha-equality constraints are disjoint from the variables in the right-hand sides and that only variables in the left-hand

sides of equality constraints may be instantiated. There may be several Δ_i such that $\Delta_i \vdash \mathcal{C}\,\theta$. As in the case of nominal terms, one can define an ordering between solutions (Δ_i, θ) and define a most general solution as a least element in the ordering. However, unlike nominal matching, here there is no unique most general solution.

The situation is similar for logical frameworks based on the lambda calculus. The solution there is to restrict the form of matching problems. Inspired by LF, [30], the type system in Section 3 is designed so that one only needs to match against a pattern term, t, such that for any variable, $X \in \mathsf{vars}(t)$, then $\mathsf{id}\big|\pi \cdot X$ is a sub-term of t. Thus, the value of X is uniquely determined. Note that a solution may only instantiate variables in the pattern and so if an atom substitution occurs in the matched term then it can be treated as a constant sub-term. Using this assumption, if a matching problem has a solution, there is a unique most general one. The algorithm to compute it is similar to that used to check alpha-equivalence, except that when a variable sub-term, $\vartheta\big|\pi \cdot X$, of the pattern is being matched against, if ϑ is not id then that constraint is postponed until after the constraint for the occurrence of the $\mathsf{id}\big|\pi \cdot X$ sub-term in the pattern has been solved and a unique variable substitution generated.

In addition, it is also important to note that due to the action of atom substitutions suspended upon variables, the freshness context of a solution may contain constraints for atoms, $a \notin \mathsf{atms}(\mathcal{C})$.

▶ **Definition 9** (Pattern Matching Problem)**.** A **pattern matching problem**, consists of two pseudo-terms-in-context or two pseudo-types-in-context, $\nabla \vdash M$ and $\Delta \vdash N$, to be matched, where $\mathsf{vars}(\nabla \vdash M) \cap \mathsf{vars}(\Delta \vdash N) = \varnothing$ and is written $(\nabla \vdash M) \stackrel{?}{\approx}_\alpha (\Delta \vdash N)$.

A solution to such a problem, if one exists, is a variable substitution, θ, such that (∇', θ) is a solution to the matching problem $\nabla \cup \{M \stackrel{?}{\approx}_\alpha N\}$ and there exists a freshness context $\Delta^\#$ (of which each constraint, $a \# X$, is such that $a \notin \mathsf{atms}(M) \cup \mathsf{atms}(N) \cup \mathsf{atms}(\nabla)$), such that $\Delta \cup \Delta^\# \vdash \nabla'$.

A **newly-freshened variant** of a term, t, is a term, written t^ν, in which all the atoms and variables have been replaced by newly generated atoms and variables with respect to those occurring in t (and maybe other elements of syntax, always specified.)

Closed terms were introduced in [15] and shown there to be decidable by an algorithm using newly-freshened variants and nominal matching. Intuitively, a closed term has no unabstracted atoms and all occurrences of a variable must appear under the *same* abstracted atoms. Here, closedness can be checked in a similar way using the matching algorithm mentioned above.

3 Type System

This section starts by introducing the syntax of environments, declarations and judgements used in this type system. The validity of environments (Definition 11), validity of sets of declarations (Definition 10), and derivability of typing judgements (Definition 13), are then defined by mutual induction.

A **type association** is a pair of a variable, X, and a type, σ, written $(X\colon \sigma)$ or an atom and a type, written $(a\colon \sigma)$. A **pseudo-environment**, Γ, is an ordered list of type associations. A pseudo-environment may contain at most one type association for each variable and atom. Here, let $\Gamma \bowtie (a\colon \tau)$, denote the result of **appending** $(a\colon \tau)$ to the end of the list that represents the pseudo-environment, Γ (similarly for a variable association) and let this notation be extended element-wise to lists of associations. The association available for a given atom, a, in Γ is denoted by Γ_a. If there is no type association for a in Γ then Γ_a is

undefined, written $\bot$ (similarly for a variable.) The domain of Γ, denoted $\mathsf{dom}(\Gamma)$, and image of Γ, denoted $\mathsf{img}(\Gamma)$, are defined as usual: $\mathsf{dom}(\Gamma) = \{a \in \mathbb{A} \mid \Gamma_a \neq \bot\} \cup \{X \in \mathbb{X} \mid \Gamma_X \neq \bot\}$ and $\mathsf{img}(\Gamma) = \{\tau \mid \exists a, \Gamma_a = \tau\} \cup \{\tau \mid \exists X, \Gamma_X = \tau\}$. It is important to note that type associations for variables are *never* appended by the typing rules, however the rules for abstractions and abstraction types do append type associations for atoms.

A **pseudo-declaration**, $\Gamma \Vdash \Delta \vdash \mathsf{f}\, t \colon \langle \sigma \hookrightarrow \tau \rangle$ or $\Gamma \Vdash \Delta \vdash \mathtt{C}\, t \colon \langle \sigma \hookrightarrow \mathsf{type} \rangle$, states the type associations and freshness constraints that a term must satisfy in order that an application or constructed type built from that term be well-formed. Thus, informally, this says that if under the type associations in Γ, and the freshness constraints in Δ, t has type σ then $\mathsf{f}\, t$ has type τ, or similarly that $\mathtt{C}\, t$ is of the kind type. In practice, users need not give complete declarations; it is sufficient to write $\Gamma \Vdash \Delta \vdash \mathsf{f}\, t \colon \tau$ or $\Gamma \Vdash \Delta \vdash \mathtt{C}\, t \colon \mathsf{type}$ and the system will infer the complete declaration by computing the type of t.

Pseudo-declarations are given for a term-former or type-former *together* with an argument term in order to allow the use of atoms of that argument in the type of the application or constructed type; for example, see the declarations for $\mathsf{all_i}$ and $\mathsf{all_e}$ in Section 5.

A **pseudo-judgement**, $\Gamma \Vdash_\Sigma \Delta \vdash t \colon \tau$ or $\Gamma \Vdash_\Sigma \Delta \vdash \tau \colon \mathsf{type}$, specifies that under a given environment, set of declarations and freshness context, either a term has a particular type or a type is well-formed.

A valid set of declarations, Σ, written $\mathsf{validD}(\Sigma)$, is defined inductively as follows.

