Nominal System T

Andrew M. Pitts *
University of Cambridge Computer Laboratory, Cambridge, UK
Andrew.Pitts@cl.cam.ac.uk

Abstract

This paper introduces a new recursion principle for inductive data modulo α-equivalence of bound names. It makes use of Odersky-style local names when recursing over bound names. It is formulated in an extension of Gödel's System T with names that can be tested for equality, explicitly swapped in expressions and restricted to a lexical scope. The new recursion principle is motivated by the nominal sets notion of "α-structural recursion", whose use of names and associated freshness side-conditions in recursive definitions formalizes common practice with binders. The new *Nominal System T* presented here provides a calculus of total functions that is shown to adequately represent α-structural recursion while avoiding the need to verify freshness side-conditions in definitions and computations. Adequacy is proved via a normalization-by-evaluation argument that makes use of a new semantics of local names in Gabbay-Pitts nominal sets.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Syntax; D.3.3 [*Programming Languages*]: Language Constructs and Features—Recursion; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Lambda calculus and related systems

General Terms Languages, Theory, Verification

Keywords binders, alpha-equivalence, recursion, types

1. Introduction

When developing mathematical semantics for programming languages, it is commonplace to ignore concrete syntax and work at the level of abstract syntax trees. Indeed, if the language involves binding constructs (as most do), one usually raises the level of abstraction even further by implicitly quotienting abstract syntax trees by an appropriate notion of α-equivalence. Working modulo α-equivalence affects how one defines functions by structural recursion and proves properties of them by structural induction—the fundamental tools of programming language semantics. For example, the parameters of a recursively defined function can have their bound names changed as necessary, but one is obliged to prove that the value of the defined function is independent of such changes. Such proofs of well-definedness up to α-equivalence are often omitted as obvious. This paper presents a calculus of (total, higher-order) functions with structural recursion modulo α-equivalence for which such proofs of well-definedness really are obvious, indeed are entirely automatic, being subsumed by a completely conventional and decidable type system. This is achieved without giving up on the practically convenient "nominal" treatment of binding in which bound names are first-class citizens that can be tested for equality, passed to functions as arguments and returned as results. We call the calculus presented in this paper *Nominal System T* since it takes Gödel's System T for primitive recursive functions of higher type [13] (formulated as a typed λ-calculus, following Tait [34]) and generalizes from numbers to inductive data modulo α-equivalence of bound names. To bring out the ideas underlying our approach without too much syntactic clutter, we use the familiar and simple example of such data, the terms of the untyped λ-calculus [2]

$$\Lambda \triangleq \{t ::= a \mid t\,t \mid \lambda a.\,t\} \tag{1}$$

where a ranges over an infinite set $\mathbb{A}$ of variables and where terms are identified up to the usual notion of α-equivalence ($\equiv_\alpha$) for λ-bound variables.

When making a structurally recursive definition of a function $f : \Lambda \to X$ in terms of functions $f_1 : \mathbb{A} \to X$, $f_2 : \Lambda \to \Lambda \to X \to X \to X$ and $f_3 : \mathbb{A} \to \Lambda \to X \to X$, one can take advantage of the identification of terms up to $\equiv_\alpha$ by restricting the applicability of the recursion equation for λ-binders:

$$\left.\begin{array}{rl} & f\,a = f_1\,a \\ & f(t\,t') = f_2\,t\,t'\,(f\,t)\,(f\,t') \\ a \notin \overline{a} \;\Rightarrow\; & f(\lambda a.\,t) = f_3\,a\,t\,(f\,t). \end{array}\right\} \tag{2}$$

Thus in the third clause we restrict the recursion equation to apply only for bound variables a that avoid some finite set $\overline{a}$ of variables—typically the ones that are involved in the definition of the functions f_1, f_2, f_3. For example, the function $f(-) = (-)[t'/a'] : \Lambda \to \Lambda$ for capture-avoiding substitution of $t' \in \Lambda$ for $a' \in \mathbb{A}$ is given by taking $f_1\,a \triangleq \mathit{if}\ a = a'\ \mathit{then}\ t'\ \mathit{else}\ a$, $f_2\,t\,t'\,x\,x' \triangleq x\,x'$, $f_3\,a\,t\,x \triangleq \lambda a.\,x$ and $\overline{a}$ to be the finite set consisting of a' and the free variables of t'.

Of course, for (2) to specify a well-defined function on α-equivalence classes, the function f_3 in general has to satisfy a condition ensuring independence of the definiens in the third clause from choice of the bound variable a used to represent the λ-abstraction $\lambda a.\,t$ in the definiendum. What kind of condition should we impose on f_3 to ensure this? The notion of *α-structural recursion* gives an answer to this question in terms of the nominal sets [11, 25]. We briefly recall what is involved; see Pitts [26] for the full story.

The main idea is to take account of name-permutations, which means for the example we are using, finite permutations π of the set $\mathbb{A}$. A *nominal set* is a set X equipped with an action of such permutations (written $\pi, x \mapsto \pi \cdot x$) for which every $x \in X$ is *finitely supported*. This means that given x, there is a finite subset

* Research supported by UK EPSRC grant EP/D000459/1

POPL'10, January 17–23, 2010, Madrid, Spain.

$\overline{a} \subseteq \mathbb{A}$ such that for any $a, a' \in \mathbb{A} - \overline{a}$, the permutation $(a\ a')$ that swaps a and a' leaves x invariant: $(a\ a') \cdot x = x$. Complementary to finite support is the notion of *freshness*: we say "a is fresh for x" and write $a \# x$ if x is supported by some finite set not containing a. For example, permutations act on elements of $\mathbb{A}$ by application and this makes $\mathbb{A}$ into a nominal set with the relation of freshness being "not equal". More generally, Λ equipped with the obvious permutation action is a nominal set and freshness is the "not a free variable of" relation. The fun begins at higher types. Given nominal sets X and Y, the nominal set $X \to_{\mathrm{fs}} Y$ consists not of all functions f from X to Y, but just the ones that are finitely supported with respect to the usual permutation action on functions; that f is supported by finite $\overline{a} \subseteq \mathbb{A}$ amounts to requiring for any $a, a' \in \mathbb{A} - \overline{a}$ and $x \in X$ that $f((a\ a') \cdot x) = (a\ a') \cdot (f\ x)$. Using these concepts Pitts [26, Remark 4.4] proves:

α-Structural Recursion Principle for Λ. *Given finitely supported functions $f_1 \in \mathbb{A} \to_{\mathrm{fs}} X$, $f_2 \in \Lambda \to_{\mathrm{fs}} \Lambda \to_{\mathrm{fs}} X \to_{\mathrm{fs}} X \to_{\mathrm{fs}} X$ and $f_3 \in \mathbb{A} \to_{\mathrm{fs}} \Lambda \to_{\mathrm{fs}} X \to_{\mathrm{fs}} X$, supported by $\overline{a} \subseteq \mathbb{A}$ say, suppose f_3 satisfies the following "freshness condition for binders"*: [1]

$$\forall a \in \mathbb{A} - \overline{a}. \forall t \in \Lambda. \forall x \in X.\ a \# f_3\, a\, t\, x. \qquad \text{(FCB)}$$

Then there is a unique function $f \in \Lambda \to_{\mathrm{fs}} X$ satisfying the scheme (2)*, necessarily also finitely supported by $\overline{a}$.* □

For the example of capture-avoiding substitution mentioned above, (FCB) holds because $f_3\, a\, t\, x = \lambda a.\, x$ and $a \# \lambda a.\, x$ since for any λ-term $x \in \Lambda$, a is not free in $\lambda a.\, x$.

The Nominal package of Urban and Berghofer [35] for the Isabelle proof assistant implements α-structural recursion (and more) within Isabelle/HOL. Growing experience with Nominal Isabelle suggests that, despite the need to prove lemmas about freshness, this is a convenient formalization within higher-order logic of structural recursion in the presence of binders. However, those freshness side-conditions mean that α-structural recursion is not a convenient basis for a *calculus*, as opposed to a logic, of "recursive functions modulo α". We will show how to modify it so that given f_1, f_2, f_3, there is always a well-defined f with good computational properties without the need to prove (FCB).

FreshML [31, 33] achieves this within the context of an impure functional programming language; freshness conditions get automatically satisfied by dynamically allocating fresh names at runtime. Stateful operational semantics do not give rise to equational calculi with good logical properties. Indeed, even such an apparently simple computational effect as dynamic allocation of names is known to interact in complicated ways with higher-order functions [27]; for example, function expressions can fail to behave extensionally [28, Example 1.2]. Instead of trying to tame dynamic allocation in this context [30], here we propose to avoid it altogether by using (a typed version of) the pure functional theory of local names of Odersky [21]. We give this a denotational semantics using nominal sets, which in turn suggests adding an explicit name-swapping operation to Odersky's calculus. Odersky's theory of local names may seem too simple: compared with the ν-calculus of Pitts and Stark [27] there is no scope extrusion of local names from function arguments and no sharing of local names between components of a tuple. Nevertheless, with the addition of name-swapping it allows us to formulate a new calculus of higher order recursive functions over Λ that we prove adequately represents α-structural recursion.

Contributions of this paper. We give a new (and simple) semantics for Odersky-style local names based upon the notion of a name-restriction operator in the Gabbay and Pitts [11] model of names and binding (Definition 1). This model suggests an extension of Gödel's system T with local names that can be tested for equality and explicitly swapped in expressions. The resulting *Nominal System T* contains a ground type `Trm` whose terms represent λ-terms modulo α-equivalence of bound names; it also contains a recursion combinator that makes use of local names when recursing over data representing λ-abstractions (see (δ`L`) in Fig. 3). We consider generalizations to other data with binders besides λ-terms in Sect. 7; the generalization relies upon the new observation for nominal sets that concretion of atom-abstractions can be made a total function in the presence of name-restriction (Theorem 22).

We identify a notion of η-long β-normal form for the well-typed expressions of Nominal System T and prove that every well-typed expression is convertible to such a normal form, unique up to a simple structural congruence (Theorem 10). The proof of this is via a version of the "normalization by evaluation" technique encompassing structural congruence of normal forms (Definition 15). Armed with this normalization property, we show that functions defined by the α-Structural Recursion Principle for Λ described above, can be faithfully represented in Nominal System T (Theorem 21). In this way we achieve the aim of producing an expressive calculus of higher-order recursive total functions that avoids the need for freshness conditions when defining and computing. Indeed, Nominal System T has a completely conventional type system and all freshness conditions associated with α-equivalence are elevated to the meta-level.

2. Syntax and Semantics

The types and expressions of Nominal System T are given in Fig 1. The part below the dotted lines is what is being added to Gödel's System T. There are two new ground types: `Atm` classifies names of object-level variables, and `Trm` classifies terms of the object-language, which we take to be the untyped λ-calculus. Expressions may involve two different kinds of identifier: *variables* $x \in \mathbb{V}$, standing for unknown expressions, and *atomic names* $a \in \mathbb{A}$, standing for unknown object-level variables. Both kinds of identifier may be bound: the binding forms are function abstraction $\lambda x.\,(-)$, name restriction $\nu a.\,(-)$ and object-level λ-abstraction $\mathtt{L}a.\,(-)$. As discussed in the Introduction, following the usual informal practice **expressions are implicitly identified up to α-equivalence of these bound identifiers**.

To simplify the presentation of the syntax we use "Church-style" explicitly-typed variables.[2] Thus we assume the countably infinite set $\mathbb{V}$ is partitioned into disjoint, countably infinite subsets $\mathbb{V}(T)$ as T ranges over types; the elements of $\mathbb{V}(T)$ are the variables of type T. Figure 2 gives the inductive definition of the set $Exp(T)$ of well-typed expressions for each type $T \in Typ$. As before, the typing rules below the dotted line are what is being added to Gödel's System T. There should be no surprises for the reader in the new rules: name restriction and name-swapping do not change the type of the expression acted upon; we have made name-equality take a polymorphic boolean type in order not to introduce a separate ground type for booleans; and the arguments for `Trm`-recursion, `trec`, have types as discussed in the Introduction.

