Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science 228 (2009) 37–52
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A Simple Nominal Type Theory

James Cheney[1]

University of Edinburgh
11 Crichton Street
Edinburgh EH8 9LE, United Kingdom

Abstract

Nominal logic is an extension of first-order logic with features useful for reasoning about abstract syntax with bound names. For computational applications such as programming and formal reasoning, it is desirable to develop constructive type theories for nominal logic that extend standard type theories for propositional, first- or higher-order logic. This has proven difficult, largely because of complex interactions between nominal logic's name-abstraction operation and ordinary functional abstraction. This difficulty already arises in the case of propositional logic and simple type theory. In this paper we show how this difficulty can be overcome, and present a *simple nominal type theory* that enjoys properties such as type soundness and strong normalization, and that can be soundly interpreted using existing *nominal set* models of nominal logic. We also sketch how recursion combinators for languages with binding structure can be provided. This is a first step towards understanding the constructive content of nominal logic and incorporating it into existing constructive logics and type theories.

Keywords: nominal terms, type systems

1 Introduction

Nominal logic [14] is a variant of first-order logic that axiomatizes name-binding and alpha-equivalence using *permutative renamings*, or swappings. So far, nominal logic has been studied primarily using classical model theory [3,14] or proof theory [2,6] to formalize explicit reasoning about equational and freshness properties. While this analysis lends itself to implementations within theorem proving systems based on classical logic [25], nominal logic has resisted incorporation into constructive systems based on typed lambda-calculi. This is unfortunate, because nominal logic seems promising as a foundation for inductive reasoning about name-binding in a variety of settings, and constructive logics have many advantages due to the propositions-as-types and proofs-as programs principles: a constructive proof of a formula can be viewed as a program for performing a computation of the corresponding type. In particular, we would like a type theory for nominal logic to

[1] Email: jcheney@inf.ed.ac.uk

doi:10.1016/j.entcs.2008.12.115

internalize the *freshness reasoning* about name-abstractions that currently needs to be performed explicitly to justify recursion and induction for nominal abstract syntax (for example, in [9,15,23]).

Although proof theories for intuitionistic nominal logic has been considered already in work by Gabbay and Cheney [6,2,5], those systems consider only provability, not proof terms. The proof trees available in such systems could themselves be viewed as proof terms, but doing so does not immediately yield a well-behaved type theory with nice properties. Moreover, both systems involve significant amounts of explicit reasoning about equality and freshness.

Nominal type theory has also been investigated by Schöpp and Stark [19,22], who have developed a family of type theories based on categorical models of nominal logic. Their approach to nominal dependent type theory is very expressive, but motivated primarily by semantics. Computationally important syntactic properties such as strong normalization and the decidability of typechecking have not been studied, and seem difficult to obtain due to the complexity of the system. Thus, the problem of identifying type theoretic presentations of nominal logic with computational properties that make them suitable for use in automated reasoning and programming remains open.

In this paper, we consider a simple case of this problem. Specifically, we introduce an extension of the simply-typed lambda calculus that incorporates *names* and *name-abstraction* types $\langle\alpha\rangle A$ with introduction form $\langle a{:}\alpha\rangle M$ (*abstraction*) and elimination form $M \mathbin{@} a$ (*concretion*). Moreover, our approach can also be extended with recursion combinators for object languages in a syntactically and semantically sound way.

A key difficulty encountered in this approach is the interaction of name-abstraction with ordinary functional (λ-) abstraction. Specifically, at a semantic level, the name-concretion operation requires that the name passed as an argument to an abstraction is not already free in the body of the abstraction. Thus, the term $M_1 = (\langle a{:}\alpha\rangle(a,b))@b$ is ill-defined — it applies the partial function $M_2 = \langle a{:}\alpha\rangle(a,b)$ to a value not in its domain. Similarly, the term $M_3 = ((\lambda x{:}\alpha.\langle a{:}\alpha\rangle(a,x))\ b)@b$ is ill-defined, since it β-reduces to M_1. And more subtly, the term $M_4 = (\langle a{:}\alpha\rangle\lambda x.x@a)$ is also ill-defined, since $(M_4@b)\ (\langle a{:}\alpha\rangle(a,b))$ also reduces to M_1.

Previous systems have dealt with this by requiring explicit reasoning about swapping, freshness and equality (cf. [2,6,9,15,19,22,23,25]). However, as pointed out by a number of authors, the name-abstraction type constructor $\langle\alpha\rangle A$ can be interpreted in two isomorphic ways: as the *quotient* of the set of pairs, with respect to an α-equivalence relation $(\alpha \times A)/_{\equiv_\alpha}$, or as a (partial) *function space*, $\alpha \multimap A$, consisting of functions that may only be applied to a "fresh" name, similar to the "magic wand" connective/type constructor in the logic of bunched implications (BI) and its type theories [11]. Schöpp and Stark's type theory takes the approach that name-abstractions have the introduction and elimination forms of *both* quotiented-pair and partial-function presentations, using BI-style bunched contexts. (Their system also considers dependent product and sum types, which add further complications.) In this paper, we consider the consequences of designing a "simple" nominal type theory with name-abstractions corresponding only to the partial-function presentation, using a simpler form of bunched contexts that is specialized to this situation.

The introduction form of the name-abstraction type is then a name-abstraction term $\langle a{:}\alpha\rangle M$, where a is *bound* in M (in the same sense as x is bound in M in a λ-abstraction $\lambda x{:}A.M$). The elimination form is the "concretion" operation $M \mathbin{@} a$, familiar from some versions of FreshML. This leads naturally to typing rules:

$$\frac{\Gamma\#a{:}\alpha \vdash M : A \quad (a \notin \Gamma)}{\Gamma \vdash \langle a{:}\alpha\rangle M : \langle\alpha\rangle A} \quad \frac{\Gamma \vdash M : \langle\alpha\rangle A \quad \Gamma \vdash a : \alpha}{\Gamma \vdash M \mathbin{@} a : B} \ (*)$$

However, the second typing rule $(*)$ fails to express the freshness constraint on concretions; it permits all of the ill-defined terms M_1–M_4 above.

In order to obtain a type theory that defines only expressions that make sense in the universe of nominal sets, the typing rule for concretions $M \mathbin{@} a$ must ensure that a cannot appear in M, no matter how M is instantiated. The basic idea is to use a context with additional structure expressing freshness information, as previously explored in [2] for full nominal logic and in [22,19] in more generality. We consider contexts Γ that may have ordinary variable bindings $\Gamma, x{:}A$, where A is an arbitrary type, as well as "fresh name bindings" $\Gamma\#a{:}\alpha$, where α is a base type of names. When we typecheck a concretion $M \mathbin{@} a$, we must be able to use the information in the context to prove that a is fresh for all of the symbols present in M. This leads to a rule

$$\frac{\Gamma \vdash a : \alpha \setminus \Gamma' \quad \Gamma' \vdash M : \langle\alpha\rangle A}{\Gamma \vdash M \mathbin{@} a : B} \ (**)$$

This rule uses an auxiliary judgment $\Gamma \vdash a : \alpha \setminus \Gamma'$ which, intuitively, removes $a{:}\alpha$ from Γ to produce Γ', *and also removes all variables that could be substituted with something containing a.* The abstraction subterm M is typechecked with respect to this diminished context Γ'. This ensures that the concretion is well-defined.

The main contributions of this paper are as follows. We introduce (Section 2) a *simple nominal type theory* (here abbreviated SNTT) based on the (**) rule, and prove type soundness (Section 2.1) and strong normalization (Section 2.2) for SNTT. We also (Section 2.3) relate SNTT to nominal logic by showing how to interpret it using nominal sets. In Section 3 we show how SNTT can be extended with conditionals, name-equality, and primitive recursion combinators which can be used to define functions such as capture-avoiding substitution. We conclude in Sections 4 and 5 by relating SNTT to other systems and discussing future directions.