▶ **Definition 10** (Valid Set of Declarations)**.**

- The empty set of declarations, $\varnothing$, is valid; $\mathsf{validD}(\varnothing)$.
- If $\mathsf{validD}(\Sigma)$, $\Gamma \Vdash \Delta \vdash \mathsf{f}\, t \colon \langle \sigma \hookrightarrow \tau \rangle$ is a valid declaration under the following conditions.
  - $\Gamma \Vdash_\Sigma \Delta \vdash t \colon \sigma$, where $\mathsf{vars}(\sigma) \subseteq \mathsf{vars}(t) \cup \mathsf{vars}(\tau)$
  - $\Gamma \Vdash_\Sigma \Delta \vdash \tau \colon \mathsf{type}$, where τ is not an abstraction type
  - $\Delta \vdash \{t,\ \sigma,\ \tau\}$ is closed
  - for any variable, $X \in \mathsf{vars}(t) \cup \mathsf{vars}(\tau)$, then the sub-term, $\mathsf{id}\big|\pi \cdot X$, occurs in either t or τ.

  Then, provided that there is no declaration in Σ for the term-former, f, it holds that $\mathsf{validD}(\Sigma \cup \{\Gamma \Vdash \Delta \vdash \mathsf{f}\, t \colon \langle \sigma \hookrightarrow \tau \rangle\})$.
- If $\mathsf{validD}(\Sigma)$, then $\Gamma \Vdash \Delta \vdash \mathtt{C}\, t \colon \langle \sigma \hookrightarrow \mathsf{type} \rangle$ is a valid declaration under the following conditions.
  - $\Gamma \Vdash_\Sigma \Delta \vdash t \colon \sigma$, where $\mathsf{vars}(\sigma) \subseteq \mathsf{vars}(t)$
  - $\Delta \vdash \{t,\ \sigma\}$ is closed
  - for any variable, $X \in \mathsf{vars}(t)$ then the sub-term, $\mathsf{id}\big|\pi \cdot X$, occurs in t

  Then, provided that there is no declaration in Σ for the type-former, $\mathtt{C}$, $\mathsf{validD}(\Sigma \cup \{\Gamma \Vdash \Delta \vdash \mathtt{C}\, t \colon \langle \sigma \hookrightarrow \mathsf{type} \rangle\})$.

Assuming a freshness context, Δ, and a valid set of declarations, Σ, a valid environment, written $\mathsf{validE}(\Gamma,\ \Sigma,\ \Delta)$, is defined as follows.

▶ **Definition 11** (Valid Environments)**.**

- The empty list of type associations, $-$, is a valid environment; $\mathsf{validE}(-,\ \Sigma,\ \Delta)$.
- If $\mathsf{validE}(\Gamma,\ \Sigma,\ \Delta)$, then $\mathsf{validE}(\Gamma \bowtie (a \colon \tau),\ \Sigma,\ \Delta \cup \Delta')$, for any Δ' that does not mention atoms in $\mathsf{dom}(\Gamma)$, provided that $a \notin \mathsf{dom}(\Gamma)$, $\Gamma \Vdash_\Sigma \Delta \vdash \tau \colon \mathsf{type}$.
- If $\mathsf{validE}(\Gamma,\ \Sigma,\ \Delta)$, then $\mathsf{validE}(\Gamma \bowtie (X \colon \tau),\ \Sigma,\ \Delta \cup \Delta')$, for any Δ' that does not mention atoms in $\mathsf{dom}(\Gamma)$, provided that $X \notin \mathsf{dom}(\Gamma)$, for any atom, a, such that $\Delta \vdash a \# X$ then $\Delta \vdash a \# \tau, a \# \mathsf{img}(\Gamma)$, and $\Gamma \Vdash_\Sigma \Delta \vdash \tau \colon \mathsf{type}$.

Informally, the final condition upon variable associations says that in a valid environment, if an atom cannot occur unabstracted in an instance of a variable, X, then it cannot occur unabstracted in the typing of that variable either.

Typing judgements are derived inductively using the rules given in Definition 13. The declarations provided by the user are required in the rules for term-formers, $(app)^\tau$, and type-formers, $(cns)^\tau$. Declarations are matched to pseudo-terms and pseudo-types using pattern matching and therefore each time an application or constructed type is typed a newly-freshened variant of the declaration for that term-former or type-former is generated; we write $\Theta \Vdash \nabla \vdash \mathsf{f}\, s : \langle \sigma \hookrightarrow \rho \rangle \in \Sigma^{\prime\prime}$ (similarly for a constructed type) to emphasise the fact that such a newly-freshened variant of a declaration is being used. In the rule for a variable, $(var)^\tau$, a predicate, typV is used.

▶ **Definition 12.** Let $\Gamma\,|_X$, for $X \in \mathsf{dom}(\Gamma)$, be the list of associations in Γ up to but not including the pair $(X : \tau)$. Write $\mathsf{typV}(\Gamma,\ \Sigma,\ \Delta,\ \vartheta\big|\pi,\ X)$ if the following conditions hold.

- for any atom $a \in \mathsf{dom}(\Gamma\,|_X)$, either $\Delta \vdash a \# X$ or $\Gamma \Vdash_\Sigma \Delta \vdash \vartheta(\pi(a)) : (\pi \cdot \Gamma_a)\,\vartheta$
- for any variable $Y \in \mathsf{dom}(\Gamma\,|_X)$, $\Gamma \Vdash_\Sigma \Delta \vdash \vartheta\big|\pi \cdot Y : (\pi \cdot \Gamma_Y)\,\vartheta$
- $\Gamma \Vdash_\Sigma \Delta \vdash (\pi \cdot \Gamma_X)\,\vartheta : \mathsf{type}$

This definition indicates that for any atom, a, that can occur unabstracted in an instance of the variable, X, then $\vartheta(\pi(a))$ must be typeable with a type compatible with the type of a. Similarly, variables that may be used in the typing of X (i.e., that occur in Γ before X) also have a type compatible with $\vartheta\big|\pi$.

In the rule $(abs)^\tau$, the atom b is not in the set of atoms occurring in the judgement, $\Gamma \Vdash_\Sigma \Delta \vdash [a : \sigma]\, t : [a : \sigma]\, \tau$ and the freshness context, $\Delta^\#$, is such that $\Delta^\# \vdash b \# t,\ b \# \sigma,\ b \# \tau$. Similarly in $(abt)^\tau$, b is not in the set of atoms occurring in $\Gamma \Vdash_\Sigma \Delta \vdash [a : \sigma]\, \tau : \mathsf{type}$ and $\Delta^\#$, is such that $\Delta^\# \vdash b \# \sigma,\ b \# \tau$.