The intended model of Nominal System T is in terms of *nominal sets* [11, 25], the definition of which was recalled in the Introduction. In other words, expressions denote mathematical objects that are finitely supported with respect to the action of permutations of atomic names. Finite permutations can be decomposed into compositions of transpositions; and name-swapping expressions $(e_1\ e_2) * e$ denote the action of such transpositions. What do expressions of the form $\nu a.\, e$ denote? To answer this question we

[1] (FCB) is equivalent to the apparently weaker condition that there is some $a \notin \overline{a}$ satisfying $\forall t \in \Lambda. \forall x \in X.\ a \# f_3\, a\, t\, x$.

[2] "Curry-style", with variables assigned types by environments, is possible and would be desirable for dependently-typed extensions of the system.

$$
\begin{array}{rcll}
T \in \mathit{Typ} & ::= & T \to T & \text{function type} \\
 & | & G & \text{ground type} \\
G & ::= & \mathtt{Nat} & \text{type of numbers} \\
\cdots\cdots & & & \\
 & | & \mathtt{Atm} & \text{type of names} \\
 & | & \mathtt{Trm} & \text{type of object-level } \lambda\text{-terms} \\
 & & & \text{modulo } \alpha\text{-equivalence}
\end{array}
$$

$$
\begin{array}{rcll}
e \in \mathit{Exp} & ::= & x & \text{variable } (x \in \mathbb{V}) \\
 & | & \mathtt{0} & \text{zero} \\
 & | & \mathtt{S}\, e & \text{successor} \\
 & | & \mathtt{nrec}\, e\, e\, e & \mathtt{Nat} \text{ recursion} \\
 & | & \lambda x.\, e & \text{function abstraction} \\
 & | & e\, e & \text{function application} \\
\cdots\cdots & & & \\
 & | & a & \text{atomic name } (a \in \mathbb{A}) \\
 & | & \nu a.\, e & \text{name restriction} \\
 & | & (e = e)_T & \text{name equality} \\
 & | & (e\ e) * e & \text{name swapping} \\
 & | & \mathtt{V}\, e & \text{object-level variable} \\
 & | & \mathtt{A}\, e\, e & \text{application term} \\
 & | & \mathtt{L} a.\, e & \lambda\text{-abstraction term} \\
 & | & \mathtt{trec}\, e\, e\, e\, e & \mathtt{Trm} \text{ recursion}
\end{array}
$$

Some defined expressions (where $a \in \mathbb{A}$ and $x, x' \in \mathbb{V}(T)$):

$$
\begin{array}{rcll}
\mathtt{new} & \triangleq & \nu a.\, a & \\
\mathtt{true}_T & \triangleq & \lambda x.\, \lambda x'.\, x & \\
\mathtt{false}_T & \triangleq & \lambda x.\, \lambda x'.\, x' & \\
\mathtt{L}(e, e') & \triangleq & \mathtt{L} a.\, (a\ e) * e' & \text{where } a \text{ is not free in } e \text{ or } e'
\end{array}
$$

Figure 1. Types T and expressions e

$$
\frac{x \in \mathbb{V}(T)}{x \in \mathit{Exp}(T)} \qquad \frac{}{\mathtt{0} \in \mathit{Exp}(\mathtt{Nat})} \qquad \frac{e \in \mathit{Exp}(\mathtt{Nat})}{\mathtt{S}\, e \in \mathit{Exp}(\mathtt{Nat})}
$$

$$
\frac{e_1 \in \mathit{Exp}(T) \qquad e_2 \in \mathit{Exp}(\mathtt{Nat} \to T \to T) \qquad e \in \mathit{Exp}(\mathtt{Nat})}{\mathtt{nrec}\, e_1\, e_2\, e \in \mathit{Exp}(T)}
$$

$$
\frac{x \in \mathbb{V}(T) \qquad e \in \mathit{Exp}(T')}{\lambda x.\, e \in \mathit{Exp}(T \to T')} \qquad \frac{e \in \mathit{Exp}(T' \to T) \qquad e' \in \mathit{Exp}(T')}{e\, e' \in \mathit{Exp}(T)}
$$

$$
\frac{a \in \mathbb{A}}{a \in \mathit{Exp}(\mathtt{Atm})} \qquad \frac{a \in \mathbb{A} \qquad e \in \mathit{Exp}(T)}{\nu a.\, e \in \mathit{Exp}(T)}
$$

$$
\frac{e_1, e_2 \in \mathit{Exp}(\mathtt{Atm}) \qquad T \in \mathit{Typ}}{(e_1 = e_2)_T \in \mathit{Exp}(T \to T \to T)} \qquad \frac{e_1, e_2 \in \mathit{Exp}(\mathtt{Atm}) \qquad e \in \mathit{Exp}(T)}{(e_1\ e_2) * e \in \mathit{Exp}(T)}
$$

$$
\frac{e \in \mathit{Exp}(\mathtt{Atm})}{\mathtt{V}\, e \in \mathit{Exp}(\mathtt{Trm})} \qquad \frac{e, e' \in \mathit{Exp}(\mathtt{Trm})}{\mathtt{A}\, e\, e' \in \mathit{Exp}(\mathtt{Trm})} \qquad \frac{a \in \mathbb{A} \qquad e \in \mathit{Exp}(\mathtt{Trm})}{\mathtt{L} a.\, e \in \mathit{Exp}(\mathtt{Trm})}
$$

$$
\frac{\begin{array}{c} e_1 \in \mathit{Exp}(\mathtt{Atm} \to T) \qquad e_2 \in \mathit{Exp}(\mathtt{Trm} \to \mathtt{Trm} \to T \to T \to T) \\ e_3 \in \mathit{Exp}(\mathtt{Atm} \to \mathtt{Trm} \to T \to T) \qquad e \in \mathit{Exp}(\mathtt{Trm}) \end{array}}{\mathtt{trec}\, e_1\, e_2\, e_3\, e \in \mathit{Exp}(T)}
$$

Figure 2. Well-typed expressions $e \in \mathit{Exp}(T)$ $[T \in \mathit{Typ}]$

consider nominal sets R equipped with the following extra structure.

Definition 1 (name-restriction operations). A *name-restriction operation* on a nominal set R is function $\mathbb{A} \times R \to R$, written $(a, r) \mapsto (\nu a)r$, that satisfies (3)–(6). △

$$\pi \cdot (\nu a)r = (\nu\, \pi(a))(\pi \cdot r) \tag{3}$$

$$a \# (\nu a)r \tag{4}$$

$$a \# r \ \Rightarrow\ (\nu a)r = r \tag{5}$$

$$(\nu a)(\nu a')r = (\nu a')(\nu a)r. \tag{6}$$

Property (3) says the function preserves the action of permutations π. In general this property is called *equivariance* (and implies that the function is supported by the empty set of atoms).

In the presence of (3), property (4) is the nominal sets way of modelling the α-equivalence that we have built in to the syntax of the term-former $\nu a.\, (-)$. For if $a' \# (a, r)$, then by a standard argument, equivariance implies $a' \# (\nu a)r$. Hence neither a', nor a by (4), are in the support of $(\nu a)r$. Therefore $(\nu a)r = (a\ a') \cdot (\nu a)r = (\nu a')((a\ a') \cdot r)$, by equivariance again. Thus $(\nu a)r$ is invariant under renaming a with a fresh atomic name a'.

The other two properties (5) and (6) correspond to basic "structural" properties of a notion of name restriction. Structural congruence for name-restriction first arose in connection with the reduction semantics of the π-calculus [18], where it involves "scope extrusion" properties in addition to (4)–(6). More recently, Gacek et al. [12, Sect. 2.3] call property (5) *strengthening* and (6) *exchange* and use these structural properties in connection with locally scoped eigenvariables and generic judgements in inductive proofs.

We can get a name-restriction operation for a nominal set of λ-terms by adjoining a constant New to the grammar in (1)

$$\Lambda[\mathit{New}] \ \triangleq\ \{t ::= a \mid t\, t \mid \lambda a.\, t \mid \mathit{New}\} \tag{7}$$

and defining $(\nu a)t$ to be $t[\mathit{New}/a]$. The ground type $\mathtt{Trm}$ stands for this nominal-set-with-restriction-operation, $\Lambda[\mathit{New}]$. The ground type $\mathtt{Atm}$ stands for a substructure of this, namely $\mathbb{A}[\mathit{New}] \triangleq \mathbb{A} \cup \{\mathit{New}\}$; and the ground type $\mathtt{Nat}$ stands for the usual set of natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$ with trivial permutation action $(\pi \cdot k = k)$ and trivial name-restriction operation $((\nu a)k = k)$. For higher types we make use of the following new result about nominal sets.

Theorem 2 (name-restriction on functions). *Suppose that X and R are nominal sets and that R is equipped with a name-restriction operation ν_R. Then there is a name-restriction operation ν on the nominal set $X \to_{\mathrm{fs}} R$ of finitely supported functions satisfying*

$$a \# x \ \Rightarrow\ ((\nu a)f)x = (\nu_R a)(f\, x) \tag{8}$$

for all $a \in \mathbb{A}$, $x \in X$ and $f \in X \to_{\mathrm{fs}} R$.

Proof. Given $a \in \mathbb{A}$ and $f \in X \to_{\mathrm{fs}} R$, we define $(\nu a)f \in X \to_{\mathrm{fs}} R$ by mapping each $x \in X$ to

$$((\nu a)f)\, x \ \triangleq\ (\nu_R a')(((a\ a') \cdot f)\, x) \quad \text{where } a' \# (a, f, x). \tag{9}$$

It is not hard to see that the right-hand side is independent of the choice of a'; that the resulting function $(\nu a)f$ is finitely supported (by $\overline{a} - \{a\}$, if $\overline{a}$ supports f); and that $(a, f) \mapsto (\nu a)f$ satisfies (3)–(6) and (8). □