2 Simple nominal type theory

The basic syntactic classes of SNTT include countable, disjoint sets of *variables* $\mathbb{V} = \{x, y, z, \ldots\}$ and *names* (or *atoms*) $\mathbb{A} = \{a, b, c, \ldots\}$; atomic data-type symbols $\delta, \delta', \ldots$ and name-type symbols $\alpha, \alpha', \ldots$; and constant symbols $\mathbf{c}, \mathbf{d}, \ldots$. Additional syntactic classes include types A, B, terms M, N, contexts Γ, and substitutions θ, whose abstract syntax is described by the following grammar rules:

$$\begin{array}{rclrcl}
FVN(()) & = & FVN(\mathbf{c}) = \varnothing & FVN(z) & = & \{z\} \quad (z \in \mathbb{V} \cup \mathbb{A}) \\
FVN(\lambda x.M) & = & FVN(M) - \{x\} & FVN(\pi_i(M)) = FVN(M \mathbin{@} a) & = & FVN(M) \\
FVN(\langle a{:}\alpha\rangle M) & = & FVN(M) - \{a\} & FVN(M,N) = FVN(M\ N) & = & FVN(M) \cup FVN(N) \\
FV(M) & = & FVN(M) \cap \mathbb{V} & FN(M) & = & FVN(M) \cap \mathbb{A}
\end{array}$$

$$FVN(\cdot) = \varnothing \qquad FVN(\theta, M/x) = FVN(\theta) \cup FVN(M) \qquad FVN(\theta, b/a) = FVN(\theta) \cup \{b\}$$

Fig. 1. Free-variables and free-names functions

$$\begin{array}{rclrcl}
x[\theta] & = & \theta(x) & a[\theta] & = & a \\
\mathbf{c}[\theta] & = & \mathbf{c} & ()[\theta] & = & () \\
\pi_i(M)[\theta] & = & \pi_i(M[\theta]) & (M, M')[\theta] & = & (M[\theta], M'[\theta]) \\
(M\ M')[\theta] & = & M[\theta]\ M'[\theta] & (\lambda y{:}A.M)[\theta] & = & \lambda y{:}A.M[\theta, y/y] \ \ (y \notin FV(\theta)) \\
(M \mathbin{@} a)[\theta] & = & M[\theta - a] \mathbin{@} \theta(a) & (\langle a{:}\alpha\rangle M)[\theta] & = & \langle a{:}\alpha\rangle M[\theta, a/a] \ \ (a \notin FN(\theta))
\end{array}$$

Fig. 2. Capture-avoiding substitution/renaming

$$\begin{aligned}
A, B &::= \mathbf{1} \mid A \times B \mid A \to B \mid \langle\alpha\rangle A \mid \alpha \mid \delta \\
M, N &::= \mathbf{c} \mid x \mid () \mid (M, N) \mid \pi_i(M) \mid \lambda x{:}A.M \mid M\ N \mid a \mid \langle a{:}\alpha\rangle M \mid M \mathbin{@} a \\
\Gamma &::= \cdot \mid \Gamma, x{:}A \mid \Gamma\#a{:}\alpha \qquad \theta ::= \cdot \mid \theta, M/x \mid \theta, b/a
\end{aligned}$$

Many of the types (units, pairing and function types) and their associated introduction and elimination forms are standard. Names a are always of some name type α; the abstraction type $\langle\alpha\rangle A$ is associated with introduction form $\langle a{:}\alpha\rangle M$, called *name-abstraction*, and elimination form $M \mathbin{@} a$, called *name-concretion*.

We define the sets of free names $FN(M)$ and free variables $FV(M)$ of a term in Figure 1, treating a as binding in the abstraction operation $\langle a{:}\alpha\rangle M$ and x as binding in $\lambda x{:}A.M$. We also extend FV and FN to substitutions. Order matters in contexts. We consider terms equivalent modulo consistent renaming of bound names and variables; also, by convention, we write contexts as $x{:}A$ or $a{:}\alpha$ instead of $\cdot, x{:}A$ or $\cdot\#a{:}\alpha$, respectively. We write $M[\theta]$ for the result of applying a substitution θ to M; this is defined in Figure 2. We give the full definitions to eliminate any risk of confusion: in particular, observe that renamings and FN are *syntactic* operations only; they should not be confused with the concepts of swapping and support in nominal sets, discussed in Section 2.3.

We assume a fixed signature $\Sigma = \{\mathbf{c} : A, \ldots\}$ assigning unique types to the constants of the language. The term well-formedness rules for SNTT are shown in Figure 4. The *restriction* judgment $\Gamma \vdash a : \alpha \setminus \Gamma'$, defined in Figure 3, intuitively means that $a{:}\alpha$ appears in Γ and Γ' is the result of removing all bindings from Γ whose values may depend on a. Note the differences between the second and third rules. We write $a{:}\alpha \in \Gamma$ or $x{:}A \in \Gamma$ to say that a binding for a or x with the given type is present in Γ and write $a \notin \Gamma$ or $x \notin \Gamma$ to say that no such binding for a or x is present respectively. We assume all signatures and contexts are valid, that is, contain no duplicate variable or name bindings.

The rewriting rules for reducing and expanding the terms of SNTT include the

$$\frac{}{\Gamma\#a{:}\alpha \vdash a : \alpha \setminus \Gamma} \qquad \frac{(a \neq b) \quad \Gamma \vdash a : \alpha \setminus \Gamma'}{\Gamma\#b{:}\beta \vdash a : \alpha \setminus \Gamma'\#b{:}\beta} \qquad \frac{\Gamma \vdash a : \alpha \setminus \Gamma'}{\Gamma, x{:}A \vdash a : \alpha \setminus \Gamma'}$$

Fig. 3. Context restriction judgment

$$\frac{\mathbf{c} : A \in \Sigma}{\Gamma \vdash \mathbf{c} : A}\,\mathsf{con} \qquad \frac{}{\Gamma \vdash () : \mathbf{1}}\,\mathsf{unitI} \qquad \frac{\Gamma \vdash M_1 : A_1 \quad \Gamma \vdash M_2 : A_2}{\Gamma \vdash (M_1, M_2) : A_1 \times A_2}\,\wedge\mathsf{I} \qquad \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \pi_i(M) : A_i}\,\wedge\mathsf{E}$$

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\,\mathsf{var} \qquad \frac{\Gamma, x{:}A \vdash M : B \quad (x \notin \Gamma)}{\Gamma \vdash \lambda x{:}A.M : A \to B}\,\Rightarrow\mathsf{I} \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\,\Rightarrow\mathsf{E}$$

$$\frac{a{:}\alpha \in \Gamma}{\Gamma \vdash a : \alpha}\,\mathsf{name} \qquad \frac{\Gamma\#a{:}\alpha \vdash M : A \quad (a \notin \Gamma)}{\Gamma \vdash \langle a{:}\alpha\rangle M : \langle\alpha\rangle A}\,\mathsf{absI} \qquad \frac{\Gamma \vdash a : \alpha \setminus \Gamma' \quad \Gamma' \vdash M : \langle\alpha\rangle A}{\Gamma \vdash M \mathbin{@} a : A}\,\mathsf{absE}$$

Fig. 4. Simple nominal type theory: well-formedness

following rules, most of which are standard:

$$\begin{array}{rcl@{\qquad}rcll}
\pi_i(M_1, M_2) & \rhd_\beta & M_i & M : A_1 \times A_2 & \rhd_\eta & (\pi_1(M), \pi_2(M)) & \\
(\lambda x.M)\ N & \rhd_\beta & M[N/x] & M : A \to B & \rhd_\eta & \lambda x{:}A.M\ x & (x \notin FV(M)) \\
(\langle a{:}\alpha\rangle M) \mathbin{@} b & \rhd_\beta & M[b/a] & M : \langle\alpha\rangle B & \rhd_\eta & \langle a{:}\alpha\rangle M \mathbin{@} a & (a \notin FN(M)) \\
 & & & M : \mathbf{1} & \rhd_\eta & () &
\end{array}$$

We write $\longrightarrow_\beta$, $\longrightarrow_\eta$ for the rewriting relations generated by $\rhd_\beta, \rhd_\eta$ and write $M \longleftrightarrow^*_{\beta\eta} N$ to indicate that M and N are $\beta\eta$-convertible to a common form. We write $M \Downarrow_\beta N$ to indicate that M converges to normal form N under β-reduction and $M \Downarrow_\beta$ for $\exists N.M \Downarrow_\beta N$.