In the rule $(app)^\tau$, $\mathsf{sol}(\theta)$ means that the variable substitution, θ, is a most general solution to the pattern matching problem $(\nabla \vdash (\mathtt{C}\, s \times \rho)) \stackrel{?}{\approx}_\alpha (\Delta \cup \Delta^\# \vdash (\mathtt{C}\, t \times \tau))$, where $\Delta^\#$ is a freshness context such that $\mathsf{atms}(s) \cup \mathsf{atms}(\rho) \# \mathsf{vars}(t) \cup \mathsf{vars}(\tau)$ and $\mathtt{C}$, is an arbitrary type-constructor used so that the argument term and application type may both be included within the same pattern matching problem. Similarly in $(cns)^\tau$, θ is a most general solution to $(\nabla \vdash s) \stackrel{?}{\approx}_\alpha (\Delta \cup \Delta^\# \vdash t)$ where $\Delta^\#$ is $\mathsf{atms}(s) \# \mathsf{vars}(t)$.

▶ **Definition 13** (Typing rules)**.**

$$\frac{\mathsf{validE}(\Gamma,\ \Sigma,\ \Delta) \quad (\pi \cdot \Gamma_X)\,\vartheta = \tau \quad \mathsf{typV}(\Gamma,\ \Sigma,\ \Delta,\ \vartheta\big|\pi,\ X)}{\Gamma \Vdash_\Sigma \Delta \vdash \vartheta\big|\pi \cdot X : \tau}\,(var)^\tau$$

$$\frac{\mathsf{validE}(\Gamma,\ \Sigma,\ \Delta) \quad \Gamma_a = \tau}{\Gamma \Vdash_\Sigma \Delta \vdash a : \tau}\,(atm)^\tau \qquad \frac{\Gamma \bowtie (b : \sigma) \Vdash_\Sigma \Delta \cup \Delta^\# \vdash (a\ b) \cdot t : (a\ b) \cdot \tau}{\Gamma \Vdash_\Sigma \Delta \vdash [a : \sigma]\, t : [a : \sigma]\, \tau}\,(abs)^\tau$$

$$\frac{\mathsf{validE}(\Gamma,\ \Sigma,\ \Delta)}{\Gamma \Vdash_\Sigma \Delta \vdash () : ()}\,(tpl : 0)^\tau \qquad \frac{\Gamma \Vdash_\Sigma \Delta \vdash t_1 : \tau_1 \ \ldots\ \Gamma \Vdash_\Sigma \Delta \vdash t_n : \tau_n}{\Gamma \Vdash_\Sigma \Delta \vdash (t_1,\ \ldots,\ t_n) : (\tau_1 \times \ldots \times \tau_n)}\,(tpl : n)^\tau$$

$$\frac{\Gamma \Vdash_\Sigma \Delta \vdash t : \sigma\,\theta \quad \Gamma \Vdash_\Sigma \Delta \vdash \tau : \mathsf{type}}{\Gamma \Vdash_\Sigma \Delta \vdash \mathsf{f}\, t : \tau}\,(app)^\tau\ \big(\Theta \Vdash \nabla \vdash \mathsf{f}\, s : \langle \sigma \hookrightarrow \rho \rangle \in \Sigma^{\prime\prime},\ \mathsf{sol}(\theta)\big)$$

$$\frac{\Gamma \bowtie (b : \sigma) \Vdash_\Sigma \Delta \cup \Delta^\# \vdash (a\ b) \cdot \tau : \mathsf{type}}{\Gamma \Vdash_\Sigma \Delta \vdash [a : \sigma]\, \tau : \mathsf{type}}\,(abt)^\tau$$

$$\frac{\mathsf{validE}(\Gamma,\ \Sigma,\ \Delta)}{\Gamma \Vdash_\Sigma \Delta \vdash () : \mathsf{type}}\,(prd : 0)^\tau \qquad \frac{\Gamma \Vdash_\Sigma \Delta \vdash \tau_1 : \mathsf{type}\ \ldots\ \Gamma \Vdash_\Sigma \Delta \vdash \tau_n : \mathsf{type}}{\Gamma \Vdash_\Sigma \Delta \vdash (\tau_1 \times \ldots \times \tau_n) : \mathsf{type}}\,(prd : n)^\tau$$

$$\frac{\Gamma \Vdash_\Sigma \Delta \vdash t\colon \sigma\,\theta}{\Gamma \Vdash_\Sigma \Delta \vdash \mathtt{C}\,t\colon \mathsf{type}}\,(cns)^\tau\ \big(\Theta \Vdash \nabla \vdash \mathtt{C}\,s\colon \langle \sigma \hookrightarrow \mathsf{type}\rangle \in \Sigma^{\prime\prime},\ \mathsf{sol}(\theta)\big)$$

$$\frac{\Gamma \Vdash_\Sigma \Delta \vdash t\colon \tau \quad \Delta \vdash \sigma \approx_\alpha \tau}{\Gamma \Vdash_\Sigma \Delta \vdash t\colon \sigma}\,(\alpha)^\tau$$

Note that in the rules, $(atm)^\tau$, $(var)^\tau$, $(tpl : 0)^\tau$ and $(prd : 0)^\tau$, the validity of the environment is needed as a premise because the environment, Γ, is not assumed to be valid and also, that in the rule, $(var)^\tau$, the suspension, $\vartheta\big|\pi$, must be applied to the type, Γ_X.

4 Meta-theory

The type system presented works uniformly on α-equivalence classes of terms and types (see Theorem 18 below.) In order to prove this property, some standard properties (weakening, strengthening and validity of typing environments) are first stated. The section is concluded with the substitution theorems.

▶ **Theorem 14** (Type Strengthening). *1. If* $\Gamma \Vdash_\Sigma \Delta\cup\Delta' \vdash t\colon \tau$ *(resp.* $\Gamma \Vdash_\Sigma \Delta\cup\Delta' \vdash \tau\colon \mathsf{type}$*) and* Δ' *mentions atoms that are not in* $\mathsf{dom}(\Gamma)$ *then* $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \tau$ *(resp.* $\Gamma \Vdash_\Sigma \Delta \vdash \tau\colon \mathsf{type}$*).*

2. If $\Gamma \bowtie (a_1\colon \sigma_1)\dots(a_n\colon \sigma_n) \Vdash_\Sigma \Delta \vdash t\colon \tau$ *(resp.* $\Gamma \bowtie (a_1\colon \sigma_1)\dots(a_n\colon \sigma_n) \Vdash_\Sigma \Delta \vdash \tau\colon \mathsf{type}$*) and* $\Delta \vdash a_i \# t,\ \tau$ $(1 \le i \le n)$ *then* $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \tau$ *(resp.* $\Gamma \Vdash_\Sigma \Delta \vdash \tau\colon \mathsf{type}$*).*

Proof. Both parts are proved by induction on the type derivation. ◀

▶ **Theorem 15** (Type Weakening).