In view of this theorem, if we interpret the ground types $\mathtt{Nat}$, $\mathtt{Atm}$ and $\mathtt{Trm}$ as $\mathbb{N}$, $\mathbb{A}[\mathit{New}]$ and $\Lambda[\mathit{New}]$ respectively, and interpret function types using exponentiation $(-) \to_{\mathrm{fs}} (-)$ in $\mathcal{N}om$, then every type of Nominal System T denotes a nominal set equipped with a restriction operation. These restriction operations are used

$$(\lambda x.\, e)\, e' \approx e[e'/x] \qquad (\beta)$$
$$e \approx \lambda x.(e\, x) \quad \text{if } x \# e \qquad (\eta)$$

when $f \triangleq \mathtt{nrec}\, e_1\, e_2$:
$$f\, \mathtt{0} \approx e_1 \qquad (\delta\mathtt{0})$$
$$f\, (\mathtt{S}\, e) \approx e_2\, e\, (f\, e) \qquad (\delta\mathtt{S})$$

$$\nu a.\, e \approx e \quad \text{if } a \# e \qquad (\nu\#)$$
$$\nu a.\, \nu a'.\, e \approx \nu a'.\, \nu a.\, e \qquad (\nu\nu)$$

when $n, n' \in \mathbb{A} \cup \{\mathtt{new}\}$:
$$(n = n')_T \approx \mathtt{true}_T \quad \text{if } n = n' \qquad (=t)$$
$$(n = n')_T \approx \mathtt{false}_T \quad \text{if } n \neq n' \qquad (=f)$$

when $\mathtt{K} = \mathtt{0}, \mathtt{S}, \mathtt{V}, \mathtt{A}$:
$$\nu a.\, \mathtt{K}\, \vec{e} \approx \mathtt{K}\, \nu a.\, \vec{e} \qquad (\nu\mathtt{K})$$
$$(e_1\ e_2) * (\mathtt{K}\, \vec{e}) \approx \mathtt{K}\, ((e_1\ e_2) * \vec{e}) \qquad (\pi\mathtt{K})$$
$$\nu a.\, \mathtt{L}a'.\, e \approx \mathtt{L}a'.\, \nu a.\, e \quad \text{if } a \neq a' \qquad (\nu\mathtt{L})$$
$$(e_1\ e_2) * \mathtt{L}a.\, e \approx \mathtt{L}a.\, (e_1\ e_2) * e \quad \text{if } a \# (e_1, e_2) \qquad (\pi\mathtt{L})$$
$$\nu a.\, \lambda x.\, e \approx \lambda x.\, \nu a.\, e \qquad (\nu\lambda)$$
$$(e_1\ e_2) * \lambda x.\, e \approx \lambda x.\, (e_1\ e_2) * (e[(e_1\ e_2) * x\ /\ x]) \quad \text{if } x \# (e_1, e_2) \qquad (\pi\lambda)$$
$$(e_1\ e_2) * a \approx (e_1 = a)_{\mathtt{Atm}}\, e_2\, ((e_2 = a)_{\mathtt{Atm}}\, e_1\, a) \qquad (\pi a)$$
$$(e_1\ e_2) * \mathtt{new} \approx \mathtt{new} \qquad (\pi\mathtt{new})$$

when $f \triangleq \mathtt{trec}\, e_1\, e_2\, e_3$:
$$f\, (\mathtt{V}\, e) \approx e_1\, e \qquad (\delta\mathtt{V})$$
$$f\, (\mathtt{A}\, e\, e') \approx e_2\, e\, e'\, (f\, e)\, (f\, e') \qquad (\delta\mathtt{A})$$
$$f\, (\mathtt{L}a.\, e) \approx \nu a.\, e_3\, a\, e\, (f\, e) \quad \text{if } a \# (e_1, e_2, e_3) \qquad (\delta\mathtt{L})$$

$\approx$ is the congruence generated by these conversions

Figure 3. Conversion $\approx \subseteq Exp(T) \times Exp(T)$ $[T \in Typ]$

to interpret name-restriction expressions $\nu a.\, e$ of each type. Finally, Nat-recursion expressions $\mathtt{nrec}\, e_1\, e_2\, e$ and Trm-recursion expressions $\mathtt{trec}\, e_1\, e_2\, e_3\, e$ are interpreted using primitive recursion for numbers and α-structural recursion [26] for λ-terms (extended with a constant New) respectively. We leave the formal details of this denotational semantics of Nominal System T to the full version of this paper and pass on to computational issues.

3. Conversion

Figure 3 gives the notion of expression equality that we use for Nominal System T: *conversion*, $e \approx e' [T \in Typ,\, e, e' \in Exp(T)]$. Rather than giving a notion of rewriting between expressions and proving canonicity (interesting though that might be), we prefer to go directly to a notion of equality and prove decidability via a normalization function (given in Sect. 5). For one thing, our conversion relation contains reductions, expansions and a "structural" conversion $(\nu\nu)$ that has no preferred orientation; so considering conversion rather than reduction seems to make more sense. Figure 3 uses the notation $e[e'/x]$ for the capture-avoiding substitution of e' for all free occurrences of x in e. It also uses the notation $x \# e$ (respectively, $a \# e$) to indicate that x is not a free variable (respectively, a is not a free atomic name) of e. (This coincides with the freshness relation when we regard $Exp(T)$ as a nominal set in Sect. 5.) Finally, the figure also uses some notation in connection with sequences of expressions: if $\vec{e} = e_1..e_n$, then $\nu a.\vec{e} \triangleq (\nu a.\, e_1)..(\nu a.\, e_n)$ and $(e\ e') * \vec{e} \triangleq ((e\ e') * e_1)..((e\ e') * e_n)$. The conversions below the dotted line in Fig. 3 are what is being added to Gödel's System T. Although we omit the proof in this extended abstract, it is not hard to show:

Theorem 3 (soundness). *Conversion is sound for equality in the nominal sets model sketched in the previous section.* □

After the fact, and rather pleasingly, the conversions in Fig. 3 for $\nu a.\, (-)$ turn out to agree with the functional theory of local names given by Odersky [21]; conversion $(\nu\lambda)$ corresponds to his ν_λ reduction, and $(\nu\mathtt{K})$ to his ν_p reduction. The structural conversions $(\nu\#)$ and $(\nu\nu)$ are not explicit in Odersky's system, but are valid up to contextual equivalence. However, Nominal System T takes a rather more "logical" view of name-equality: in Odersky's system $\nu a.\, a$ is not a value (canonical form) and $\nu a.\, a == \nu a.\, a$ is a stuck expression that does not reduce; whereas here $\nu a.\, a$ is a normal form (denoting the constant New in our nominal sets model) and $(\nu a.\, a = \nu a.\, a)_T$ is convertible to $\mathtt{true}_T$ by the conversion $(=t)$. Note that in general $\nu a.(e = e')_T \not\approx ((\nu a.\, e) = (\nu a.\, e'))_T$; for example, if $a \neq a'$, then using $(=f)$ and $(\nu\#)$ we have

$$\nu a.\nu a'.(a = a')_T \approx \nu a.\nu a'.\, \mathtt{false}_T \approx \mathtt{false}_T \qquad (10)$$

whereas, using $(\nu\#)$ and $(=t)$, we have

$$((\nu a.\nu a'.\, a) = (\nu a.\nu a'.\, a'))_T \approx (\mathtt{new} = \mathtt{new})_T \approx \mathtt{true}_T \qquad (11)$$

and it is a corollary of Theorem 3 that $\mathtt{true}_T \not\approx \mathtt{false}_T$. Note also that although

$$(\nu a.e)e' \approx \nu a.\, (e\, e') \quad \text{if } a \# e' \qquad (12)$$

is provable from the conversions (β), (η) and $(\nu\lambda)$ in Fig. 3, in general $e\, (\nu a.\, e') \not\approx \nu a.\, (e\, e')$. Evidently Odersky's is a different notion of "local name" from the more common one (in Scheme, ML, Haskell, ...) based on dynamic allocation of globally fresh names. It has better logical properties (for example, it does not disturb function extensionality) and is in a sense more general than dynamic allocation, because the latter can be encoded in it via a continuation monad; cf. Shinwell and Pitts [32].

The conversions in Fig. 3 for $(e_1\ e_2) * (-)$ correspond to the defining properties, at each type, of the permutation action in the nominal sets model. In particular, the rather complicated looking conversion $(\pi\lambda)$ just reflects the usual definition of the action of permutations on functions (see Pitts [26, Sect. 3.2], for example).

The last three conversions in Figure 3, $(\delta\mathtt{V})$–$(\delta\mathtt{L})$, give the new "recursion modulo α" scheme which in Sect. 6 is shown to adequately represent the α-structural recursion of Pitts [26]. We give some examples of its use.

Example 4 (substitution, non-capturing and capturing). Consider $\mathtt{sub} \triangleq \lambda x.\, \lambda y.\, \lambda z.\, \mathtt{trec}\, e_1\, e_2\, e_3\, z \in Exp(\mathtt{Atm} \to \mathtt{Trm} \to \mathtt{Trm} \to \mathtt{Trm})$, where
$e_1 \triangleq \lambda x'.\, (x = x')_{\mathtt{Trm}}\, y\, (\mathtt{V}\, x')$,
$e_2 \triangleq \lambda y.\, \lambda y'.\, \lambda z.\, \lambda z'.\, \mathtt{A}\, z\, z'$,
$e_3 \triangleq \lambda x.\, \lambda y.\, \lambda z.\, \mathtt{L}(x, z)$
and where $\mathtt{L}(x, z) \triangleq \mathtt{L}a.\, (a\ \ x) * z$ is an instance of the definition given at the bottom of Fig. 1. It is a non-binding binary operation for object-level λ-abstraction (the analogue of the atom-abstraction operation $e, e' \mapsto \texttt{<}e\texttt{>}e'$ in FreshML [33]). If $e, e' \in Exp(\mathtt{Trm})$, then we claim that $\mathtt{sub}\ a'\, e'\, e$ represents the capture-avoiding substitution of (the λ-term represented by) e' for $\mathtt{V}\, a'$ in (the λ-term represented by) e. For example, $\mathtt{L}a.\, \mathtt{V}\, a'$ represents the λ-term $\lambda a.\, a'$; so, assuming a and a' are distinct atomic names, $\mathtt{sub}\ a'\, (\mathtt{V}\, a)\, (\mathtt{L}a.\, \mathtt{V}\, a')$ should represent $(\lambda a.\, a')[a/a']$, that is, $\lambda a''.\, a$ where $a'' \neq a$. Indeed, one can use the conversion equa-

tions in Fig. 3 to calculate that

$$\begin{aligned}
&\mathit{sub}\ a'\,(\mathtt{V}\,a)\,(\mathtt{L}a.\,\mathtt{V}\,a')\\
&\quad= \mathit{sub}\ a'\,(\mathtt{V}\,a)\,(\mathtt{L}a''.\,\mathtt{V}\,a') \quad \text{where } a'' \neq a, a'\\
&\quad\approx \mathtt{trec}\,e_1[a'/x, \mathtt{V}\,a/y]\,e_2\,e_3\,(\mathtt{L}a''.\,\mathtt{V}\,a')\\
&\quad\approx \nu a''.\,e_3\,a''\,(\mathtt{V}\,a')\,(\mathtt{trec}\,e_1[a'/x, \mathtt{V}\,a/y]\,e_2\,e_3\,(\mathtt{V}\,a'))\\
&\quad\approx \nu a''.\,\mathtt{L}(a'', \mathtt{V}\,a)\\
&\quad\triangleq \nu a''.\,\mathtt{L}a'''.(a'''\ a'')*(\mathtt{V}\,a) \quad \text{where } a''' \neq a, a', a''\\
&\quad\approx \nu a''.\,\mathtt{L}a'''.\mathtt{V}\,a\\
&\quad\approx \mathtt{L}a'''.\,\mathtt{V}\,a
\end{aligned}$$

bearing in mind that we identify expressions up to α-equivalence of bound atomic names. Doing so has the consequence that the object-level capture-avoidance property of sub is delegated to the properties of meta-level α-equivalence. This is an idea familiar from higher-order abstract syntax [23], except that here we are "baking in" to the meta-language just object-level α-equivalence rather than object-level $\alpha\beta\eta$-equivalence (in order to keep hold of a very simple yet expressive recursion principle). The claim that sub correctly represents capture-avoiding substitution is substantiated in Sect. 6. It is worth noting that we can also represent the kind of *capturing* substitution that may occur when a λ-term context has its hole filled. Holes are represented by variables x; and hole-filling by substitution of expressions for variables, $e[e'/x]$. For example the context $\lambda a.[-]$ can be represented by the open expression $\mathtt{L}(a, x) = \mathtt{L}a'.\,(a'\ a)*x$. Filling the hole in $\lambda a.[-]$ with a gives $\lambda a.\,a$; and correspondingly, the substituted expression $\mathtt{L}(a, x)[\mathtt{V}\,a\ /\ x]$ is indeed convertible to $\mathtt{L}a.\,\mathtt{V}\,a$ (cf. Lemma 19(ii)).