Besides arbitrary signatures involving higher-order constants, we also consider *nominal signatures* (following [15,24]). These are signatures in which all constants $\mathbf{c}$ have types $A \to \delta$, where A is first-order (i.e., does not use $\to$) and δ is a data type. We also define a sublanguage of the terms of SNTT called *ground nominal terms*, which are essentially the same as the ground nominal terms of [15,24]:

$$M_0, N_0 ::= () \mid \mathbf{c}\ M_0 \mid (M_0, N_0) \mid \langle a{:}\alpha\rangle M_0 \mid a$$

SNTT includes all of the rules of simply-typed lambda calculus with unit and product types (which we call simple type theory or STT), so any well-typed pure λ-term can be typechecked in SNTT with the same type. Also any well-typed ground nominal term can be typechecked in SNTT with respect to its nominal signature and a context consisting only of names. We will later show (Corollary 2.13) that SNTT is a conservative extension of both systems.

Examples. Before proceeding to the formal results, we show some illustrative examples (Figure 5) of well- and ill-formed terms involving both name-abstraction and λ-abstraction. Example (a) illustrates a term whose type corresponds to a "weakening" law $A \to \langle\alpha\rangle A$ for name-abstraction types. Example (b) shows how to typecheck the term $\lambda x{:}\langle\alpha\rangle A.\langle a{:}\alpha\rangle x@a$, which is the fully η-expanded identity function at type $\langle\alpha\rangle A$. Examples (c) and (d) provide partial derivations of non-typeable terms; in (d), there is no name that can be filled in for ?? that will make the term typecheck. Examples (e)–(j) shows additional properties of the name-abstraction type, omitting derivations. Example (e) gives a term that proves a "name-exchange" law; this is one direction of an isomorphism of types $\langle\alpha\rangle\langle\alpha'\rangle A$

$$\text{(a)}\quad \cfrac{\cfrac{\cfrac{}{x{:}A\#a{:}\alpha \vdash x : A}}{x{:}A \vdash \langle a{:}\alpha\rangle x : \langle\alpha\rangle A}}{\cdot \vdash \lambda x{:}A.\langle a{:}\alpha\rangle x : A \to \langle\alpha\rangle A} \qquad \text{(b)}\quad \cfrac{\cfrac{\cfrac{\cfrac{}{x{:}\langle\alpha\rangle A\#a{:}\alpha \vdash a : \alpha \setminus x{:}\langle\alpha\rangle A} \quad \cfrac{}{x{:}\langle\alpha\rangle A \vdash x : \langle\alpha\rangle A}}{x{:}\langle\alpha\rangle A\#a{:}\alpha \vdash x@a : A}}{x{:}\langle\alpha\rangle A \vdash \langle a{:}\alpha\rangle x@a : \langle\alpha\rangle A}}{\cdot \vdash \lambda x{:}\langle\alpha\rangle A.\langle a{:}\alpha\rangle x@a : \langle\alpha\rangle A \to \langle\alpha\rangle A}$$

$$\text{(c)}\quad \cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{a{:}\alpha \vdash a : \alpha \setminus \cdot}}{a{:}\alpha, x{:}\langle\alpha\rangle A \vdash a : \alpha \setminus \cdot} \quad \begin{array}{c}\text{stuck}\\ \cdot \vdash x : \langle\alpha\rangle A\end{array}}{a{:}\alpha, x{:}\langle\alpha\rangle A \vdash x \,@\, a : A}}{a{:}\alpha \vdash \lambda x{:}\langle\alpha\rangle A.x \,@\, a : \langle\alpha\rangle A \to A}}{\vdash \langle a{:}\alpha\rangle\lambda x{:}\langle\alpha\rangle A.x \,@\, a : \langle\alpha\rangle(\langle\alpha\rangle A \to A)} \qquad \text{(d)}\quad \begin{array}{c}\text{stuck since no name } a{:}\alpha \text{ is known}\\ \cfrac{x{:}\langle\alpha\rangle A \vdash x \,@??: A}{\cdot \vdash \lambda x.x \,@?? : \langle\alpha\rangle A \to A}\end{array}$$

$$\text{(e)}\ \vdash \lambda x : \langle\alpha\rangle\langle\alpha'\rangle A.\ \langle a : \alpha'\rangle\langle b : \alpha\rangle((x \,@\, b) \,@\, a) : \langle\alpha\rangle\langle\alpha'\rangle A \to \langle\alpha'\rangle\langle\alpha\rangle A$$

$$\text{(f)}\ \not\vdash \lambda x : \langle\alpha\rangle\langle\alpha\rangle A.\ \langle a : \alpha\rangle(x \,@\, a) \,@\, a : \langle\alpha\rangle\langle\alpha\rangle A \to \langle\alpha\rangle A$$

$$\text{(g)}\ \vdash \lambda x : \langle\alpha\rangle(A \to B).\ \lambda y : \langle\alpha\rangle A.\ \langle a : \alpha\rangle(x \,@\, a)\ (y \,@\, a) : \langle\alpha\rangle(A \to B) \to \langle\alpha\rangle A \to \langle\alpha\rangle B$$

$$\text{(h)}\ \not\vdash \lambda x : \langle\alpha\rangle A \to \langle\alpha\rangle B.\ \langle a : \alpha\rangle\lambda y.(x\ ??) \,@\, a : (\langle\alpha\rangle A \to \langle\alpha\rangle B) \to \langle\alpha\rangle(A \to B)$$

$$\text{(i)}\ \vdash \lambda x : \langle\alpha\rangle(A \times B).\ (\langle a : \alpha\rangle\pi_1(x \,@\, a), \langle a\rangle\pi_2(x \,@\, a)) : \langle\alpha\rangle(A \times B) \to \langle\alpha\rangle A \times \langle\alpha\rangle B$$

$$\text{(j)}\ \vdash \lambda x : \langle\alpha\rangle A \times \langle\alpha\rangle B.\ \langle a : \alpha\rangle(\pi_1(x) \,@\, a, \pi_2(x) \,@\, a) : (\langle\alpha\rangle A \times \langle\alpha\rangle B) \to \langle\alpha\rangle(A \times B)$$

Fig. 5. Example derivations and non-derivations

$$\frac{}{\Gamma \vdash \cdot : \cdot} \qquad \frac{\Gamma \vdash M : A \quad \Gamma \vdash \theta : \Gamma'}{\Gamma \vdash \theta, M/x : \Gamma', x{:}A} \qquad \frac{\Gamma \vdash b : \alpha \setminus \Gamma_0 \quad \Gamma_0 \vdash \theta : \Gamma'}{\Gamma \vdash \theta, b/a : \Gamma'\#a{:}\alpha}$$

Fig. 6. Well-formed substitutions

and $\langle\alpha'\rangle\langle\alpha\rangle A$. Example (f) shows the failure of a "contraction" law for name-abstraction; (g) shows that we can push name-abstraction inside function types; (h) shows that the converse fails in SNTT; and (i) and (j) show that name-abstractions can be pushed into and pulled out of pairs.

2.1 Formal properties

To simplify the following discussion, we introduce a "well-formedness" judgment for substitutions in Figure 6. For example, note that we may not re-use a name in a substitution once it has been used to replace some other name; that is, $\theta = a/b, a/c$ and $\theta = (a, b)/x, b/c$ are ill-formed. This is because the third rule requires us to typecheck $\theta, b/a$ by typechecking θ after removing b and all variables that may depend on b from the context. Moreover, substitutions can only perform *injective* renamings.