1. *If* $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \tau$ *(resp.* $\Gamma \Vdash_\Sigma \Delta \vdash \tau\colon \mathsf{type}$*), then* $\Gamma' \Vdash_\Sigma \Delta' \vdash t\colon \tau$ *(resp.* $\Gamma' \Vdash_\Sigma \Delta' \vdash \tau\colon \mathsf{type}$*), for any* Γ', Δ' *such that* $\Gamma' = \Gamma \bowtie \Gamma_1$, $\Delta' \supseteq \Delta$ *and* $\mathsf{validE}(\Gamma',\ \Sigma,\ \Delta')$.
2. *If* $\Gamma \bowtie (a_1\colon \sigma_1) \bowtie (a_2\colon \sigma_2) \bowtie \Gamma' \Vdash_\Sigma \Delta \vdash t\colon \tau$ *(resp.* $\Gamma \bowtie (a_1\colon \sigma_1) \bowtie (a_2\colon \sigma_2) \bowtie \Gamma' \Vdash_\Sigma \Delta \vdash \tau\colon \mathsf{type}$*), then* $\Gamma \bowtie (a_2\colon \sigma_2) \bowtie (a_1\colon \sigma_1) \bowtie \Gamma' \Vdash_\Sigma \Delta \vdash t\colon \tau$ *(resp.* $\Gamma \bowtie (a_2\colon \sigma_2) \bowtie (a_1\colon \sigma_1) \bowtie \Gamma' \Vdash_\Sigma \Delta \vdash \tau\colon \mathsf{type}$ *provided* $\Delta \vdash a_1 \# \sigma_2$.

Proof. By simultaneous induction on the type derivation. ◀

▶ **Theorem 16** (Validity of Typing Environments). *For any given* $\mathsf{validD}(\Sigma)$*:*
If $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \tau$ *then* $\mathsf{validE}(\Gamma,\ \Sigma,\ \Delta)$ *and* $\Gamma \Vdash_\Sigma \Delta \vdash \tau\colon \mathsf{type}$.
If $\Gamma \Vdash_\Sigma \Delta \vdash \tau\colon \mathsf{type}$ *then* $\mathsf{validE}(\Gamma,\ \Sigma,\ \Delta)$.

Proof. Both parts are proved by induction on the type derivation, using weakening (in the case where the last rule used was $(atm)^\tau$) and strengthening in the cases where the last rule used is an abstraction rule (for types or terms). ◀

▶ **Lemma 17.** *If* $\Delta \vdash \mathsf{ds}(\vartheta_1\big|\pi_1,\ \vartheta_2\big|\pi_2) \# X$ *then* $\Delta \vdash \mathsf{ds}(\vartheta_1\big|\pi_1,\ \vartheta_2\big|\pi_2) \# \Gamma_X$.
For any M *(pseudo-term or pseudo-type), if* $\Delta \vdash \mathsf{ds}(\vartheta_1\big|\pi_1,\ \vartheta_2\big|\pi_2) \# M$, *then* $\Delta \vdash (\pi_1 \cdot M)\,\vartheta_1 \approx_\alpha (\pi_2 \cdot M)\,\vartheta_2$.

Proof. The first part is a consequence of the definition of valid environment. The second part is proved by induction on the definition of $\approx_\alpha$. ◀

▶ **Theorem 18** (Unicity of Types). *If* $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \tau_1$ *and* $\Gamma \Vdash_\Sigma \Delta \vdash t'\colon \tau_2$, *where* $\Delta \vdash t \approx_\alpha t'$, *then* $\Delta \vdash \tau_1 \approx_\alpha \tau_2$.

Proof. 1. If $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \tau_1$ and $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \tau_2$, then $\Delta \vdash \tau_1 \approx_\alpha \tau_2$.

2. If $\Gamma \Vdash_\Sigma \Delta \vdash t : \tau$, $\Delta \vdash t \approx_\alpha t'$, $\Delta \vdash \Gamma \approx_\alpha \Gamma'$ then $\Gamma' \Vdash_\Sigma \Delta \vdash t' : \tau$, where $\Delta \vdash \Gamma \approx_\alpha \Gamma'$ indicates that for each mapping $(a : \sigma)$ (resp. $(X : \sigma)$) in Γ, Γ' contains a mapping $(a : \sigma')$ (resp. $(X : \sigma')$) such that $\Delta \vdash \sigma \approx_\alpha \sigma'$. Similarly for types: if $\Gamma \Vdash_\Sigma \Delta \vdash \tau : \mathsf{type}$, $\Delta \vdash \tau \approx_\alpha \tau'$, $\Delta \vdash \Gamma \approx_\alpha \Gamma'$ then $\Gamma' \Vdash_\Sigma \Delta \vdash \tau' : \mathsf{type}$.

The first part is proved by induction on the type derivation for $\Gamma \Vdash_\Sigma \Delta \vdash t : \tau_1$. The only case that is not straightforward is the case in which the derivation concludes using the rule $(app)^\tau$. In this case, the unicity of the type is a consequence of the fact that declarations do not overlap (that is, there is a unique declaration that matches the term considered) and matching problems have unique most general solutions under the assumptions for declarations.

The second part is proved by induction on the type derivation. In the case of a variable, Lemma 17 is needed to derive $\mathsf{typV}(\Gamma',\ \Sigma,\ \Delta,\ \vartheta' \big| \pi',\ X)$. ◀

▶ **Theorem 19** (Preservation of Types by Atom Substitution). *If* $\Gamma \Vdash_\Sigma \Delta \vdash t : \tau$ *(resp.* $\Gamma \Vdash_\Sigma \Delta \vdash \tau : \mathsf{type}$*) and* π, ϑ, Γ', Δ' *are such that:*

- $\forall X \in \mathsf{dom}(\Gamma)$, $\Gamma' \Vdash_\Sigma \Delta' \vdash \vartheta \big| \pi \cdot X : (\pi \cdot \Gamma_X)\,\vartheta$
- $\forall a \in \mathsf{dom}(\Gamma)$, *either* $\Delta \vdash a \# t$ *or* $\Gamma' \Vdash_\Sigma \Delta' \vdash \vartheta(\pi(a)) : (\pi \cdot \Gamma_a)\,\vartheta$

then $\Gamma' \Vdash_\Sigma \Delta' \vdash (\pi \cdot t)\,\vartheta : (\pi \cdot \tau)\,\vartheta$ *(resp.* $\Gamma' \Vdash_\Sigma \Delta' \vdash (\pi \cdot \tau)\,\vartheta : \mathsf{type}$*)*

Proof. By induction on the type derivation. In the case where the last rule applied is $(app)^\tau$ or $(cns)^\tau$, one relies on the fact that declarations are closed (that is, there are no unabstracted atoms.) The cases of abstraction rules (for terms or types) follow by induction, since atom substitutions are capture-avoiding. ◀