Example 5 (length of a λ-term). Consider the function $|-| : \Lambda \to \mathbb{N}$ satisfying $|a| = 1$, $|t\,t'| = |t| + |t'|$ and $|\lambda a.\,t| = |t| + 1$. What could be simpler? And yet formal recursion schemes for λ-terms have found it tricky: see Gordon and Melham [14, Sect. 3.3] and Norrish [20, Sect. 3]. We can represent this function in Nominal System T more or less directly by

$$\begin{aligned}
&\mathit{len} \triangleq \lambda x.\,\mathtt{trec}\,e_1\,e_2\,e_3\,x \in \mathit{Exp}(\mathtt{Trm} \to \mathtt{Nat}),\ \text{where}\\
&\quad e_1 \triangleq \lambda x.\,\mathtt{S}\,\mathtt{0}\\
&\quad e_2 \triangleq \lambda x, x', y, y'.\,\mathit{plus}\ y\,y'\\
&\quad e_3 \triangleq \lambda x, y, z.\,\mathtt{S}\,z
\end{aligned}$$

and where $\mathit{plus} \triangleq \lambda x, y.\,\mathtt{nrec}\,x\,(\lambda x', y'.\,\mathtt{S}\,y')\,y$ is the usual primitive recursive definition of addition. For example $|\lambda a.\,a| = 2$ and

$$\begin{aligned}
\mathit{len}\,(\mathtt{L}a.\,\mathtt{V}\,a) &\approx \mathtt{trec}\,e_1\,e_2\,e_3\,(\mathtt{L}a.\,\mathtt{V}\,a)\\
&\approx \nu a.\,e_3\,a\,(\mathtt{V}\,a)\,(\mathtt{trec}\,e_1\,e_2\,e_3\,(\mathtt{V}\,a))\\
&\approx \nu a.\,\mathtt{S}(e_1\,a)\\
&\approx \nu a.\,\mathtt{S}(\mathtt{S}\,\mathtt{0})\\
&\approx \mathtt{S}(\mathtt{S}\,\mathtt{0})
\end{aligned}$$

where the last step is the "strengthening" conversion $(\nu\#)$.

Example 6 (computing with bound variables). The nominal treatment of binders allows us to compute with bound names. What if we try to do something with them that would break object-level α-equivalence?—such as trying to compute a list of bound variables in a λ-term:

$$\begin{aligned}
\mathit{bv}\,(\mathtt{V}\,e) &\approx \mathit{nil}\\
\mathit{bv}\,(\mathtt{A}\,e\,e') &\approx \mathit{append}\,(\mathit{bv}\,e)\,(\mathit{bv}\,e')\\
\mathit{bv}\,(\mathtt{L}a.\,e) &\approx \nu a.\,\mathit{cons}\,a\,(\mathit{bv}\,e)
\end{aligned}$$

where we encode lists of atomic names as certain expressions of type $\mathtt{Trm}$ and nil, cons and append are suitable encodings of nil, cons and append operations for such lists. Clearly we can get the above conversions by defining $\mathit{bv} \triangleq \lambda x.\,\mathtt{trec}\,e_1\,e_2\,e_3\,x$ for suitable choices of expression e_1, e_2 and e_3. However, all that bv computes is a list of $\mathtt{new}$'s whose length is the number of occurrences of bound variables in the λ-term. For example $\mathit{bv}\,(\mathtt{L}a.\mathtt{L}a'.\,\mathtt{V}\,a) \approx \mathit{cons}\,\mathtt{new}\,(\mathit{cons}\,\mathtt{new}\,\mathit{nil})$. Compare this with the Fresh Objective Caml function `listBvars` of Shinwell and Pitts [31, p. 15].

Remark 7 ($\mathtt{new}$ versus $\nu a.\,(-)$). Can the binder $\nu a.\,(-)$ be eliminated in favour of a constant $\mathtt{new}$? In the ν-calculus [27], $\mathtt{new}$ computes a dynamically allocated fresh name and the corresponding local name binder $\nu x.\,e$ can be defined as $(\lambda x.\,e)\,\mathtt{new}$. This does not work for Nominal System T; for example, $\nu a.\,(a = \mathtt{new})_T \approx \mathtt{false}_T$ whereas $(\lambda x.\,(x = \mathtt{new})_T)\,\mathtt{new} \approx \mathtt{true}_T$. On the other hand in the nominal sets model that was sketched in Section 2, name-restriction for the types of Nominal System T is ultimately defined in terms of a constant New. So it may be possible, if inconvenient, to reformulate the system with a constant $\mathtt{new}$ rather than a binder $\nu a.\,(-)$.

Remark 8 (incompleteness). We remarked above (Theorem 3) that conversion is a sound axiomatization of properties of equality in the nominal sets model of Nominal System T. It is not a complete axiomatization for the usual recursion-theoretic reasons. In fact the definition of conversion was chosen to be as weak as possible subject to the criteria that it be decidable, have relatively simple normal forms (see Sect. 4) and adequately represent α-structural recursion when restricted to closed expressions (see Sect. 6). Even if we restrict to the "finite" part of the system, there are non-convertible expressions that are identified in the model. For example, $\lambda x.\,(a\ a)*x$ denotes the identity function in the model, but is not convertible to $\lambda x.\,x$. It would certainly be interesting to investigate stronger, but still sound and decidable, notions of conversion.

4. Normal Forms

Figure 4 gives a notion of η-long β-normal form for Nominal System T. We call elements n of the subset $Nf(T) \subseteq \mathit{Exp}(T)$ *normal forms* of type T; and elements u of the subset $Neu(T) \subseteq \mathit{Exp}(T)$ *neutral forms* of type T. Note that in the figure, $G \in \{\mathtt{Nat}, \mathtt{Atm}, \mathtt{Trm}\}$ is a ground type; so, as usual for simply typed λ-calculus, only neutral forms of ground type are normal forms. In addition, note that name-restriction $\nu a.\,(-)$ only occurs in the normal form $\mathtt{new}$ (recall that $\mathtt{new}$ is $\nu a.\,a$ by definition) and applied to neutral forms *of ground type*; similarly, name-swapping is only applied to neutral forms of ground type.

If $u \in Neu(G)$ and a does not occur free in u, then u and $\nu a.\,u$ are different elements of $Nf(G)$ even though $u \approx \nu a.\,u$ by $(\nu\#)$. Similarly, $\nu a.\nu a'.\,u$ and $\nu a'.\nu a.\,u$ are different elements of $Nf(G)$ (so long as $a \neq a'$) even though $\nu a.\nu a'.\,u \approx \nu a'.\nu a.\,u$ by $(\nu\nu)$. However, these are essentially the only instances where conversion between normal forms does not coincide with syntactic identity (modulo α-equivalence, of course). More precisely, it is a consequence of the Normalization Theorem below that conversion restricted to normal forms coincides with the following simple notion of structural congruence.

Definition 9 (structural congruence). The relations of *structural congruence*

$$\begin{aligned}
n &\equiv n' && [T \in \mathit{Typ},\ n, n' \in Nf(T)]\\
u &\equiv u' && [T \in \mathit{Typ},\ u, u' \in Neu(T)]
\end{aligned}$$

comprise the congruence on normal and neutral forms generated by

$$\nu a.\,u \equiv u \qquad \text{if } a \# u \tag{13}$$

$$\nu a.\nu a'.\,u \equiv \nu a'.\nu a.\,u \tag{14}$$

for all $u \in Neu(G)$ and $G \in \{\mathtt{Nat}, \mathtt{Atm}, \mathtt{Trm}\}$. △

$$\frac{}{\mathtt{0} \in Nf(\mathtt{Nat})} \qquad \frac{n \in Nf(\mathtt{Nat})}{\mathtt{S}\, n \in Nf(\mathtt{Nat})}$$

$$\frac{x \in \mathbb{V}(T) \qquad n \in Nf(T')}{\lambda x.\, n \in Nf(T \to T')} \qquad \frac{u \in Neu(G)}{u \in Nf(G)} \qquad \frac{x \in \mathbb{V}(T)}{x \in Neu(T)}$$

$$\frac{\begin{array}{c} n_1 \in Nf(T) \\ n_2 \in Nf(\mathtt{Nat} \to T \to T) \\ u \in Neu(\mathtt{Nat}) \end{array}}{\mathtt{nrec}\, n_1\, n_2\, u \in Neu(T)} \qquad \frac{u \in Neu(T \to T') \qquad n \in Nf(T)}{u\, n \in Neu(T')}$$

$$\frac{a \in \mathbb{A}}{a \in Nf(\mathtt{Atm})} \qquad \frac{}{\mathtt{new} \in Nf(\mathtt{Atm})} \qquad \frac{n \in Nf(\mathtt{Atm})}{\mathtt{V}\, n \in Nf(\mathtt{Trm})}$$

$$\frac{n, n' \in Nf(\mathtt{Trm})}{\mathtt{A}\, n\, n' \in Nf(\mathtt{Trm})} \qquad \frac{a \in \mathbb{A} \qquad n \in Nf(\mathtt{Trm})}{\mathtt{L}a.\, n \in Nf(\mathtt{Trm})}$$

$$\frac{u \in Neu(\mathtt{Atm}) \qquad n \in Nf(\mathtt{Atm})}{(u = n)_T, (n = u)_T \in Neu(T \to T \to T)}$$

$$\frac{a \in \mathbb{A} \qquad u \in Neu(G)}{\nu a.\, u \in Neu(G)} \qquad \frac{n, n' \in Nf(\mathtt{Atm}) \qquad u \in Neu(G)}{(n\ n') * u \in Neu(G)}$$

$$\frac{\begin{array}{c} n_1 \in Nf(\mathtt{Atm} \to T) \qquad n_2 \in Nf(\mathtt{Trm} \to \mathtt{Trm} \to T \to T \to T) \\ n_3 \in Nf(\mathtt{Atm} \to \mathtt{Trm} \to T \to T) \qquad u \in Neu(\mathtt{Trm}) \end{array}}{\mathtt{trec}\, n_1\, n_2\, n_3\, u \in Neu(T)}$$

(N.B. G ranges over the ground types $\mathtt{Nat}$, $\mathtt{Atm}$ and $\mathtt{Trm}$.)

Figure 4. Normal forms n and neutral forms u

Properties (13) and (14) correspond to the second two defining properties, (5) and (6) respectively, of the semantic notion of name-restriction that we introduced in the previous section. (We noted there that the first two defining properties (3) and (4) correspond to α-equivalence.)

Theorem 10 (normalization theorem). *Each typeable expression is convertible to a normal form, which is unique up to structural congruence. More precisely, for each $T \in Typ$ there is a function $nf_T : Exp(T) \to Nf(T)$ satisfying:*

$$\forall e \in Exp(T).\ e \approx nf_T\, e \tag{15}$$

$$\forall e \in Exp(T). \forall n \in Nf(T).\ e \approx n \ \Rightarrow\ nf_T\, e \equiv n \tag{16}$$

and hence also

$$\forall e, e' \in Exp(T).\ e \approx e' \ \Rightarrow\ nf_T\, e \equiv nf_T\, e'. \tag{17}$$

The proof of the theorem is sketched in the next section. It is a corollary of the proof that the normalization functions nf_T are computable. Since structural congruence $\equiv$ is evidently a decidable relation, it follows that conversion is decidable too.

5. Normalization by Evaluation

We prove Theorem 10 using a version of "normalization by evaluation" (NbE). Since its introduction by Berger and Schwichtenberg [4], NbE has been applied to a number of applied λ-calculi, both typed and untyped, including ones encompassing Gödel's System T [3, 8]. In each case the terms of the calculus are evaluated in a suitable model that characteristically mixes syntax (variables, at least) with semantics (extensional functions, usually). "Suitable model" means one for which a reification function (an "inverse of evaluation") can be defined from the model back to the terms; composing reification with evaluation yields the normalization function. The reification function usually involves the need to find syntactic variables that are "fresh with respect to semantic objects". As explained by Pitts [26, Sect. 6], the theory of nominal sets provides a convenient way of making sense of this kind of freshness and it is the way we adopt here. However, this use of nominal sets in NbE is orthogonal to the use we made of them in Section 2 to provide a standard model of Nominal System T; and it should also be stressed that there are other ways of dealing with the kind of freshness required by reification, or indeed of avoiding it in some circumstances: see Abel et al. [1, Sect. 1].