We write id_Γ for the identity substitution on Γ. We abbreviate $M[\mathsf{id}_\Gamma, N/x]$ as $M[N/x]$ and $M[\mathsf{id}_\Gamma, b/a]$ as $M[b/a]$, provided Γ is clear from context. We say that a context Γ is *contained* in another context Γ' (written $\Gamma \preceq \Gamma'$) if $\Gamma' \vdash \mathsf{id}_\Gamma : \Gamma$. Two contexts are *equivalent* ($\Gamma \equiv \Gamma'$) if $\Gamma \preceq \Gamma' \preceq \Gamma$. For example, $y{:}B\#a{:}\alpha, x{:}A \preceq x{:}A, y{:}B\#a{:}\alpha, z{:}C$, while $x{:}A, y{:}B \equiv y{:}B, x{:}A$.

Weakening can be established using relatively straightforward techniques. The proof is not completely trivial because we need to prove weakening for the context restriction judgment as well, which requires proving several auxiliary properties which we omit here.

Lemma 2.1 (Weakening) *If* $\Gamma \preceq \Gamma'$ *and* $\Gamma \vdash M : A$ *then* $\Gamma' \vdash M : A$.

Next, we can prove a general substitution lemma:

Lemma 2.2 (General substitution) *If* $\Gamma' \vdash \theta : \Gamma$ *and* $\Gamma \vdash M : A$ *then* $\Gamma' \vdash M[\theta] : A$.

More traditional substitution and renaming properties follow immediately by suitable choices of θ.

Lemma 2.3 (Substitution) *Suppose x does not appear in Γ. If* $\Gamma \vdash N : B$ *and* $\Gamma, x{:}B \vdash M : A$ *then* $\Gamma \vdash M[N/x] : A$.

Lemma 2.4 (Renaming) *Suppose b does not appear in Γ. If* $\Gamma \# a{:}\alpha \vdash M : A$ *then* $\Gamma \# b{:}\alpha \vdash M[b/a] : A$.

We will also need a few properties of the context restriction judgment that are easy to prove by induction:

Lemma 2.5 (Restriction) *If* $\Gamma \vdash a : \alpha \setminus \Gamma'$ *then* $a \notin \Gamma'$ *and* $\Gamma' \# a{:}\alpha \preceq \Gamma$.

We conclude by showing "local soundness" and "local completeness" properties stating that name-abstractions can be eliminated and then reintroduces without gaining or losing information (and vice versa).

Lemma 2.6 (Local soundness) *If* $\Gamma \vdash M : A$ *and* $M \rhd_\beta N$ *then* $\Gamma \vdash N : A$.

Proof. Proof is by cases on the rewriting step and inversion on derivations. For the case of name-abstraction, the reduction step is straightforward using renaming and weakening:

$$\dfrac{\Gamma \vdash b : \alpha \setminus \Gamma' \qquad \dfrac{\Gamma' \# a{:}\alpha \vdash M : A}{\Gamma' \vdash \langle a{:}\alpha\rangle M : \langle\alpha\rangle A}\,\mathsf{absI}}{\Gamma \vdash (\langle a{:}\alpha\rangle M) \mathbin{@} b : A}\,\mathsf{absE} \quad \Longrightarrow \quad \dfrac{\dfrac{\Gamma' \# a{:}\alpha \vdash M : A}{\Gamma' \# b{:}\alpha \vdash M[b/a] : A}\,R}{\Gamma \vdash M[b/a] : A}\,W$$

since by Lemma 2.5, $b{:}\alpha \notin \Gamma'$ and $\Gamma' \# b{:}\alpha \preceq \Gamma$. □

Lemma 2.7 (Local completeness) *If* $M \rhd_\eta N$ *then* $\Gamma \vdash M : A$ *if and only if* $\Gamma \vdash N : A$.

Proof. Again, the only new case is for name-abstraction. If we have $M : \langle\alpha\rangle A \rhd_\eta \langle a{:}\alpha\rangle M \mathbin{@} a$ for some $a \notin FN(M)$, then

$$\Gamma \vdash M : \langle\alpha\rangle A \iff \dfrac{\dfrac{\overline{\Gamma \# a{:}\alpha \vdash a : \alpha \setminus \Gamma} \qquad \Gamma \vdash M : \langle\alpha\rangle A}{\Gamma \# a{:}\alpha \vdash M \mathbin{@} a : A}\,\mathsf{absE}}{\Gamma \vdash \langle a{:}\alpha\rangle M \mathbin{@} a : \langle\alpha\rangle A}\,\mathsf{absI}$$

□

Theorem 2.8 (Subject reduction) *If* $\vdash M : A$ *and* $M \longrightarrow_\beta N$ *or* $M \longrightarrow_\eta N$ *then* $\vdash N : A$.

We have formalized the development to this point using the Isabelle/HOL-Nominal system [25].

2.2 Strong normalization and canonicalization

In this section we wish to show that SNTT enjoys the properties of strong normalization and canonicalization (existence of β-normal η-long canonical forms). Both the Church-Rosser and Strong Normalization theorems can be proved using essentially the same arguments as for the λ-calculus.

Theorem 2.9 (Church-Rosser) *If $M \longrightarrow_\beta M_1$ and $M \longrightarrow_\beta M_2$ then there exists N such that $M_1 \longrightarrow_\beta^* N$ and $M_2 \longrightarrow_\beta^* N$.*

Theorem 2.10 (Strong Normalization) *If $M : A$ then $M \Downarrow_\beta V$ for a unique $V : A$.*

Strong normalization has several useful consequences: we can obtain unique β-normal, η-long canonical forms by first η-expanding each subterm once according to its type and then β-normalizing. This gives a decision procedure for $\beta\eta$-equivalence.

Corollary 2.11 (Canonicalization) *If $M : A$ then M has a unique $\beta\eta$-canonical form.*

Corollary 2.12 (Decidability) *Both β- and $\beta\eta$-equivalence are decidable.*

Moreover, SNTT is a conservative extension of both ordinary simple type theory (STT) and the language of ground nominal terms.

Corollary 2.13 (Conservativity) *A judgment $\vdash M : A$ of* STT *is derivable if and only if $\vdash M : A$ is derivable in* SNTT*. Moreover, given a nominal signature Σ and ground nominal term M_0, M_0 is a well-formed term of type δ with names $a_1{:}\alpha_1, \ldots, a_n{:}\alpha_n$ if and only if $a_1{:}\alpha_1 \# \cdots \# a_n{:}\alpha_n \vdash M_0 : \delta$ is derivable in* SNTT*.*

2.3 Nominal set semantics

Although the subject reduction and strong normalization theorems imply that SNTT is a consistent equational theory, they do not explain how SNTT relates to nominal abstract syntax. In this section we show how the judgments and equations of SNTT can be interpreted using *nominal sets* [14]. This means that SNTT is sound for reasoning about names, name-abstractions, and functions in ordinary nominal logic. We leave the issues of expressiveness and completeness (for example, with respect to a suitable generalization of cartesian closed categories) for future work.

In this section, we consider the special case where there is a single name-type α, which shall correspond to the set $\mathbb{A}$ of all names. Generalizing to multiple name-types is straightforward but notationally burdensome.