▶ **Theorem 20** (Preservation of Types by Variable Substitution). *If* $\Gamma \Vdash_\Sigma \Delta \vdash t : \tau$ *(resp.* $\Gamma \Vdash_\Sigma \Delta \vdash \tau : \mathsf{type}$*) and* θ *is a variable substitution such that:*

- $\forall X \in \mathsf{dom}(\Gamma)$, $\Gamma' \Vdash_\Sigma \Delta' \vdash \theta(X) : \Gamma_X\,\theta$ *and* $\Delta' \vdash \Delta\,\theta$
- $\forall a \in \mathsf{dom}(\Gamma)$, *either* $\Delta \vdash a \# t$ *or* $\Gamma' \Vdash_\Sigma \Delta' \vdash a : \Gamma_a\,\theta$

then $\Gamma' \Vdash_\Sigma \Delta' \vdash t\,\theta : \tau\,\theta$ *(resp.* $\Gamma' \Vdash_\Sigma \Delta' \vdash \tau\,\theta : \mathsf{type}$*)*

Proof. By induction on the type derivation. In the case where the term is of the form $\theta \big| \pi \cdot X$ and the last rule applied is $(var)^\tau$ Theorem 19 for atom substitutions is used. ◀

5 Extended Example

First-order logic is a proto-typical system with binding. We consider the language of Arithmetic, and start the specification by defining type-formers for natural numbers, propositions and proofs, and term-formers to build numbers, $\mathsf{0}$ (zero) and s (successor) and propositions, bot ($\bot$), imp ($\Rightarrow$) and all ($\forall$).

$$\frac{}{- \Vdash \varnothing \vdash \mathtt{N} : \mathsf{type}}$$

$$\frac{}{- \Vdash \varnothing \vdash \mathtt{Prop} : \mathsf{type}}$$

$$\frac{P : \mathtt{Prop} \Vdash \varnothing \vdash \mathtt{Proof}\ P : \mathsf{type}}{}$$

$$\frac{}{- \Vdash \varnothing \vdash \mathsf{0} : \mathtt{N}}$$

$$\frac{X : \mathtt{N} \Vdash \varnothing \vdash \mathsf{s}\,X : \mathtt{N}}{}$$

$$\frac{}{- \Vdash \varnothing \vdash \mathsf{bot} : \mathtt{Prop}}$$

$$\frac{P_1 : \mathtt{Prop},\ P_2 : \mathtt{Prop} \Vdash \varnothing \vdash \mathsf{imp}\,(P_1,\ P_2) : \mathtt{Prop}}{}$$

$$\frac{P : \mathtt{Prop} \Vdash \varnothing \vdash \mathsf{all}\,[x : \mathtt{N}]\,P : \mathtt{Prop}}{}$$

Now, define declarations for the predicates used to build proofs; the introduction and elimination of imp and all, $\mathsf{imp_i}$, $\mathsf{imp_e}$, $\mathsf{all_i}$ and $\mathsf{all_e}$ and the elimination of bot, $\mathsf{bot_e}$.

$$\begin{array}{r} P_1 \colon \mathtt{Prop},\ P_2 \colon \mathtt{Prop},\ Q \colon \mathtt{Proof}\ P_2 \\ \Vdash x \# P_1,\ x \# P_2 \vdash \mathsf{imp_i}\,[x \colon \mathtt{Proof}\ P_1]\,Q \colon \mathtt{Proof}\ \mathsf{imp}\,(P_1,\ P_2) \end{array}$$

$$\begin{array}{r} P_1 \colon \mathtt{Prop},\ P_2 \colon \mathtt{Prop},\ Q \colon \mathtt{Proof}\ \mathsf{imp}\,(P_1,\ P_2),\ Q_1 \colon \mathtt{Proof}\ P_1 \\ \Vdash \varnothing \vdash \mathsf{imp_e}\,(Q,\ Q_1) \colon \mathtt{Proof}\ P_2 \end{array}$$

$$\begin{array}{r} P \colon \mathtt{Prop},\ Q \colon \mathtt{Proof}\ \mathsf{bot} \\ \Vdash \varnothing \vdash \mathsf{bot_e}\,(P,\ Q) \colon \mathtt{Proof}\ P \end{array}$$

$$\begin{array}{r} P \colon \mathtt{Prop},\ Q \colon [x \colon \mathtt{N}]\,\mathtt{Proof}\ P \\ \Vdash \varnothing \vdash \mathsf{all_i}\,Q \colon \mathtt{Proof}\ \mathsf{all}\,[x \colon \mathtt{N}]\,P \end{array}$$

$$\begin{array}{r} P \colon \mathtt{Prop},\ Q \colon \mathtt{Proof}\ \mathsf{all}\,[x \colon \mathtt{N}]\,P,\ N \colon \mathtt{N} \\ \Vdash x \# N \vdash \mathsf{all_e}\,(Q,\ N) \colon \mathtt{Proof}\ [x \mapsto N] \cdot P \end{array}$$

Note that in the declaration for $\mathsf{bot_e}$ one must use variables, P and Q, as arguments because of the restriction that all variables in the type of $\mathsf{bot_e}$ should occur in its arguments or in its types. Notice also that in the declaration for $\mathsf{all_i}$ above, the variable, Q, of type $[x \colon \mathtt{N}]\,\mathtt{Proof}\ P$, is used, in other words, n is not unabstracted, as expected. The full declaration, which can be inferred is $P \colon \mathtt{Prop},\ Q \colon [x \colon \mathtt{N}]\,\mathtt{Proof}\ P \Vdash \varnothing \vdash \mathsf{all_i}\,Q \colon \langle [x \colon \mathtt{N}]\,\mathtt{Proof}\ P \hookrightarrow \mathtt{Proof}\ \mathsf{all}\,([x \colon \mathtt{N}]\,P)\rangle$. When this declaration is used to type terms built with $\mathsf{all_i}$, pattern matching is used to obtain the values of P and Q.

An induction principle over the natural numbers could be defined as follows.

$$\begin{array}{r} P \colon \mathtt{Prop},\ Q_0 \colon \mathtt{Proof}\ [x \mapsto 0] \cdot P,\ Q_1 \colon [n \colon \mathtt{N}]\,[p \colon \mathtt{Proof}\ [x \mapsto n] \cdot P]\,\mathtt{Proof}\ [x \mapsto \mathtt{s}\,n] \cdot P \\ \Vdash n \# P \vdash \mathsf{ind}\,(Q_0,\ Q_1) \colon \mathtt{Proof}\ \mathsf{all}\,[x \colon \mathtt{N}]\,P \end{array}$$

An encoding of a system in a logical framework is adequate if it faithfully reflects the properties of the encoded system. For instance, in the case of an encoding of first-order logic, one needs to show that the terms used in the dependent type system represent first-order terms, that formulae and proofs correspond to their standardly acknowledged notions, and that only provable propositions have a proof in the system. The goal is to prove that there is a bijection between proofs in first-order logic and the corresponding terms in this system. A formal specification of first-order logic terms, formulae and proofs is given and then it is shown that these are encoded by terms of the correct type and that encoded terms represent only well-formed terms, formulae and proofs. The theorems and proofs presented here follow closely those given in [24] to prove the adequacy of LF but are much simpler due to the fact that here lambda calculus β-reduction is not involved and therefore all terms are of canonical form.