$\downarrow_T : [\![T]\!] \to Nf(T)$:

$$\begin{aligned} \downarrow_G n &\triangleq n \\ \downarrow_{T \to T'} f &\triangleq \lambda x.\, \downarrow_{T'} (f(\uparrow_T x)) \\ &\quad \text{where } x \in \mathbb{V}(T) \text{ and } x \# f \end{aligned}$$

$\uparrow_T : Neu(T) \to [\![T]\!]$:

$$\begin{aligned} \uparrow_G u &\triangleq u \\ \uparrow_{T \to T'} u &\triangleq \lambda d \in [\![T]\!].\, \uparrow_{T'} (u\, (\downarrow_T d)) \end{aligned}$$

Figure 5. Reification $\downarrow_T$ and reflection $\uparrow_T$

The model in which we evaluate the expressions of Nominal System T uses nominal sets based on permutations not just of atomic names $a \in \mathbb{A}$, but also of variables $x \in \mathbb{V}(T)$ (for each type $T \in Typ$). In other words we use the "many-atom-sorted" version of nominal sets described by Pitts [26, Sect. 3] with finite permutations of $\mathbb{A} \cup \mathbb{V}$ that map atomic names to atomic names and variables to variables, preserving their types. Letting such permutations act on expressions in the obvious way, each $Exp(T)$ is a nominal set with the support of $e \in Exp(T)$ being the finite set of free atomic names and free variables of e. Thus $a \# e$ (respectively $x \# e$) holds if and only if a (respectively x) does not occur free in e. The permutation action $\pi, e \mapsto \pi \cdot e$ preserves the property of being normal or neutral, so that $Nf(T)$ and $Neu(T)$ are nominal subsets of $Exp(T)$. Using the nominal sets $Nf(G)$ of normal forms at ground types G and taking finitely supported functions $(X \to_{\mathrm{fs}} Y)$ at higher types we get:

Definition 11 (interpretation of types). For each type $T \in Typ$ we define a nominal set $[\![T]\!]$ as follows:

$$\begin{aligned} [\![G]\!] &\triangleq Nf(G) \quad (G = \mathtt{Nat}, \mathtt{Atm}, \mathtt{Trm}) \\ [\![T \to T']\!] &\triangleq [\![T]\!] \to_{\mathrm{fs}} [\![T']\!]. \end{aligned}$$ △

Figure 5 defines equivariant functions of *reification* $\downarrow_T : [\![T]\!] \to Nf(T)$ and *reflection* $\uparrow_T : Neu(T) \to [\![T]\!]$ simultaneously by recursion on the structure of types $T \in Typ$. As mentioned above, we take advantage of the nominal sets notion of freshness ($\#$) in the clause for $\downarrow_{T \to T'}$: since f is finitely supported we can always find an $x \in \mathbb{V}(T)$ satisfying $x \# f$ and the normal form $\lambda x.\, \downarrow_{T'} (f(\uparrow_T x))$ is independent of which one we use. Apart from this use of nominal sets, the definitions of $[\![T]\!]$, $\downarrow_T$ and $\uparrow_T$ are quite standard, that is, they do not depend upon the new features that we have added to the simply typed λ-calculus. It is when we come to the definition of evaluation, $[\![e]\!]\rho$, whose properties are specified in Fig. 6, that these new features have to be taken into account. The figure makes use of some auxiliary functions, defined in Fig. 7, which in turn make use of $\downarrow_T$ and $\uparrow_T$. (This was why we gave the definition of reification and reflection before defining evaluation.) Evaluation takes place in the presence of environments of the following kind.

$$
\begin{aligned}
[\![x]\!]\rho &= \rho(x)\\
[\![\lambda x.\, e]\!]\rho &= \lambda d \in [\![T]\!].\, [\![e]\!](\rho[x{\mapsto}d])\\
[\![e\, e']\!]\rho &= [\![e]\!]\rho\,([\![e']\!]\rho)\\
[\![\mathtt{K}\,\vec{e}]\!]\rho &= \mathtt{K}([\![\vec{e}]\!]\rho) \quad \text{for } \mathtt{K} = \mathtt{0}, \mathtt{S}, \mathtt{V}, \mathtt{A}\\
[\![\mathtt{L}a.\, e]\!]\rho &= \mathtt{L}a.\,([\![e]\!]\rho) \quad \text{if } a \# \rho\\
[\![a]\!]\rho &= a\\
[\![\nu a.\, e]\!]\rho &= (\nu a)([\![e]\!]\rho) \quad \text{if } a \# \rho\\
[\![(e = e')_T]\!]\rho &= eq_T([\![e]\!]\rho, [\![e]\!]\rho)\\
[\![(e_1\ e_2) * e]\!]\rho &= ([\![e_1]\!]\rho\ [\![e_2]\!]\rho) \odot [\![e]\!]\rho\\
[\![\mathtt{nrec}\, e_1\, e_2\, e]\!]\rho &= nrec_T([\![e_1]\!]\rho, [\![e_2]\!]\rho, [\![e]\!]\rho)\\
[\![\mathtt{trec}\, e_1\, e_2\, e_3\, e]\!]\rho &= trec_T([\![e_1]\!]\rho, [\![e_2]\!]\rho, [\![e_3]\!]\rho, [\![e]\!]\rho)
\end{aligned}
$$

Figure 6. Evaluation $[\![e]\!]\rho \in [\![T]\!]$ $[e \in \mathit{Exp}(T),\ \rho \in \mathit{Env}]$

Definition 12 (environments). The nominal set $\mathit{Env} \triangleq (T \in \mathit{Typ}) \to_{\mathrm{fs}} \mathbb{V}(T) \to_{\mathrm{fs}} [\![T]\!]$ of *environments* consists of all finitely supported functions mapping, for each type $T \in \mathit{Typ}$, variables $x \in \mathbb{V}(T)$ to elements $\rho(x) \in [\![T]\!]$. The *initial environment* $\rho_0 \in \mathit{Env}$ is given by:

$$\rho_0\, x \triangleq \uparrow_T x \quad [T \in \mathit{Typ},\ x \in \mathbb{V}(T)] \tag{18}$$

(relying upon the fact that $\mathbb{V}(T) \subseteq \mathit{Neu}(T)$). If $\rho \in \mathit{Env}$, $x \in \mathbb{V}(T)$ and $d \in [\![T]\!]$, then $\rho[x{\mapsto}d]$ denotes the *updated environment* mapping x to d and otherwise acting like ρ. △

The fact that environments are finitely supported objects is used in Fig. 6 in the clauses for λ-abstraction terms $\mathtt{L}a.\, e$ and name-restriction expressions $\nu a.\, e$. Indeed, Fig. 6 constitutes a definition of $[\![e]\!]\rho$ by α-structural recursion [26] over expressions e, for all environments ρ simultaneously; and this requires the following "freshness conditions on binders" to be verified:

$$x \# \lambda d \in [\![T]\!].\, [\![e]\!](\rho[x{\mapsto}d]) \tag{19}$$

$$a \# \mathtt{L}a.\,([\![e]\!]\rho) \tag{20}$$

$$a \# (\nu a)([\![e]\!]\rho). \tag{21}$$

The first follows from standard properties of the environmental semantics of λ-abstraction: see the discussion following equation (101) in Pitts [26]; property (20) is trivial, since a is never free in $\mathtt{L}a.\, n$ for any $n \in \mathit{Nf}(\mathtt{Trm})$; and property (21) can be proved by induction on the type of e. Further applications of α-structural recursion are needed in the definitions of $(\nu a)(-)$, $(n_1\ n_2)\odot -$ and $\mathit{trec}(f_1, f_2, f_3, -)$ in Fig. 7.

The following lemma shows that for normal forms, reification provides a left inverse for evaluation in the initial environment (18), modulo structural congruence; it is proved by induction on the (height of the) derivations of $n \in \mathit{Nf}(T)$ and $u \in \mathit{Neu}(T)$ from the rules in Fig. 4.

Lemma 13. *(i)* $\forall n \in \mathit{Nf}(T).\ \downarrow_T([\![n]\!]\rho_0) \equiv n$.
(ii) $\forall u \in \mathit{Neu}(T).\ [\![u]\!]\rho_0 = \uparrow_T u$. □

Definition 14 (normalization function). For each $T \in \mathit{Typ}$ we define $\mathit{nf}_T : \mathit{Exp}(T) \to \mathit{Nf}(T)$ to map $e \in \mathit{Exp}(T)$ to

$$\mathit{nf}_T\, e \triangleq \downarrow_T([\![e]\!]\rho_0)$$

where $\rho_0 \in \mathit{Env}$ is the initial environment given in (18). △

To prove that nf_T has the desired properties (15) and (16) we continue to follow the standard pattern of a NbE proof and introduce a suitable logical relation between semantics and syntax. However, the presence of structural congruence $\equiv$ in Theorem 10

$(\nu a)d \in [\![T]\!]$ $[T \in \mathit{Typ}, a \in \mathbb{A}, d \in [\![T]\!]]$:

$$
\begin{aligned}
(\nu a)(\mathtt{K}\,\vec{n}) &\triangleq \mathtt{K}\,(\nu a)\vec{n} \quad \text{where } \mathtt{K} = \mathtt{0}, \mathtt{S}, \mathtt{V}, \mathtt{A}\\
(\nu a)(\mathtt{L}a'.\, n) &\triangleq \mathtt{L}a'.\,(\nu a)n \quad \text{if } a \neq a'\\
(\nu a)a' &\triangleq \begin{cases} \mathtt{new} & \text{if } a = a'\\ a' & \text{if } a \neq a' \end{cases}\\
(\nu a)\mathtt{new} &\triangleq \mathtt{new}\\
(\nu a)u &\triangleq \nu a.\, u\\
((\nu a)f)\, d &\triangleq (\nu a')(((a\ a') \cdot f)\, d) \quad \text{where } a' \# (a, f, d)
\end{aligned}
$$

$eq_T(n, n') \in [\![T \to T \to T]\!]$ $[T \in \mathit{Typ}, n, n' \in [\![\mathtt{Atm}]\!]]$:

$$
\begin{aligned}
eq_T(n, n') &\triangleq \begin{cases} \lambda d, d' \in [\![T]\!].\, d & \text{if } n = n' \in \mathbb{A} \cup \{\mathtt{new}\}\\ \lambda d, d' \in [\![T]\!].\, d' & \text{if } n \neq n' \in \mathbb{A} \cup \{\mathtt{new}\} \end{cases}\\
eq_T(u, n) &\triangleq \uparrow_{T\to T\to T}(u = n)_T\\
eq_T(n, u) &\triangleq \uparrow_{T\to T\to T}(n = u)_T
\end{aligned}
$$