We recapitulate some of the basic definitions concerning permutations, group actions and nominal sets needed for the semantics (introduced in prior work by

Gabbay and Pitts [7,14,15]). We write $\mathcal{P}_{\mathsf{fin}}(\mathbb{A})$ for the set of all finite sets of names and $\mathbf{G} = \mathit{FSym}(\mathbb{A})$ for the group of all *(finite) permutations* π on $\mathbb{A}$; that is, invertible functions such that $\pi(a) = a$ for all but finitely many $a \in \mathbb{A}$. We often write $\pi \cdot a$ for $\pi(a)$ and $\pi \cdot S$ for $\{\pi \cdot x \mid x \in S\}$ if $S \in \mathcal{P}_{\mathsf{fin}}(\mathbb{A})$. A $\mathbf{G}$*-set* is a structure $X = (|X|, \cdot_X)$ consisting of a set X equipped with a permutation action $\cdot_X : \mathbf{G} \times |X| \to |X|$, satisfying $\mathsf{id} \cdot x = x$ and $\pi \cdot \pi' \cdot x = (\pi \circ \pi') \cdot x$. We write $S \lhd x$ to indicate that $S \in \mathcal{P}_{\mathsf{fin}}(\mathbb{A})$ *supports* $x \in |X|$; that is, that $\forall a, b \in \mathbb{A} - S.\ (a\ b) \cdot_X x = x$. A *nominal set* is a $\mathbf{G}$-set in which every element has a finite support (there is then necessarily a unique least finite support). If an element x of a nominal set has empty support, it is called *equivariant*, and clearly $\pi \cdot x = x$ for any $\pi \in \mathbf{G}$.

It has been established in previous work [7] that nominal sets form a category [2] **Nom**, which includes a terminal nominal set $\mathbf{1}_{\mathbf{Nom}}$, standard constructions including Cartesian products $X \times_{\mathbf{Nom}} Y$ and function spaces $X \to_{\mathbf{Nom}} Y$, and a *name-abstraction* construction $\langle\mathbb{A}\rangle X$. We briefly review these constructions.

Definition 2.14 The terminal nominal set $\mathbf{1}_{\mathbf{Nom}}$ is defined by taking $|\mathbf{1}_{\mathbf{Nom}}| = \{\star\}$. The swapping action is defined by $\pi \cdot \star = \star$.

Definition 2.15 The cartesian product of two nominal sets X, Y is defined by taking $|X \times_{\mathbf{Nom}} Y| = |X| \times |Y|$ and defining the action by the rule $\pi \cdot (x, y) = (\pi \cdot x, \pi \cdot y)$.

Definition 2.16 The nominal set $X \to_{\mathbf{Nom}} Y$ of (finitely-supported) functions from X to Y is defined by taking $|X \to_{\mathbf{Nom}} Y| = \{f : |X| \to |Y| \mid \exists S \in \mathcal{P}_{\mathsf{fin}}(\mathbb{A}).\ S \lhd f\}$ where the swapping action is defined as $(\pi \cdot f)(x) = \pi \cdot (f(\pi^{-1} \cdot x))$.

Definition 2.17 The set of *names* $\mathbb{A}$ is a nominal set with $\pi \cdot_{\mathbb{A}} a = \pi(a)$.

Definition 2.18 Given nominal set X, the set of *abstractions* of X is called $\langle\langle\mathbb{A}\rangle\rangle X$ and defined by taking $|\langle\langle\mathbb{A}\rangle\rangle X| = (\mathbb{A} \times |X|)/_{\equiv_\alpha}$, where $\equiv_\alpha$ is the least equivalence relation satisfying

$$(a \notin \mathrm{supp}(y) \wedge x = (a\ b) \cdot_X y) \Rightarrow (a, x) \equiv_\alpha (b, y)\ .$$

The swapping action is defined as $(b\ b') \cdot [(a, x)]_\alpha = [((b\ b') \cdot a, (b\ b') \cdot x)]_\alpha$. We often abbreviate $[(a, x)]_\alpha$ as $\langle\langle a\rangle\rangle x$. Moreover, if $y \in \langle\langle\mathbb{A}\rangle\rangle X$ and $a \notin \mathrm{supp}(y)$ then we define $y \mathbin{@} a$ as $y(a)$, viewing y as a partial function.

Semantic name-abstractions satisfy analogues of the beta-reduction and eta-expansion laws for SNTT name-abstractions. We need the following key properties of name-abstraction and concretion:

Proposition 2.19 (i) *If* $\langle\langle a\rangle\rangle x \in \langle\langle\mathbb{A}\rangle\rangle X$ *and* $b \notin \mathrm{supp}(\langle\langle a\rangle\rangle x)$ *then* $(\langle\langle a\rangle\rangle x) \mathbin{@} b = (a\ b) \cdot x$ *and* $\mathrm{supp}(\langle\langle a\rangle\rangle x) = \mathrm{supp}(x) - \{a\}$.

(ii) *If* $y \in \langle\langle\mathbb{A}\rangle\rangle X$ *and* $a \notin \mathrm{supp}(y)$ *then* $y = \langle\langle a\rangle\rangle(y \mathbin{@} a)$ *and* $\mathrm{supp}(y \mathbin{@} a) \subseteq \mathrm{supp}(y) \cup \{a\}$.

[2] **Nom** is, as noted elsewhere [7,22,19], isomorphic to a well-known category called the *Schanuel topos*.

$$
\begin{aligned}
\mathcal{T}[\![\mathbf{1}]\!] &= \mathbf{1}_{\mathbf{Nom}}\\
\mathcal{T}[\![\delta]\!] &= \mathcal{T}_0[\![\delta]\!]\\
\mathcal{T}[\![\alpha]\!] &= \mathbb{A}\\
\mathcal{T}[\![A \times B]\!] &= \mathcal{T}[\![A]\!] \times_{\mathbf{Nom}} \mathcal{T}[\![B]\!]\\
\mathcal{T}[\![A \to B]\!] &= \mathcal{T}[\![A]\!] \to_{\mathbf{Nom}} \mathcal{T}[\![B]\!]\\
\mathcal{T}[\![\langle\alpha\rangle A]\!] &= \langle\!\langle\mathbb{A}\rangle\!\rangle(\mathcal{T}[\![A]\!])
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![\Gamma \vdash x : A]\!]\gamma &= \gamma(x) \qquad \mathcal{E}[\![\Gamma \vdash \mathbf{c} : A]\!]\gamma = \mathcal{E}_0[\![\mathbf{c} : A]\!]\\
\mathcal{E}[\![\Gamma \vdash a : \alpha]\!]\gamma &= \gamma(a) \qquad \mathcal{E}[\![\Gamma \vdash () : \mathbf{1}]\!]\gamma = \star\\
\mathcal{E}[\![\Gamma \vdash (M,N) : A_1 \times A_2]\!]\gamma &= (\mathcal{E}[\![\Gamma \vdash M : A_1]\!]\gamma, \mathcal{E}[\![\Gamma \vdash N : A_2]\!]\gamma)\\
\mathcal{E}[\![\Gamma \vdash \pi_i(M) : A_i]\!]\gamma &= \Pi_i(\mathcal{E}[\![\Gamma \vdash M : A_1 \times A_2]\!]\gamma)\\
\mathcal{E}[\![\Gamma \vdash \lambda x.M : A \to B]\!]\gamma &= \Lambda v \in \mathcal{T}[\![A]\!].\ \mathcal{E}[\![\Gamma, x{:}A \vdash M : B]\!]\gamma[x \mapsto v]\\
\mathcal{E}[\![\Gamma \vdash M\ N : B]\!]\gamma &= (\mathcal{E}[\![\Gamma \vdash M : A \to B]\!]\gamma)(\mathcal{E}[\![\Gamma \vdash N : A]\!]\gamma)\\
\mathcal{E}[\![\Gamma \vdash \langle a\rangle M : \langle\alpha\rangle A]\!]\gamma &= \langle\!\langle a'\rangle\!\rangle \mathcal{E}[\![\Gamma\#a{:}\alpha \vdash M : A]\!]\gamma[a \mapsto a']\\
&\qquad (a' \notin \mathrm{supp}(\gamma))\\
\mathcal{E}[\![\Gamma \vdash M\,@\,a : A]\!]\gamma &= \mathcal{E}[\![\Gamma' \vdash M : \langle\alpha\rangle A]\!] \quad (\Gamma' \vdash a : \alpha \setminus \Gamma)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{S}[\![\Gamma' \vdash \cdot : \cdot]\!]\gamma &= [\cdot]\\
\mathcal{S}[\![\Gamma' \vdash \theta, M/x : \Gamma, x : A]\!]\gamma &= \mathcal{S}[\![\Gamma' \vdash \theta : \Gamma]\!][x \mapsto \mathcal{E}[\![M]\!]\gamma]\\
\mathcal{S}[\![\Gamma' \vdash \theta, b/a : \Gamma\#a{:}\alpha]\!]\gamma &= \mathcal{S}[\![\Gamma'' \vdash \theta : \Gamma']\!][a \mapsto \gamma(b)] \quad (\Gamma' \vdash b : \alpha \setminus \Gamma'')
\end{aligned}
$$

Fig. 7. Type, substitution, and expression interpretations

Suppose we are given an interpretation of the data types δ as nominal sets $\mathcal{T}_0[\![\delta]\!]$. We interpret the other types of SNTT as nominal sets $\mathcal{T}[\![A]\!]$ as shown in Figure 7. We define the *universe* $\mathcal{U}$ of the interpretation as the disjoint union of all interpretations of types $\biguplus_A \mathcal{T}[\![A]\!]$. This is a nominal set.