The following grammars define the syntax of the sets of terms (Trm) and formulae (Frm) of first-order logic.

▶ **Definition 21** (First-order Logic Terms and Formulae)**.**

$$\mathsf{T},\ \mathsf{T}' ::= 0 \mid s(\mathsf{T}) \mid x \qquad \mathsf{F},\ \mathsf{F}' ::= \bot \mid \mathsf{F} \to \mathsf{F}' \mid \forall x.\,\mathsf{F}$$

Let $\mathsf{fvars}(\mathsf{T})$ and $\mathsf{fvars}(\mathsf{F})$ denote respectively the set of free variables in the term, T, and the formula, F. Extend this notation element-wise to sets of formulae.

A translation function $[\![\mathsf{T}]\!]$ is defined by induction on T, from Trm to terms in the system using the term-formers, $\mathsf{0}$ and s, for which, see Section 5. Note that the free variables of T are encoded as unabstracted atoms in $[\![\mathsf{T}]\!]$. A corresponding translation function is defined over the elements of Frm.

$$\begin{array}{llll} [\![0]\!] & = \mathsf{0} & [\![\bot]\!] & = \mathsf{bot} \\ [\![s(\mathsf{T})]\!] & = \mathsf{s}\,[\![\mathsf{T}]\!] & [\![\mathsf{F}_1 \to \mathsf{F}_2]\!] & = \mathsf{imp}\,([\![\mathsf{F}_1]\!],\ [\![\mathsf{F}_2]\!]) \\ [\![x]\!] & = x & [\![\forall x.\,\mathsf{F}]\!] & = \mathsf{all}\,[x\colon \mathtt{N}]\,[\![\mathsf{F}]\!] \end{array}$$

Using the declarations given above, it can now be proved both that translated terms and formulae are typeable terms of the system and that typeable encoded terms correspond exactly to well-formed first-order logic terms and formulae.

▶ **Theorem 22.** *For any term,* $\mathsf{T} \in \mathsf{Trm}$*, such that* $\mathsf{fvars}(\mathsf{T}) = x_1,\ \ldots,\ x_n$ *and freshness context,* Δ*, if* $x_1\colon \mathtt{N},\ \ldots,\ x_n\colon \mathtt{N} \Vdash_\Sigma \Delta \vdash [\![\mathsf{T}]\!]\colon \mathtt{N}$*.*

Similarly for any $\mathsf{F} \in \mathsf{Frm}$*, such that* $\mathsf{fvars}(\mathsf{F}) = x_1,\ \ldots,\ x_n$*, and freshness context,* Δ*,* $x_1\colon \mathtt{N},\ \ldots,\ x_n\colon \mathtt{N} \Vdash_\Sigma \Delta \vdash [\![\mathsf{F}]\!]\colon \mathtt{Prop}$*,*

Proof. By structural induction on the syntax of elements of Trm and Frm. ◀

▶ **Theorem 23.** *If* $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \mathtt{N}$ *is a derivable typing judgement and the environment,* Γ*, contains only type associations for unabstracted atoms of the form* $(a\colon \mathtt{N})$*, then* $t \equiv [\![\mathsf{T}]\!]$ *for some term* $\mathsf{T} \in \mathsf{Trm}$*.*

Similarly, if $\Gamma \Vdash_\Sigma \Delta \vdash t\colon \mathtt{Prop}$ *is a derivable typing judgement and the environment,* Γ*, contains only type associations for unabstracted atoms of the form* $(a\colon \mathtt{N})$*, then* $t \equiv [\![\mathsf{F}]\!]$ *for some formula* $\mathsf{F} \in \mathsf{Frm}$*.*

Proof. By induction on typing judgement derivations. The only applicable cases are when the first step of the derivation is by one of the rules, (atm), (app) or (α). ◀

In order to show the adequacy of the encoding of proofs of first-order formulae, first, a natural deduction presentation is given for first-order logic, inspired by the one used in [24] to prove the adequacy of the encoding in LF.

Let judgements have the form $\mathcal{E} \vdash_\mathcal{V} \mathsf{P}\colon \mathsf{F}$, indicating that there is a proof P of the formula F, using the list of hypotheses, $\mathcal{E}$, and the set of free variables, $\mathcal{V}$, where $(\mathsf{fvars}(\mathcal{E}) \subseteq \mathcal{V})$. The introduction rules for implication and universal quantification are shown below.

$$\frac{\mathcal{E},\ (v_{\mathsf{F}_1}\colon \mathsf{F}_1) \vdash_\mathcal{V} \mathsf{P}\colon \mathsf{F}_2}{\mathcal{E} \vdash_\mathcal{V} \to_i (\mathsf{P} \setminus v_{\mathsf{F}_1})\colon \mathsf{F}_1 \to \mathsf{F}_2} \qquad \frac{\mathcal{E} \vdash_{\mathcal{V} \cup \{x\}} \mathsf{P}\colon \mathsf{F}}{\mathcal{E} \vdash_\mathcal{V} \forall_i(x.\,\mathsf{P})\colon \forall x.\,\mathsf{F}}$$

Here, v_{F_1} is the variable name of a proof of F_1, the notation $(\mathsf{P} \setminus v_{\mathsf{F}_1})$ denotes the proof P where v_{F_1} is discharged, and in the rule for $\forall_i$, the condition $\mathsf{fvars}(\mathcal{E}) \subseteq \mathcal{V}$ on judgements implies that x is not used in $\mathcal{E}$.