$(n_1\ n_2)\odot d \in [\![T]\!]$ $[T \in \mathit{Typ}, n_1, n_2 \in [\![\mathtt{Atm}]\!], d \in [\![T]\!]]$:

$$
\begin{aligned}
(n_1\ n_2)\odot(\mathtt{K}\,\vec{n}) &\triangleq \mathtt{K}\,(n_1\ n_2)\odot\vec{n} \quad \text{where } \mathtt{K} = \mathtt{0}, \mathtt{S}, \mathtt{V}, \mathtt{A}\\
(n_1\ n_2)\odot \mathtt{L}a.\, n &\triangleq \mathtt{L}a.\,(n_1\ n_2)\odot n \quad \text{if } a \# (n_1, n_2)\\
(n_1\ n_2)\odot a &\triangleq eq(n_1, a)_{\mathtt{Atm}}\, n_2\, (eq(n_2, a)_{\mathtt{Atm}}\, n_1\, a)\\
(n_1\ n_2)\odot \mathtt{new} &\triangleq \mathtt{new}\\
(n_1\ n_2)\odot u &\triangleq (n_1\ n_2) * u\\
(n_1\ n_2)\odot f &\triangleq \lambda d \in [\![T]\!].\, (n_1\ n_2)\odot(f((n_1\ n_2)\odot d))
\end{aligned}
$$

$nrec_T(d, f, n) \in [\![T]\!]$ $[T \in \mathit{Typ}, d \in [\![T]\!], f \in [\![\mathtt{Nat} \to T \to T]\!], n \in [\![\mathtt{Nat}]\!]]$
is defined by primitive recursion and a direct definition on neutral forms:

$$
\begin{aligned}
nrec_T(d, f, \mathtt{0}) &= d\\
nrec_T(d, f, \mathtt{S}\, n) &= f\, n\, (nrec_T(d, f, n))\\
nrec_T(d, f, u) &= \uparrow_T(\mathtt{nrec}(\downarrow_T d)(\downarrow_{\mathtt{Nat}\to T\to T} f)\, u)
\end{aligned}
$$

$trec_T(f_1, f_2, f_3, n) \in [\![T]\!]$ $[T \in \mathit{Typ}, f_1 \in [\![\mathtt{Atm} \to T]\!], f_2 \in [\![\mathtt{Trm}\to\mathtt{Trm}\to T\to T\to T]\!], f_3 \in [\![\mathtt{Atm}\to\mathtt{Trm}\to T\to T]\!], n \in [\![\mathtt{Trm}]\!]]$
is defined by α-structural recursion and a direct definition on neutral forms:

$$
\begin{aligned}
trec_T(\vec{f}, \mathtt{V}\, n) &= f_1\, n\\
trec_T(\vec{f}, \mathtt{A}\, n\, n') &= f_2\, n\, n'\, (trec_T(\vec{f}, n))\, (trec_T(\vec{f}, n'))\\
trec_T(\vec{f}, \mathtt{L}a.\, n)) &= (\nu a)(f_3\, a\, n\, (trec_T(\vec{f}, n))) \quad \text{if } a \# (\vec{f})\\
trec_T(\vec{f}, u) &= \uparrow_T(\mathtt{trec}(\downarrow_{\mathtt{Atm}\to\mathtt{Trm}} f_1)(\downarrow_{\mathtt{Trm}\to\mathtt{Trm}\to T\to T\to T} f_2)\\
&\qquad (\downarrow_{\mathtt{Atm}\to\mathtt{Trm}\to T\to T} f_3)\, u)
\end{aligned}
$$

Figure 7. Auxiliary functions used in Fig. 6

complicates matters. We could have dealt with it by quotienting out in the model and using $Nf(G)/\equiv$ at ground types and finitely supported functions over those nominal sets at higher types. However, with an eye to formalizations of the proof of normalization in system such as Coq [`coq.inria.fr`] or Agda [`wiki.portal.chalmers.se/agda`], we prefer the more intensional model we have given. As a result we use the following ternary logical relation, rather than a binary one.

Definition 15 (**logical relation**). The nominal subsets $\mathcal{R}_T \subseteq [\![T]\!] \times [\![T]\!] \times \mathit{Exp}(T)$ are defined by recursion on the structure of $T \in \mathit{Typ}$:

$$(n_1, n_2, e) \in \mathcal{R}_G \;\Leftrightarrow\; n_1 \equiv n_2 \approx e$$
$$(f_1, f_2, e) \in \mathcal{R}_{T' \to T} \;\Leftrightarrow\; \forall (d_1, d_2, e') \in \mathcal{R}_{T'}.\ (f_1\, d_1, f_2\, d_2, e\, e') \in \mathcal{R}_T.$$

△

Lemma 16. *(i)* $\forall (d_1, d_2, e) \in \mathcal{R}_T.\ \downarrow_T d_1 \equiv \downarrow_T d_2 \approx e$.
(ii) $\forall u_1, u_2 \in \mathit{Neu}(T), e \in \mathit{Exp}(T).\ u_1 \equiv u_2 \approx e \;\Rightarrow\; (\uparrow_T u_1, \uparrow_T u_2, e) \in \mathcal{R}_T$.

Proof. Both properties are proved simultaneously by induction on the structure of T, using the easily verified fact that if $(d_1, d_2, e) \in \mathcal{R}_T$ and $e \approx e'$, then $(d_1, d_2, e') \in \mathcal{R}_T$. □

Although we do not give the details in this extended abstract, one can prove the following properties of $\mathcal{R}$.

- "Fundamental property" for the logical relation: $\mathcal{R}$ is respected by all the syntactical constructions of Nominal System T.
- $([\![e]\!]\rho_0, [\![e']\!]\rho_0, e') \in \mathcal{R}_T$ holds for each pair of convertible expressions e and e' in Fig. 3 (T being the type of e and e').
- $\{(d_1, d_2) \mid (d_1, d_2, e) \in \mathcal{R}_T\}$ is a partial equivalence relation on $[\![T]\!]$ (for each $e \in \mathit{Exp}(T)$).

From these properties it follows that:

$$\forall e, e' \in \mathit{Exp}(T).\ e \approx e' \;\Rightarrow\; ([\![e]\!]\rho_0, [\![e']\!]\rho_0, e') \in \mathcal{R}_T. \quad (22)$$

This allows us to complete the proof of the normalization theorem:

Proof of Theorem 10. If $e \in \mathit{Exp}(T)$, then by (22) and reflexivity of $\approx$, we have $([\![e]\!]\rho_0, [\![e]\!]\rho_0, e) \in \mathcal{R}_T$; and hence by Lemma 16(i), $nf_T\, e \triangleq \downarrow_T([\![e]\!]\rho_0) \approx e$, which is property (15). If $e \approx n \in \mathit{Nf}(T)$, then by (22) again we have $([\![e]\!]\rho_0, [\![n]\!]\rho_0, n) \in \mathcal{R}_T$ and hence by Lemmas 13(i) and 16(i), $nf_T\, e \triangleq \downarrow_T([\![e]\!]\rho_0) \equiv \downarrow_T([\![n]\!]\rho_0) \equiv n$, which is property (16). □

It should be evident from the definitions in Figs 5–7 that nf_T is computable. A prototype implementation using Fresh Objective Caml [31] is available from the author's web pages. The facilities that language provides for computing with binders allows the implementation to stick quite closely to the definitions in the figures, except that the dependently typed family $[\![T]\!]$ $[T \in \mathit{Typ}]$ has to be implemented by a single, reflexive data type. A language with dependent types, such as Agda [`wiki.portal.chalmers.se/agda`] or Coq [`coq.inria.fr`], would allow an even closer fit—were those languages to have the "nominal" features of Fresh Objective Caml, or better, of Nominal System T itself. As it is, for Agda or Coq implementations, the definitions in the figures would have to be adapted to deal with our informal treatment of α-conversion in the syntax of Nominal System T, for example by using well-scoped de Bruijn indexes. It would be nice to have "nominal" versions of these dependently typed systems that provide the kind of facilities for computing with name-abstraction and name-restriction that we are considering here and which would make possible a more or less direct encoding of the definitions in Figs 5–7.

6. Representational Adequacy

In this section we will show that functions defined by the α-structural recursion principle for Λ described in the Introduction can be faithfully represented in Nominal System T. To do so, we restrict attention to *closed expressions*, by which we mean ones with no free variables.

Definition 17 (**closed expressions and normal forms**). For each $T \in \mathit{Typ}$, let $\mathit{Cexp}(T)$ denote the subset of $\mathit{Exp}(T)$ consisting of well-typed expressions of type T that have no free variables; and let $\mathit{Cnf}(T)$ denote the subset of normal forms with no free variables. △

Closed expressions may still have free atomic names, the latter being normal forms representing object-level free variables. For example, if $a \neq a'$ are distinct atomic names, then $\mathtt{L}a.\,\mathtt{V}\,a' \in \mathit{Cnf}(\mathtt{Trm})$ is a closed normal form representing the open λ-term $\lambda a.\, a' \in \Lambda$. More generally, every open λ-term is faithfully represented in $\mathit{Cnf}(\mathtt{Trm})$. To see this, note that it follows from the rules of formation in Fig. 4 and Definition 9 that neutral forms always contain at least one free variable; and hence:

- $\mathit{Cnf}(\mathtt{Trm})$ is in bijection with the set $\Lambda[\mathit{New}]$ defined in (7), that is, the λ-terms (modulo α-equivalence, of course) over a constant New; $\mathit{Cnf}(\mathbb{A})$ is in bijection with $\mathbb{A} \cup \{\mathit{New}\}$; and $\mathit{Cnf}(\mathtt{Nat})$ is in bijection with $\mathbb{N}$.
- Structural congruence at ground types (`Nat`, `Atm`, `Trm`) is the identity relation.

So we get a simple form of "representational adequacy" [22] result for the object language Λ (defined in (1)) within Nominal System T: the map $\ulcorner - \urcorner : \Lambda \to \mathit{Cnf}(\mathtt{Trm})$ satisfying

$$\left.\begin{array}{rcl} \ulcorner a \urcorner & = & \mathtt{V}\,a \\ \ulcorner t\, t' \urcorner & = & \mathtt{A}\ulcorner t \urcorner \ulcorner t' \urcorner \\ \ulcorner \lambda a.\, t \urcorner & = & \mathtt{L}a.\ulcorner t \urcorner \end{array}\right\} \quad (23)$$

(well-defined by α-structural recursion for Λ!) *gives a bijection between* Λ *and the subset of* $\mathit{Cnf}(\mathtt{Trm})$ *of closed normal forms not involving* `new` (and hence not involving any use of name-restriction). The fact that the representation (23) is so trivial is good; the "coding gap" between object- and meta-language is very small—we just have to take care with the extra normal form `new` when manipulating the object-language from within Nominal System T.