A valuation is a function γ from a finite subset of $\mathbb{V} \cup \mathbb{A}$ (recall that $\mathbb{V}$ and $\mathbb{A}$ are disjoint) to the universe $\mathcal{U}$. We write $[\cdot]$ for the empty valuation and $\gamma[x \mapsto v]$ or $\gamma[a \mapsto b]$ for the result of extending valuation γ with a binding for a variable $x \notin \mathrm{dom}(\gamma)$ or name $a \notin \mathrm{dom}(\gamma)$, respectively. We define swappings to act on valuations pointwise: $(\pi \cdot \gamma)(x) = \pi \cdot (\gamma(x))$ for all $x \in \mathrm{dom}(\gamma)$. This swapping action makes valuations into a nominal set, isomorphic to the set of finite products of $\mathcal{U}$ indexed by subsets of $\mathbb{V} \cup \mathbb{A}$.

We define the set of valuations *satisfying* a context Γ as follows:

$$
\begin{aligned}
[\![\cdot]\!] &= \{[\cdot]\}\\
[\![\Gamma, x{:}A]\!] &= \{\gamma[x \mapsto v] \mid \gamma \in [\![\Gamma]\!], v \in \mathcal{T}[\![A]\!]\}\\
[\![\Gamma\#a{:}\alpha]\!] &= \{\gamma[a \mapsto b] \mid \gamma \in [\![\Gamma]\!], b \in \mathbb{A} - \mathrm{supp}(\gamma)\}
\end{aligned}
$$

Intuitively a valuation satisfies a context Γ if it maps variables to values of the appropriate types in Γ and satisfies all of the freshness constraints in Γ. Note, in particular, that no two names in $\mathrm{dom}(\gamma)$ can be mapped to the same name if $\gamma \in [\![\Gamma]\!]$ for some Γ.

Now suppose equivariant interpretations $\mathcal{E}_0[\![\mathbf{c} : A]\!] \in \mathcal{T}[\![A]\!]$ are fixed for each constant $\mathbf{c} : A \in \Sigma$. We interpret well-formed expressions M as functions $\mathcal{E}[\![\Gamma \vdash M : A]\!] : [\![\Gamma]\!] \to \mathcal{T}[\![A]\!]$ as shown in Figure 7 (technically, the definition is by recursion on derivation trees). Note that the side-condition on the definition of name-abstraction that can always be satisfied by renaming the bound name a away from the support of γ. We also interpret (well-formed) substitutions θ as functions $\mathcal{S}[\![\Gamma' \vdash \theta : \Gamma]\!] : [\![\Gamma']\!] \to [\![\Gamma]\!]$. We often abbreviate these to just $\mathcal{E}[\![M]\!]$ or $\mathcal{S}[\![\theta]\!]$ respectively. We have the following basic properties:

Proposition 2.20 (Equivariance) *For any Γ, M, A satisfying $\Gamma \vdash M : A$, we have $\mathcal{E}[\![M]\!]$ is equivariant, in the sense that for any $\pi \in FSym(\mathbb{A})$ and $\gamma \in [\![\Gamma]\!]$,*

$\pi \cdot (\mathcal{E}[\![M]\!]\gamma) = \mathcal{E}[\![M]\!](\pi \cdot \gamma)$. Moreover, $\mathrm{supp}(\mathcal{E}[\![M]\!]\gamma) \subseteq \mathrm{supp}(\gamma)$.

Proof. The first part follows by induction on M. The second is immediate: for any equivariant function $f : X \to Y$ on nominal sets X, Y, $\mathrm{supp}(f(x)) \subseteq \mathrm{supp}(x)$.□

Lemma 2.21 (Soundness of restriction) *If $\gamma \in [\![\Gamma]\!]$ and $\Gamma \vdash a : \alpha \setminus \Gamma'$ then there exists $\gamma' \in [\![\Gamma']\!]$ such that $\gamma(a) \notin \mathrm{supp}(\gamma')$ and γ' agrees with γ on Γ'.*

Theorem 2.22 (Semantic soundness) *Let $\Gamma, \gamma \in [\![\Gamma]\!]$ be given. Then (1) if $\Gamma \vdash M : A$ then $\mathcal{E}[\![M]\!]\gamma \in \mathcal{T}[\![A]\!]$, (2) if $\Gamma \vdash \theta : \Gamma'$ then $\mathcal{S}[\![\theta]\!]\gamma \in [\![\Gamma']\!]$*

Proposition 2.23 (Semantic substitution) *If $\Gamma \vdash M : A$ and $\Gamma' \vdash \theta : \Gamma$ then for any $\gamma \in [\![\Gamma']\!]$ we have $\mathcal{E}[\![M[\theta]]\!]\gamma = \mathcal{E}[\![M]\!](\mathcal{S}[\![\theta]\!]\gamma)$.*

Theorem 2.24 (Equational soundness) *If $\Gamma \vdash M, N : A$ and $M \longleftrightarrow^*_{\beta\eta} N$ then for any $\gamma \in [\![\Gamma]\!]$ we have $\mathcal{E}[\![M]\!]\gamma = \mathcal{E}[\![N]\!]\gamma$.*

3 Extensions

By itself, SNTT is not very expressive. It cannot, for example, define the size, substitution, or free-variables functions for a nominal datatype representing the syntax of a language such as the λ-calculus or π-calculus. To do so, we need booleans, numbers, conditionals, name-equality tests, and perhaps additional type constructions such as lists. More importantly, we need structural recursion over nominal datatypes. In this section we show how these features can be accommodated soundly in SNTT, by adding types, constants, and rewriting rules.

3.1 Conditionals and name-equality

Consider the type **bool**, constants **true**, **false**, and conditionals and equality tests:

$$\textbf{if } (-) \textbf{ then } (-) \textbf{ else } (-) : \textbf{bool} \to A \to A \to A \qquad (=) : \alpha \to \alpha \to \textbf{bool}$$

The reduction rules and denotational semantics for booleans and conditionals are standard. For equality tests, we consider rules:

$$\begin{array}{rl} a = a \rhd_\beta & \textbf{true} \\ a = b \rhd_\beta & \textbf{false} \end{array} \qquad \mathcal{E}[\![M = N]\!]\gamma = \begin{cases} \textbf{true} & (\mathcal{E}[\![M]\!]\gamma = \mathcal{E}[\![N]\!]\gamma) \\ \textbf{false} & (\mathcal{E}[\![M]\!]\gamma \neq \mathcal{E}[\![N]\!]\gamma) \end{cases}$$

Note that in the reduction rules, we only consider names, not arbitrary terms.