A natural deduction judgement, J, of the form $\mathcal{E} \vdash_\mathcal{V} \mathsf{P}\colon \mathsf{F}$ is translated to a typing judgement of the system, $[\![\mathsf{J}]\!]$, as follows.

$$[\![\mathcal{E} \vdash_\mathcal{V} \mathsf{P}\colon \mathsf{F}]\!] = [\![\mathcal{V}]\!] \bowtie [\![\mathcal{E}]\!] \Vdash_\Sigma [\![\mathsf{P}]\!]^{\Delta}_{\mathsf{vars}([\![\mathsf{J}]\!])} \vdash [\![\mathsf{P}]\!]\colon \mathtt{Proof}\ [\![\mathsf{F}]\!]$$

Here, if $\mathcal{V}$ is $\{x_1,\ \ldots,\ x_n\}$ then $[\![\mathcal{V}]\!] = x_1\colon \mathtt{N},\ \ldots,\ x_n\colon \mathtt{N}$ and $[\![\mathcal{E}]\!]$ contains $v_{\mathsf{F}_i}\colon \mathtt{Proof}\ [\![\mathsf{F}_i]\!]$ for each $(v_{\mathsf{F}_i}\colon \mathsf{F}_i)$ in Γ. The translation function from proofs to terms, $[\![\mathsf{P}]\!]$, is defined inductively; two cases are given.

$$[\![\forall_i(x.\,\mathsf{P})]\!] = \mathsf{all}_\mathsf{i}\,[x\colon \mathtt{N}]\,[\![\mathsf{P}]\!] \quad [\![\to_i (\mathsf{P} \setminus v_{\mathsf{F}_1})]\!] = \mathsf{imp}_\mathsf{i}\,[v_{\mathsf{F}_1}\colon \mathtt{Proof}\ [\![\mathsf{F}_1]\!]]\,[\![\mathsf{P}]\!]$$

A second translation function from a proof, P, to a freshness context, parameterised by a set of variables, $\mathcal{X}$, and written $[\![\mathsf{P}]\!]^{\Delta}_{\mathcal{X}}$ is also required; again two cases are given.

$$[\![\forall_i(x.\,\mathsf{P})]\!]^{\Delta}_{\mathcal{X}} = [\![P]\!]^{\Delta}_{\mathcal{X}} \quad [\![\rightarrow_i (\mathsf{P} \setminus v_{\mathsf{F}_1})]\!]^{\Delta}_{\mathcal{X}} = \{v_{\mathsf{F}_1} \# X \mid X \in \mathcal{X}\} \cup [\![P]\!]^{\Delta}_{\mathcal{X}}$$

One can now prove the following property relating natural deduction proofs in first-order logic and their encoding in the system presented here.

▶ **Theorem 24.** *If* $\mathcal{E} \vdash_{\mathcal{V}} \mathsf{P} : \mathsf{F}$ *is a derivable judgement,* J*, of natural deduction then* $[\![\mathcal{V}]\!] \bowtie [\![\mathcal{E}]\!] \Vdash_{\Sigma} [\![\mathsf{P}]\!]^{\Delta}_{\mathsf{vars}([\![\mathsf{J}]\!])} \vdash [\![\mathsf{P}]\!] : \mathtt{Proof}\,[\![\mathsf{F}]\!]$ *is a derivable typing judgement.*

Proof. By induction on the syntax of the proof, P. In the case where P is of the form $\rightarrow_i (\mathsf{P}' \setminus v_{\mathsf{F}_1})$ of these, the use of the rule (α) follows from the fact that $[\![\mathsf{P}]\!]^{\Delta}_{\mathsf{vars}([\![\mathsf{J}]\!])} \vdash v_{\mathsf{F}_1} \# [\![\mathsf{F}_1]\!],\ v_{\mathsf{F}_1} \# [\![\mathsf{F}_2]\!]$. ◀

Finally, in order to complete the adequacy proof, it is shown that only provable first-order formulae are encoded by terms of type $\mathtt{Proof}\,t$.

▶ **Theorem 25.** *If* $\Gamma \Vdash_{\Sigma} \Delta \vdash s : \mathtt{Proof}\,t$ *where the environment,* Γ*, contains only type associations for atoms either of the form* $(a : \mathtt{N})$ *or* $(v_{\mathsf{F}_1} : \mathtt{Proof}\,t_i)$ *where* $t_i \equiv [\![\mathsf{F}_i]\!]$ *is an encoding of some formula* F_i*, then* $s \equiv [\![\mathsf{P}]\!]$ *where* P *is a proof by natural deduction of some formula* F *such that* $t \equiv [\![\mathsf{F}]\!]$.

Proof. By induction on typing judgement derivations. In the derivation for $\mathsf{imp_i}$, $\Delta \vdash v_{\mathsf{F}_1} \# t_1,\ v_{\mathsf{F}_1} \# t_2,\ v_{\mathsf{F}_1} \# \mathsf{imp_i}\,[v_{\mathsf{F}_1} : \mathtt{Proof}\,t_1]\,s'$ and so v_{F_1} cannot be a free variable of P or F. ◀

6 Related Work

Nominal sets have been used to give semantics to systems based on nominal abstract syntax (see, for instance, [31, 18, 7, 11]) and proof theories for nominal logic have also been considered [19, 6]). Atom substitutions and their properties have been defined as systems of equational rules in [17, 21, 15]. Nominal equational theories have been investigated in [22, 10, 15] amongst others, and type systems for nominal terms and equational theories, using rank-1 polymorphic types, are defined in [14, 13, 12]. However, although proofs play an important role in all of these works, none of these systems deal explicitly with proof terms, and do not yield directly a nominal type theory. Nominal type theory has been investigated by Schöpp and Stark [34], using categorical models of nominal logic. The nominal dependent type theories developed following this approach are very expressive, but it is not clear whether their computational properties make them suitable for use in a logical framework. Nominal type theory as a basis for logical frameworks has been investigated by Cheney [8, 9], as extensions of a typed λ-calculus with names, name-abstraction and concretion operators, and name-abstraction types. A system combining λ-calculus and nominal features is also investigated by Pitts [32] to define a nominal version of Gödel's System T. A key difficulty encountered when following the approach of combining λ-calculus and nominal syntax is the interaction between name abstraction and functional abstraction (see [8] for a detailed discussion.) Westbrook [36] extends the Calculus of Inductive Constructions with a name-abstraction construct in the style of [8].

One of the best known examples of logical frameworks is LF [24], based on a typed λ-calculus with dependent types. The system presented here has similar expressive power, however there is no primitive notion of functional abstraction, instead there are term- and type- constructors in the user-defined signature. Other differences with LF include the

distinction between atoms (which can be abstracted or unabstracted), variables (which cannot be abstracted but can be instantiated, with non-capture-avoiding substitutions), and the use of name swappings (or more generally, permutations), to axiomatise α-equivalence.

Compared with previous approaches to the definition of λ-free logical frameworks [26, 1, 33], abstraction is a first-class ingredient in the syntax presented here and can be used in arguments for term- or type-constructors (as in [26, 1, 33]), or on its own (unlike [26, 1]) although user-defined constructors of abstraction type are not allowed. Also, as in [26, 1, 33], instantiation is a primitive notion in the system; it is used instead of the application operation used in λ-calculus based logical frameworks. However the approach here does not rely on explicit lists of arities and η-long normal forms as in [26, 1]. The triggering of suspended atom substitutions by instantiating variables is similar to the hereditary substitution mechanism of DMBEL [33].