Turning next to the representation in Nominal System T of functions on Λ, first note that normalization preserves the property of being closed. For it is easy to see from the form of its definition that the normalization function is equivariant; that is, if we permute the atomic names and variables of an expression, the atomic names and variables of its normal form are correspondingly permuted:

$$nf_T(\pi \cdot e) \;=\; \pi \cdot (nf_T\, e). \quad (24)$$

So like all equivariant functions, $nf_T : \mathit{Exp}(T) \to \mathit{Nf}(T)$ has the property that the support of $nf_T\, e$ is contained in the support of e. Thus we have:

Lemma 18. *For each well-typed expression* $e \in \mathit{Exp}(T)$*, the free variables and free atomic names of its normal form* $nf_T\, e$ *are contained in those of* e*. In particular* nf_T *restricts to an equivariant function* $nf_T : \mathit{Cexp}(T) \to \mathit{Cnf}(T)$. □

Lemma 19. *(i) If* G *is a ground type,* $e \in \mathit{Cexp}(G)$ *and* $a, a' \in \mathbb{A}$*, then* $(a\ a') * e \approx (a\ a') \cdot e$*, the result of swapping* a *and* a' *in* e.
(ii) If $e \in \mathit{Cexp}(\mathtt{Trm})$ *and* $a \in \mathbb{A}$*, then* $\mathtt{L}a.\, e \approx \mathtt{L}(a, e)$*, as defined at the bottom of Fig. 1.* □

Definition 20 (**representable functions**). Given $T \in \mathit{Typ}$, let X denote the quotient nominal set $\mathit{Cexp}(T)/\equiv$ of closed normal

forms of type T modulo structural congruence. (We write $[n]$ for the equivalence class of $n \in \mathit{Cnf}(T)$.) Suppose that the functions $f_1 \in \mathbb{A} \to_{\mathrm{fs}} X$, $f_2 \in \Lambda \to_{\mathrm{fs}} \Lambda \to_{\mathrm{fs}} X \to_{\mathrm{fs}} X$ and $f_3 \in \mathbb{A} \to_{\mathrm{fs}} \Lambda \to_{\mathrm{fs}} X \to_{\mathrm{fs}} X$ are supported by the finite subset $\overline{a} \subseteq \mathbb{A}$. We say that (f_1, f_2, f_3) is *representable* by the closed expressions $e_1 \in \mathit{Cexp}(\mathtt{Atm} \to T)$, $e_2 \in \mathit{Cexp}(\mathtt{Trm} \to \mathtt{Trm} \to T \to T \to T)$ and $e_3 \in \mathit{Cexp}(\mathtt{Atm} \to \mathtt{Trm} \to T \to T)$ if the free atomic names of (e_1, e_2, e_3) are in $\overline{a}$ and

$$f_1\, a \;=\; [nf_T(e_1\, a)] \tag{25}$$

$$f_2\, t\, t'\, [n]\, [n'] \;=\; [nf_T(e_2\, \ulcorner t \urcorner\, \ulcorner t' \urcorner\, n\, n')] \tag{26}$$

$$a \notin \overline{a} \;\Rightarrow\; f_3\, a\, t\, [n] \;=\; [nf_T(e_3\, a\, \ulcorner t \urcorner\, n)] \tag{27}$$

for all $a \in \mathbb{A}$, $t, t' \in \Lambda$ and $n, n' \in \mathit{Cnf}(\mathtt{Trm})$. (The right-hand sides of these equations are in X because of Lemma 18.) $\triangle$

Recalling the "freshness condition on binders" (FCB) from the Introduction, we have:

Theorem 21 (representation of α-structural recursion). *Let (f_1, f_2, f_3) be as in the above definition and suppose f_3 satisfies (FCB). Let $f \in \Lambda \to_{\mathrm{fs}} X$ be the function defined from these functions by α-structural recursion, that is, the unique function satisfying* (2)*. If (f_1, f_2, f_3) is representable by (e_1, e_2, e_3), then f is represented by*

$$e \;\triangleq\; \lambda x.\, \mathtt{trec}\, e_1\, e_2\, e_3\, x \in \mathit{Cexp}(\mathtt{Trm} \to T) \tag{28}$$

in the sense that for all $t \in \Lambda$

$$f\, t \;=\; [nf_T(e\, \ulcorner t \urcorner)]. \tag{29}$$

Proof. By the uniqueness part of α-structural recursion, to prove (29) it suffices to show that $\lambda t \in \Lambda.\ [nf_T(e\, \ulcorner t \urcorner)]$ satisfies the recursion scheme (2) that defines f. For first clause in (2) we have $e(\mathtt{V}\, a) \approx \mathtt{trec}\, e_1\, e_2\, e_3(\mathtt{V}\, a) \approx e_1\, a$ by (28), (β) and $(\delta \mathtt{V})$; hence by (23), (17) and (25),

$$[nf_T(e\, \ulcorner a \urcorner)] = [nf_T(e\, (\mathtt{V}\, a))] = [nf_T(e_1\, a)] = f_1(a)$$

as required. The argument for the second clause in (2) is similar. For the third clause, if $a \notin \overline{a}$ then for any $t \in \Lambda$

$$\begin{aligned} a \,\#\, f_3\, a\, t\, [nf_T(e\, \ulcorner t \urcorner)] \quad & \text{by (FCB)} \\ = [nf_T(e_3\, a\, \ulcorner t \urcorner\, (nf_T(e\, \ulcorner t \urcorner)))] \quad & \text{by (27)} \\ = [nf_T(e_3\, a\, \ulcorner t \urcorner (e\, \ulcorner t \urcorner))] \quad & \text{by (15) and (17).} \end{aligned}$$

Structurally congruent normal forms have equal sets of free atomic names; and hence the support of an equivalence class $[n] \in X$ is the same as the support of any of its representatives n. Therefore we have $a \,\#\, nf_T(e_3\, a\, \ulcorner t \urcorner (e\, \ulcorner t \urcorner))$ and so by $(\nu\#)$

$$nf_T(e_3\, a\, \ulcorner t \urcorner (e\, \ulcorner t \urcorner)) \;\approx\; \nu a.\, nf_T(e_3\, a\, \ulcorner t \urcorner (e\, \ulcorner t \urcorner)). \tag{30}$$

Therefore

$$\begin{aligned} & e\, (\mathtt{L} a.\, \ulcorner t \urcorner) \\ & \quad \approx \mathtt{trec}\, e_1\, e_2\, e_3\, (\mathtt{L} a.\, \ulcorner t \urcorner) && \text{by (28) and } (\beta) \\ & \quad \approx \nu a.\, e_3\, a\, \ulcorner t \urcorner\, (\mathtt{trec}\, e_1\, e_2\, e_3\, \ulcorner t \urcorner) && \text{by } (\delta \mathtt{L}), \\ & && \text{since } a \,\#\, (e_1, e_2, e_3) \\ & \quad \approx \nu a.\, e_3\, a\, \ulcorner t \urcorner (e\, \ulcorner t \urcorner) && \text{by (28) and } (\beta) \\ & \quad \approx \nu a.\, nf_T(e_3\, a\, \ulcorner t \urcorner (e\, \ulcorner t \urcorner)) && \text{by (15)} \\ & \quad \approx nf_T(e_3\, a\, \ulcorner t \urcorner (e\, \ulcorner t \urcorner)) && \text{by (30).} \end{aligned}$$

Hence by (16) and from above we have

$$[nf_T(e\, (\mathtt{L} a.\, \ulcorner t \urcorner))] = [nf_T(e_3\, a\, \ulcorner t \urcorner (e\, \ulcorner t \urcorner))] = f_3\, a\, t\, [nf_T(e\, \ulcorner t \urcorner)]$$

as required. $\square$

Example 2, continued. Let us use Theorem 21 to prove that the expression $\mathtt{sub}$ defined in Example 4 does indeed represent capture-avoiding substitution on λ-terms, in the sense that

$$\mathtt{sub}\, a' \ulcorner t' \urcorner \ulcorner t \urcorner \;\approx\; \ulcorner t[t'/a'] \urcorner \tag{31}$$

holds for all $a' \in \mathbb{A}$ and $t, t' \in \Lambda$. Fixing a' and t', in the theorem take $T = \mathtt{Trm}$ and the functions (f_1, f_2, f_3) to be

$$\begin{aligned} f_1\, a \;&\triangleq\; \mathit{if}\ a = a'\ \mathit{then}\ [\ulcorner t' \urcorner]\ \mathit{else}\ [\ulcorner a \urcorner] \\ f_2\, t_1\, t_2\, [n_1]\, [n_2] \;&\triangleq\; [\mathtt{A}\, n_1\, n_2] \\ f_3\, a\, t_1\, [n_1] \;&\triangleq\; [\mathtt{L} a.\, n_1]. \end{aligned}$$

They are supported by the finite set $\overline{a}$ consisting of a' and the free atomic names of t'; and f_3 satisfies (FCB). Let f by the function defined from them by α-structural recursion. An easy proof by α-structural induction [26] shows that for all $t \in \Lambda$

$$f\, t \;=\; [\ulcorner t[t'/a'] \urcorner]. \tag{32}$$

It is not hard to see that (f_1, f_2, f_3) is representable (in the sense of Definition 20) by $(e_1[a'/x, \ulcorner t' \urcorner/y], e_2, e_3)$, where e_1, e_2 and e_3 are as in Example 4 (using Lemma 19(ii) to verify (27) for this particular e_3). So by definition of $\mathtt{sub}$ and by the theorem we have $f\, t = [nf_{\mathtt{Trm}}(\mathtt{sub}\, a' \ulcorner t' \urcorner \ulcorner t \urcorner)]$. Combining this with (32) yields $[\ulcorner t[t'/a'] \urcorner] = [nf_{\mathtt{Trm}}(\mathtt{sub}\, a' \ulcorner t' \urcorner \ulcorner t \urcorner)]$; and so by the Normalization Theorem 10 we do indeed have (31).

Example 3, continued. The bijection $\ulcorner - \urcorner : \mathbb{N} \cong \mathit{Cnf}(\mathtt{Nat})$ between natural numbers k and closed normal forms of type $\mathtt{Nat}$ is given by

$$\left.\begin{aligned} \ulcorner 0 \urcorner \;&=\; \mathtt{O} \\ \ulcorner k+1 \urcorner \;&=\; \mathtt{S} \ulcorner k \urcorner. \end{aligned}\right\} \tag{33}$$

As in the previous example, one can apply Theorem 21 to the definitions in Example 5 to show that

$$\mathtt{len} \ulcorner t \urcorner \;\approx\; \ulcorner |t| \urcorner \tag{34}$$

holds for all $t \in \Lambda$. Thus the expression $\mathtt{len}$ does indeed correctly represent the length function $|-|$ on λ-terms.

7. Atom-Abstraction Types

Pitts [26] develops the α-structural recursion principle for a wide class of languages involving binding operations, namely those that can be specified via a *nominal signature* [36, Definition 2.1]. In this paper so far, for sake of simplicity we have restricted attention to a single such language, the untyped λ-calculus, Λ. To extend Theorem 21 to the full range of "nominal data types", one first needs to extend Nominal System T with syntax for product types and *atom-abstraction types*. The latter correspond to the *atom-abstraction construction on nominal sets* of Gabbay and Pitts [11, Sect. 5]. This sends a nominal set X to the quotient nominal set $[\mathbb{A}]X \triangleq (\mathbb{A} \times X)/\!\sim$, where $\sim$ captures the essence of α-equivalence: $(a, x) \sim (a', x') \;\Leftrightarrow\; (a\ a'') \cdot x = (a'\ a'') \cdot x'$ for some (indeed, any) a'' such that $a'' \,\#\, (a, x, a', x')$. We write the equivalence class of (a, x) in $[\mathbb{A}]X$ as $\langle a \rangle x$.