In order for the Church-Rosser and semantic soundness properties to hold, it is essential that the second β-rule $a = b \longrightarrow_\beta$ **false** is valid no matter how a and b are interpreted. Thus, we need to ensure that two syntactically distinct names never are identified as the result of a β-reduction. This is accomplished by context restriction in the absE rule. Furthermore, at the denotational level, the equality reduction rule is sound for well-typed terms precisely because if context Γ contains two distinct names a, b, then $\gamma \in [\![\Gamma]\!]$ must satisfy $\gamma(a) \neq \gamma(b)$.

3.2 Numbers and lists

Natural numbers and lists also can be defined in **Nom**. Thus, we can extend SNTT with the types **nat** and $\mathbf{list}_A$ and constants:

$$\begin{array}{rcl} \mathbf{zero} & : & \mathbf{nat} \\ \mathbf{succ} & : & \mathbf{nat} \to \mathbf{nat} \\ (+) & : & \mathbf{nat} \to \mathbf{nat} \to \mathbf{nat} \end{array} \qquad \begin{array}{rcl} []_A & : & \mathbf{list}_A \\ M ::_A N & : & A \to \mathbf{list}_A \to \mathbf{list}_A \\ \mathbf{append}_A & : & \mathbf{list}_A \to \mathbf{list}_A \to \mathbf{list}_A \\ \mathbf{remove}_A & : & \langle\alpha\rangle\mathbf{list}_\alpha \to \mathbf{list}_\alpha \end{array}$$

Standard recursion primitives can also be added. The semantics of these operations are standard, except for **remove**, which we define operationally as:

$$\mathbf{remove}(\langle a\rangle[]) \rhd_\beta [] \qquad \begin{array}{rcl} \mathbf{remove}(\langle a\rangle(a :: M)) & \rhd_\beta & \mathbf{remove}(\langle a\rangle M) \\ \mathbf{remove}(\langle a\rangle(b :: M)) & \rhd_\beta & b :: \mathbf{remove}(\langle a\rangle M) \quad (a \neq b) \end{array}$$

Note that, like name-equality, these rules rely on the fact that syntactically distinct names always differ semantically.

3.3 Nominal recursion combinators

One of the main advertised benefits of nominal logic (and related approaches such as binding algebras [4]) over other techniques has been the availability of principles for inductive reasoning and recursive definitions that extend well-known principles for induction and recursion for "first-order" languages without binding.

As one example, we consider a signature Σ_Λ with constants that define the syntax of lambda-terms modulo alpha-equivalence using nominal terms:

$$\Sigma_\Lambda = \{\mathbf{var} : \alpha \to \Lambda, \quad \mathbf{app} : \Lambda \times \Lambda \to \Lambda, \quad \mathbf{lam} : \langle\alpha\rangle\Lambda \to \Lambda\}$$

It has been shown in several places (see e.g. [3,15]) that it is possible to model this language using a least fixed point construction on nominal sets , since $X \mapsto \langle\langle\mathbb{A}\rangle\rangle X$ is a continuous operator on nominal sets. The resulting nominal set $\Lambda \cong \mathbb{A} + \Lambda \times \Lambda + \langle\langle\mathbb{A}\rangle\rangle\Lambda$ is isomorphic to the set of all (open) lambda-terms modulo α-equivalence.

Unfortunately, existing approaches to recursion on nominal terms still seem more complex than for ordinary datatypes, since additional reasoning about freshness needs to be performed. For example, in the approaches of Norrish [9], Pitts [15] and Urban and Berghofer [23], the function defining the **lam** case must satisfy a "freshness condition on binders" (FCB) stating, informally, that bound names do not escape.

In SNTT, this constraint is internalized into the implicit restrictions on names and name-abstractions enforced by the type system. We can therefore (as in [22,19]) simply introduce a recursion combinator and associated $\beta\eta$-conversions for lambda-term syntax as follows, by analogy with ordinary algebraic datatypes:

$$\begin{array}{l} \mathbf{rec}^B_\Lambda : (\alpha \to B) \to (B \times B \to B) \to (\langle\alpha\rangle B \to B) \to (\Lambda \to B) \in \Sigma_\Lambda \\ \mathbf{rec}^B_\Lambda\ f_{\mathbf{var}}\ f_{\mathbf{app}}\ f_{\mathbf{lam}}\ (\mathbf{var}\ M) \rhd_\beta f_{\mathbf{var}}\ M \\ \mathbf{rec}^B_\Lambda\ f_{\mathbf{var}}\ f_{\mathbf{app}}\ f_{\mathbf{lam}}\ (\mathbf{app}\ M) \rhd_\beta f_{\mathbf{app}}\ (\mathbf{rec}^B_\Lambda(\pi_1(M)), \mathbf{rec}^B_\Lambda(\pi_2(M))) \\ \mathbf{rec}^B_\Lambda\ f_{\mathbf{var}}\ f_{\mathbf{app}}\ f_{\mathbf{lam}}\ (\mathbf{lam}\ M) \rhd_\beta f_{\mathbf{lam}}\ (\langle a\rangle\mathbf{rec}^B_\Lambda(M \mathbin{@} a)) \\ \qquad M : \Lambda \rhd_\eta \mathbf{rec}^\Lambda_\Lambda\ (\mathbf{var})\ (\mathbf{app})\ (\mathbf{lam})\ M \end{array}$$

$$\begin{array}{rcll}
\mathit{size} & = & \lambda x.\mathbf{rec}_{\Lambda}^{\mathbf{nat}}\ (\lambda v.1)\ (\lambda m,n.m+n+1)\ (\lambda n.n+1)\ x & : \Lambda \to \mathsf{int} \\
\mathit{subst} & = & \lambda x,f.\mathbf{rec}_{\Lambda}^{\Lambda}\ (f)\ (\mathbf{app})\ (\mathbf{lam})\ x & : \Lambda \to (\alpha \to \Lambda) \to \Lambda \\
\mathit{subst}1 & = & \lambda x,y,v.\mathit{subst}\ x\ (\lambda w.\mathbf{if}\ v=w\ \mathbf{then}\ y\ \mathbf{else}\ \mathbf{var}(w)) & : \Lambda \to \Lambda \to \alpha \to \Lambda \\
\mathit{fvs} & = & \lambda x.\mathbf{rec}_{\Lambda}^{\mathbf{list}_\alpha}(\lambda v.[v])\ (\mathbf{append})\ (\mathbf{remove})\ x & : \Lambda \to \mathbf{list}_\alpha
\end{array}$$

Fig. 8. Examples of recursive definitions.

A few examples are shown in Figure 8, including the size function, capture-avoiding substitution functions (for both simultaneous and single substitution), and the free variables function. All of these examples typecheck, and this is all that is necessary to ensure that they correspond to total, terminating functions on λ-terms.

As in (pure) FreshML, we cannot define a "bound variables function" bvs that returns, for example, $[a]$ given $\mathbf{lam}(\langle a\rangle\mathbf{var}(a))$. No such "function" exists in the nominal set semantics. Moreover, the type system prevents us from defining such a function because given a variable of type $\langle\alpha\rangle A$, there is no way to generate a name of type α that we can return in the case for **lam**. Conversely, many (computable) functions exist in the nominal set semantics but cannot be defined using SNTT. In particular, functions that rely on "well-behaved" use of local names, such as Pitts' definition of normalization by evaluation [15], cannot be handled in SNTT.

As noted earlier, SNTT signatures and terms generalize the nominal signatures and ground nominal terms considered in prior work [15,24]. Given any nominal signature Σ, it is straightforward to define recursion principles for all of its datatypes, along with suitable β-reduction and η-expansion rules, to obtain an extended system SNTT(Σ). Moreover, we conjecture that Theorems 2.8–2.22 continue to hold for any such extension, but have not proved this; a complete investigation is deferred to future work.

4 Related and Future Work

A great deal of research on both nominal and other techniques for abstract syntax with binding informs and motivates this work; we cannot give a complete survey here. We have discussed closely related research in the body of the paper, and in this section will briefly discuss additional closely related and recent work; more detailed discussion can be found in [3,15].