There may exist similarities between the work here and that of 'contextual modal type theory' [28], and the dependent type system of this paper may benefit from a study of the handling of type environments and substitutions therein.

7 Conclusions and Future Work

This paper has presented a dependent type system for nominal terms with atom substitutions. A definition of matching over this syntax have been given together with an algorithm for solving such problems. This algorithm has been implemented but the complexity of problems has not been analysed; this is left for future work. A set of axioms and rules was then defined for determining the typeability of pseudo-terms and pseudo-types in this system in the presence of user-defined declarations for term-formers and type-formers. An extended example for first-order logic was presented and its adequacy proven. The type system itself has not been implemented. In its present form, the inclusion of the rule $(\alpha)^\tau$ means that the inference of derivations is not completely syntax-directed. This property, that derivable typing judgements hold for alpha-equivalent classes of types, may be derivable and if so should help in the development of a type inference algorithm for the system. Further benefits may also be gained from the inclusion of some of the more sophisticated ideas used for other type systems considered in [12]. Although we have not included computation rules in our language, dynamic features, such as reduction in the λ-calculus, or proof normalisation for a logic, may be represented using relations between terms. However, a more direct definition using equality axioms, such as in [26], would be easier to use. An extension of the logical framework to include a user-defined nominal equational theory, specified as a set of equality axioms or rewriting rules, also remains as future work.

References

1 Robin Adams. Lambda-free logical frameworks. *CoRR*, abs/0804.1879, 2008.

2 Stuart F. Allen, Robert L. Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. The Nuprl open logical environment. In *Automated Deduction – CADE-17*, 2000.

3 Franz Baader and Wayne Snyder. Unification theory. In *Handbook of Automated Reasoning*. Elsevier, 2001.

4 Christophe Calvès and Maribel Fernández. Matching and alpha-equivalence check for nominal terms. *Journal of Computer System Sciences*, 76, 2010.

5 Christophe Calvès and Maribel Fernández. The first-order nominal link. In *Logic-Based Program Synthesis and Transformation – 20th International Symposium, LOPSTR 2010,*

Hagenberg, Austria, July 23-25, 2010, Revised Selected Papers, volume 6564 of *Lecture Notes in Computer Science*, pages 234–248. Springer, 2011.

6 James Cheney. A simpler proof theory for nominal logic. In *FoSSaCS*, 2005.

7 James Cheney. Completeness and Herbrand theorems for nominal logic. *Journal of Symbolic Logic*, 71, 2006.

8 James Cheney. A simple nominal type theory. *Electronic Notes in Theoretical Computer Science*, 228, 2009.

9 James Cheney. A dependent nominal type theory. *Logical Methods in Computer Science*, 8, 2012.

10 Ranald A. Clouston. *Equational Logic for Names and Binding*. PhD thesis, University of Cambridge, 2010.

11 Roy L. Crole and Frank Nebel. Nominal lambda calculus: An internal language for fm-cartesian closed categories. *Electr. Notes Theor. Comput. Sci.*, 298:93–117, 2013.

12 Elliot Fairweather. *Type Systems for Nominal Terms*. PhD thesis, King's College London, 2014.

13 Elliot Fairweather, Maribel Fernández, and Murdoch J. Gabbay. Principal types for nominal theories. In *Proceedings of the 18th International Symposium on Fundamentals of Computation Theory (FCT 2011)*, 2011.

14 Maribel Fernández and Murdoch J. Gabbay. Curry-style types for nominal terms. In *Types for Proofs and Programs (TYPES'06)*. Springer, 2007.

15 Maribel Fernández and Murdoch J. Gabbay. Nominal rewriting. *Information and Computation*, 205, 2007.

16 Maribel Fernández and Murdoch J. Gabbay. Closed nominal rewriting and efficiently computable nominal algebra equality. In *Proceedings of the 5th International Workshop on Logical Frameworks and Meta-Languages (LFMTP 2010)*, 2010.

17 Murdoch J. Gabbay. A study of substitution, using nominal techniques and Fraenkel-Mostowski sets. *Theoretical Computer Science*, 410, 2009.

18 Murdoch J. Gabbay. Two-level nominal sets and semantic nominal terms: an extension of nominal set theory for handling meta-variables. *Mathematical Structures in Computer Science*, 21, 2011.

19 Murdoch J. Gabbay and James Cheney. A sequent calculus for nominal logic. In *Proceedings of the 19th IEEE Symposium on Logic in Computer Science (LICS 2004)*, 2004.

20 Murdoch J. Gabbay and Aad Mathijssen. Nominal algebra. In *18th Nordic Workshop on Programming Theory*, 2006.

21 Murdoch J. Gabbay and Aad Mathijssen. Capture-avoiding substitution as a nominal algebra. *Formal Aspects of Computing*, 20, 2008.

22 Murdoch J. Gabbay and Aad Mathijssen. Nominal universal algebra: Equational logic with names and binding. *Journal of Logic and Computation*, 19, 2009.

23 Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax involving binders. In *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS 1999)*, pages 214–224. IEEE Computer Society Press, July 1999.

24 Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science (LICS 1987)*. IEEE Computer Society Press, 1987.

25 Jordi Levy and Mateu Villaret. An efficient nominal unification algorithm. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA 2010)*, 2010.

26 Zhaohui Luo. PAL$^+$: a lambda-free logical framework. *Journal of Functional Programming*, 13, 2003.

27 Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In *Types for Proofs and Programs, (TYPES'93)*, 1994.

28 Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computational Logic*, 9, 2008.

29 Bengt Nordström, Kent Petersson, and Jan Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

30 Frank Pfenning and Carsten Schürmann. System description: Twelf – a meta-logical framework for deductive systems. In *Automated Deduction – CADE-16*, 1999.

31 Andrew M. Pitts. Nominal logic: A first order theory of names and binding. In *Proceedings of the 4th International Symposium on Theoretical Aspects of Computer Software (STACS 2001)*, 2001.

32 Andrew M. Pitts. Nominal system T. In *Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 2010)*, 2010.

33 Gordon Plotkin. An algebraic framework for logics and type theories, 2006. Talk given at LFMTP'06.

34 Ulrich Schöpp and Ian Stark. A Dependent Type Theory with Names and Binding. In *CSL*, 2004.

35 Christian Urban, Andrew M. Pitts, and Murdoch J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323, 2004.

36 Edwin Westbrook. *Higher-order Encodings with Constructors*. PhD thesis, Washington University in St. Louis, 2008.