SNTT's contexts are a simplified form of the bunched contexts in the $\alpha\lambda$-calculus [11]. It would be natural to add a type $\alpha\#A$ consisting of "fresh" pairs (a,x) where $a \notin \mathrm{supp}(x)$; this would be a simple form of the $*$-type in the $\alpha\lambda$-calculus, and would be adjoint to name-abstraction However, we would need to make contexts more complicated to handle this.

Pottier [16] has recently revisited the problem of inferring freshness information for "pure" FreshML [12]. This approach reduces freshness inference to set constraint solving and is aimed towards practical programming rather than deduction. This system can typecheck more programs, but may require programmers to decorate types with freshness constraints.

Recursion for nominal abstract syntax has been studied by several authors

[9,23,15]. Recursion principles have also been developed using other techniques such as binding algebras [4], functor category semantics [8], the Theory of Contexts [1], modal type systems [20] and parametricity [26]. Schürmann, Poswolsky, and Sarnat's ∇-calculus [21] is a core language for the Elphin programming language, a language for functional programming with simply-typed higher-order abstract syntax. More recently [18] have extended this approach in the language Delphin that permits functional programming over arbitrary dependent LF signatures. Pientka [13] has developed a related, but distinct approach to functional programming over LF specifications using explicit contexts and context polymorphism.

These approaches seem to require more effort to use than recursion in SNTT. Previous nominal techniques involve checking freshness side-conditions; techniques such as binding algebras and the Theory of Contexts rely on nontrivial type-theoretic or category-theoretic foundations; and most previous higher-order techniques do not provide equality on names. SNTT's simplicity comes at the (apparent) price of expressiveness; however, little is known about how any of these approaches compare (or should be compared) in terms of expressiveness. Comparing the expressiveness of these approaches seems like a crucial area for future work.

Another immediate direction for future work is extending SNTT to richer type theories such as dependent or polymorphic types which could serve as proof systems for larger fragments of nominal logic. We believe that SNTT addresses the main obstacle to combining nominal terms with ordinary λ-calculi, but, as in the dependent type theories of Schöpp and Stark [22,19], there may be complications arising from the interaction of names and dependency or polymorphism.

One feature of nominal logic not presently reflected in SNTT is the *equivariance principle* [14] stating that validity is preserved by bijective (or, equivalently, injective) renaming. In SNTT, this would take the form of an *explicit swapping* proof term $\pi \cdot M$, where π is a permutation. We have omitted explicit permutation terms because they seem to complicate the presentation (for example, by making the Church-Rosser property more difficult to prove) without contributing much; nevertheless, they may be necessary to model full nominal logic.

Another direction for future work is developing a type-theoretic version of nominal logic's *freshness principle*, which says that we may always obtain a fresh name. In SNTT, a natural way to model the freshness principle is to incorporate a *name-generation* term $\nu a{:}\alpha.M$. By analogy with the ν construct in the π-calculus, and with FreshML's `let fresh` construct, such an operator would permit us to obtain a fresh name a and use it within M. However, the behavior of $\nu a{:}\alpha.M$ as a proof term is problematic. Similar name-generation operators has been studied independently of name-binding by Pitts and Stark [17] and Odersky [10]. Pitts and Stark's semantics for ν is, like FreshML's `let fresh`, generative; conversely, Odersky's approach is purely functional but it admits well-formed, "stuck" terms such as $\nu a.a$ that are not considered to be values. Neither approach seems compatible with a denotational reading of proof terms as pure functions, at least without making the system considerably more complex to prevent ν-generated names from "escaping". Resolving the tension between $\nu a.M$'s intuitive generative reading and the needs of a pure type theory seems a significant challenge, which we hope to tackle next.

Of course, we are also interested in completely formalizing and verifying the metatheory of SNTT itself; we have already formalized the key syntactic properties (up to subject reduction) using Isabelle/HOL-Nominal system [25].

5 Conclusion

Although proof systems for classical nominal logic have been studied extensively, constructive techniques have received less attention. This paper develops a simple type theory for nominal terms, called SNTT, that combines ordinary unit, pair, and function types with the names and name-abstractions of nominal logic. SNTT is sound and strongly normalizing, can be interpreted using nominal sets, and can easily be extended with recursion combinators for languages with binding that are simpler, albeit less expressive, than in previous systems. Extensions to richer type theories or larger fragments of nominal logic remain to be investigated.

Acknowledgement

The author was supported by EPSRC grant EP/F028288/1 while performing this research. Thanks are also due to Andrew Pitts, Christian Urban and the anonymous reviewers for detailed comments and suggestions.

References

[1] Anna Bucalo, Furio Honsell, Marino Miculan, Ivan Scagnetto, and Martin Hoffman. Consistency of the theory of contexts. *J. Funct. Program.*, 16(3):327–372, 2006.

[2] J. Cheney. A simpler proof theory for nominal logic. In *FOSSACS 2005*, volume 3441 of *LNCS*, pages 379–394. Springer-Verlag, 2005.

[3] J. Cheney. Completeness and Herbrand theorems for nominal logic. *Journal of Symbolic Logic*, 71(1):299–320, 2006.

[4] M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In *LICS 1999*, pages 193–202. IEEE Press, 1999.

[5] M. J. Gabbay. Fresh logic: proof-theory and semantics for FM and nominal techniques. *Journal of Applied Logic*, 5(2):356–387, June 2007.

[6] M. J. Gabbay and J. Cheney. A sequent calculus for nominal logic. In *LICS 2004*, pages 139–148. IEEE, 2004.

[7] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2002.

[8] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In Giuseppe Longo, editor, *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, pages 204–213, Washington, DC, 1999. IEEE, IEEE Press.

[9] M. Norrish. Recursive function definitions for types with binders. In *TPHOLs*, number 3223 in LNCS, pages 241–256. Springer-Verlag, 2004.

[10] Martin Odersky. A functional theory of local names. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 48–59, January 1994.

[11] Peter O'Hearn. On bunched typing. *J. Funct. Program.*, 13(4):747–796, 2003.

[12] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In *MPC 2000*, number 1837 in LNCS, pages 230–255. Springer-Verlag, 2000.

[13] Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *POPL*, pages 371–382, 2008.

[14] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 183:165–193, 2003.

[15] Andrew M. Pitts. Alpha-structural recursion and induction. *Journal of the ACM*, 53(3):459–506, May 2006.

[16] François Pottier. Static name control for FreshML. In *LICS 2007*, pages 356–365, Wroclaw, Poland, July 2007.

[17] Andrew Pitts and Ian Stark. Observable properties of higher order functions that dynamically create local names, or: What's *new*? In *MFCS 1993*, number 711 in LNCS, pages 122–141. Springer-Verlag, 1993.

[18] Adam Poswolsky and Carsten Schürmann. Practical programming with higher-order encodings and dependent types. In *ESOP*, number 4960 in LNCS, pages 93–107, 2008.

[19] Ulrich Schöpp. *Names and Binding in Type Theory.* PhD thesis, University of Edinburgh, 2006.

[20] Carsten Schürmann, Joelle Despeyroux, and Frank Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266:1–57, 2001.

[21] Carsten Schürmann, Adam Poswolsky, and Jeffrey Sarnat. The [triangle]-calculus. functional programming with higher-order encodings. In Pawel Urzyczyn, editor, *TLCA*, volume 3461 of *Lecture Notes in Computer Science*, pages 339–353. Springer, 2005.

[22] Ulrich Schöpp and Ian Stark. A dependent type theory with names and binding. In *CSL 2004*, number 3210 in LNCS, pages 235–249, Karpacz, Poland, 2004.

[23] C. Urban and S. Berghofer. A recursion combinator for nominal datatypes implemented in Isabelle/HOL. In *IJCAR*, volume 4130 of *LNCS*, pages 498–512. Springer-Verlag, 2006.

[24] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. *Theoretical Computer Science*, 323(1–3):473–497, 2004.

[25] Christian Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.

[26] G. Washburn and S. Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *ICFP 2003*, pages 249–262, 2003.