Ever since the introduction of this construct it has been known that the elements of $[\mathbb{A}]X$ have a dual nature. On one hand they are "abstractions-as-pairs", with the identity of the atomic name a in the pair (a, x) anonymized via permutations when we pass to the equivalence class $\langle a \rangle x$. On the other hand they also represent "abstractions-as-partial-functions", since $[\mathbb{A}]X$ is isomorphic to the nominal set of those partial functions f from $\mathbb{A}$ to X whose domain of definition is $\{a \mid a \,\#\, f\}$. This bijection is mediated by the partial operation of *concretion*; if $a \,\#\, c \in [\mathbb{A}]X$, there is a unique element $c \circledcirc a \in X$ satisfying $c = \langle a \rangle (c \circledcirc a)$ and called the

- A new type-former $\texttt{[Atm]}T$ for atom-abstraction types.
- New forms of expression $\alpha a.\, e$ and $e \mathbin{@} e'$, where $\alpha a.\, (-)$ is a binder (like $\mathtt{L}a.\, (-)$) and the typing rules are:

$$\frac{a \in \mathbb{A} \qquad e \in \mathit{Exp}(T)}{\alpha a.\, e \in \mathit{Exp}(\texttt{[Atm]}T)} \qquad \frac{e \in \mathit{Exp}(\texttt{[Atm]}T) \qquad e' \in \mathit{Exp}(\texttt{Atm})}{e \mathbin{@} e' \in \mathit{Exp}(T)}$$

- New conversions:

$$\begin{array}{rll} (\alpha a.\, e) \mathbin{@} e' \approx \nu a.\, (a\ e') * e & \text{if } a \# e' & (\alpha\beta) \\ e \approx \alpha a.\, (e \mathbin{@} a) & \text{if } a \# e & (\alpha\eta) \\ \nu a.\, \alpha a'.\, e \approx \alpha a'.\, \nu a.\, e & \text{if } a \neq a' & (\nu\alpha) \\ (e_1\ e_2) * \alpha a.\, e \approx \alpha a.\, (e_1\ e_2) * e & \text{if } a \# (e_1, e_2) & (\pi\alpha) \end{array}$$

- New normal and neutral forms:

$$\frac{a \in \mathbb{A} \qquad n \in \mathit{Nf}(T)}{\alpha a.\, n \in \mathit{Nf}(\texttt{[Atm]}T)} \qquad \frac{u \in \mathit{Neu}(\texttt{[Atm]}T) \qquad n \in \mathit{Nf}(\texttt{Atm})}{u \mathbin{@} n \in \mathit{Neu}(T)}$$

Figure 8. Extension with atom-abstraction

concretion of c at a:

$$(\langle a' \rangle x) \mathbin{@} a \triangleq \begin{cases} x & \text{if } a = a' \\ (a'\ a) \cdot x & \text{if } a \neq a' \text{ and } a \# x \\ \text{undefined} & \text{otherwise.} \end{cases} \tag{35}$$

The undefinedness in the third clause is forced by the necessity of making the right-hand side independent of the choice of representative (a', x) for the equivalence class $\langle a' \rangle x$. The fact that concretion is a partial operation creates the same problem as does the freshness condition on binders (FCB) for α-structural recursion; calculating with concretions involves proving freshness conditions ($a \# c$). We have seen that name-restriction operations (Definition 1) provide a simple solution for the "FCB problem". They also provide one for the "concretion problem", as the following result shows.

Theorem 22 (atom-abstractions as total functions). *If R is a nominal set equipped with a name-restriction operation ν_R, then there is a name-restriction operation ν on the nominal set $[\mathbb{A}]R$ of atom-abstractions satisfying*

$$a \neq a' \ \Rightarrow\ (\nu a)(\langle a' \rangle r) = \langle a' \rangle((\nu_R a) r) \tag{36}$$

for all $a, a' \in \mathbb{A}$ and $r \in R$. In this case there are equivariant functions preserving name restriction

$$[\mathbb{A}]R \overset{i}{\rightarrowtail} (\mathbb{A} \to_{\mathrm{fs}} R) \overset{p}{\twoheadrightarrow} [\mathbb{A}]R$$

whose composition is the identity on $[\mathbb{A}]R$. Furthermore, the partial operation of concretion extends to a total function $(-) \mathbin{\overline{@}} (-) : [\mathbb{A}]R \times \mathbb{A} \to R$ that corresponds to function application under the inclusion $i : [\mathbb{A}]R \rightarrowtail (\mathbb{A} \to_{\mathrm{fs}} R)$. □

The proof of this result, which we omit in this extended abstract, shows that the extended concretion function $(-) \mathbin{\overline{@}} (-) : [\mathbb{A}]R \times \mathbb{A} \to R$ satisfies

$$(\langle a' \rangle r) \mathbin{\overline{@}} a = \begin{cases} r & \text{if } a = a' \\ (\nu a')((a'\ a) \cdot r) & \text{if } a \neq a'. \end{cases} \tag{37}$$

This suggests extending Nominal System T as in Fig. 8. The new form of β-conversion for atom-abstraction, $(\alpha\beta)$, is particularly pleasing, since it combines in one equation all the main players: atom-abstraction $\alpha a.\, (-)$, concretion $(-) \mathtt{@} e$, locally scoped names $\nu a.\, (-)$, and explicit swapping $(e_1\ e_2) * (-)$.

We leave to future work investigating whether Theorem 10 can be extended to encompass these features (plus appropriate ones for product types $T \times T'$). To extend Theorem 21 to the full generality of Pitts [26] one would then consider ground types equipped with a nominal signature of constructors and a suitable recursion combinator. For example, `Trm` would have constructors `V : Atm -> Trm`, `A : Trm × Trm -> Trm` and `L : [Atm]Trm -> Trm` (with $\mathtt{L}(\alpha a.\, e)$ replacing the $\mathtt{L}a.\, e$ construct). This is probably best done as part of an investigation of Martin-Löf's constructive type theory extended with atom-abstraction/concretion, locally scoped atoms, atom-equality tests and explicit swapping.

8. Context

SNTT. The "simple nominal type theory" (SNTT) of Cheney [6] is the work most closely related to the results presented in this paper. The motivation behind both works is the same: to produce a calculus combining simple type theory[3] with some of the distinctive features of the nominal sets model of names and binders, particularly atom-abstraction/concretion and the good recursive properties of the associated nominal data types. Moreover, both aim to avoid the need for freshness side-conditions while defining and computing in the calculus. Although SNTT achieves most of these aims, it is at the expense of a non-trivial type system and a lack of expressiveness. We consider each point in turn.

As far as the type system goes, SNTT uses bunched contexts containing information about object-level freshness. So the aim of avoiding freshness conditions is only partially met: terms are only meaningful in context and concretion is still partial, its well-definedness mediated by freshness conditions in the context. By contrast, Nominal System T has a completely conventional type system and all freshness conditions associated with α-equivalence have been elevated to the meta-level (in much the same way as for systems based on higher-order abstract syntax [23]).

As far as expressiveness goes, SNTT lacks name-restriction $\nu a.\, e$ and name-swapping $(e_1\ e_2) * e$. Cheney [6, Sect. 4] discusses the limitations caused by lack of a $\nu a.\, (-)$ construct. (See also Fig. 5 of that paper.) Name-restriction is an important feature of the informal meta-theory of programming languages and logics, one that demands a formalization—and of course we claim to be providing a pleasant one here. The fact that explicit name-swapping (and more generally name-permutation) is also very important for meta-theory is gradually gaining currency. One might think that name-swapping only occurs in the dynamics of nominal meta-languages and not in the semantic specifications written in those meta-language. However, note that with explicit name-swapping we can use a meta-level binder like $\mathtt{L}a.\, (-)$ to express the binary operation $\mathtt{L}(-,-)$ taking an expression e that computes an object-level variable and an expression e' that computes a piece of object-level syntax and combining them to compute an object-level binder: $\mathtt{L}(e, e') = \mathtt{L}a.\, (a\ e) * e'$ (where a is any atomic name not free in either e or e'). Example 4 provided an example of this operation in use. If we extend Nominal System T as in Sect. 7, we can define the general form of this non-binding operation for atom-abstraction which is a characteristic feature of FreshML [33][4]:

$$\texttt{<}e\texttt{>}e' \triangleq \alpha a.\, (a\ e) * e' \quad \text{where } a \# (e, e'). \tag{38}$$

For example, using this we can form the expression

$$\lambda f.\, \alpha a.\, \lambda x.\, f(\texttt{<}a\texttt{>}x) \mathbin{@} a \tag{39}$$

of type $(\texttt{[Atm]}T \texttt{->} \texttt{[Atm]}T') \texttt{->} \texttt{[Atm]}(T \texttt{->} T')$ whose denotation in the nominal sets model is the "shocking" [16, Sect. 2.5] isomor-

[3] initially, and dependent type theory in the long run

[4] Be warned, Cheney [6] uses the notation $\langle a \rangle(-)$ for the *binding* operation that is denoted here by $\alpha a.\, (-)$.

phism

$$[\mathbb{A}]R \to_{\mathrm{fs}} [\mathbb{A}]R' \cong [\mathbb{A}](R \to_{\mathrm{fs}} R') \tag{40}$$

noted by Gabbay [10, Corollary 9.6.9]. This is not expressible in SNTT: see Cheney [6, Fig. 5].

Nevertheless, SNTT is a very interesting system whose metatheory is even simpler than the one presented here. It would be interesting to investigate translating it into Nominal System T (extended as in Sect. 7); perhaps the translation of its bunched contexts might provide useful conditions in a conditional-equational calculus more expressive than the simple equational notion of conversion we have given here.

Westbrook et al. [37] use some of the ideas behind SNTT to design an extension of the calculus of inductive constructions, the type theory underlying Coq. This is certainly the right direction in which to go. However, we believe that the use of Odersky-style locally-scoped names will play just as important role for expressivity in "nominal" versions of dependently typed systems as it does here for the simply typed Nominal System T.

Focusing on binding and computation. It has become very common to use typed λ-calculus as a uniform method of representing syntax involving binding [23]. The pros and cons of this method compared with "nominal" techniques have been vigorously debated [5, 7]. In comparing systems that employ them, one should bear in mind the purpose for which they are designed: is it representation plus proof (classical or constructive), or representation plus computation (functional or logical), or both? Here the primary focus is on functional computation with representations and the analyses of Poswolsky and Schürmann [29] and Licata et al. [16] are pertinent: most previous uses of typed λ-calculus representations identify "functions-as-data" with "functions-as-computation" (Miller [17] is an early exception) and this leads to complications such as modalities [24] when trying to develop recursion and induction for higher-order abstract syntax. These authors advocate separating the two notions of function, leading to forms of locally scoped symbols in [15, 16, 29, 37] similar to the notion of atom-abstraction (α-binding) considered in the extended system of Sect. 7. This should not be confused with the notion of name-restriction (ν-binding).[5] For one thing the latter does not change the type of expressions, whereas the former does. The nominal sets model makes the difference between the two notions clear. Incidentally, Theorem 22 provides an interesting semantical insight: in the presence of name-restriction, it seems that it is consistent to regard types of "functions-as-data" as subtypes of types of "functions-as-computation".

Harper has coined the term "pronominal" for the use of locally scoped symbols as "pronouns referring to a designated binding site", contrasting it with the nominal approach to symbols as nouns with independent existence. Leaving aside the important fact that, unlike in [16, 29], *here we have a boolean-valued equality test on names*, is Nominal System T nominal or pronominal? The answer is not so clear. We are used to the idea of free variables in λ-calculus being implicitly λ-bound; in other words their "designated binding site" is an implicit top-level. In Nominal System T we can definitely think of free atomic names as having an implicit top-level designated binding site as well; but they are ν-bound rather than λ-bound. What makes this possible is the fact that in our system, like Odersky's, ν-binding commutes with λ-abstraction and tupling (see the $(\nu\lambda)$ and (νK) conversions in Fig. 3).

A characteristic of using typed λ-calculus to represent binding is that one gets substitution and β-equality "for free" in addition to renaming and α-equality. This is often seen [24] as a strength of the approach, but I am not so sure. There are very many different forms of substitution; and many forms of name-binding that have nothing to do with substitution whatsoever. The approach here is to strive for the simplest possible system providing an expressive and familiar form of recursion modulo renaming; one that makes it easy for the user to deal with the many different kinds of object-language substitution on a case-by-case basis. So compared with [15, 29], here some things are not automatic. Similarly, Licata and Harper [15] incorporate types classifying closed object-level expressions, whereas I would prefer to let the user make inductive definitions of such types in the yet-to-be-explored dependently-typed version of Nominal System T. On the other hand the use of locally-scoped symbols of any data type, rather than just at name types like `Atm` as here, seems an interesting feature of [16, 17, 29].

Nominal versus presheaf representations. The nominal sets model of names and binding has close connections with the use of certain presheaf categories to model binding [9]. Indeed the category $\mathcal{N}om$ of nominal sets and equivariant functions is a sheaf subcategory of the presheaf category $\mathcal{S}et^{\mathbb{I}}$ of functors to the category of sets from the category $\mathbb{I}$ of finite sets and injective functions. The use of name-restriction operations on nominal sets (Definition 1) brings the connection even closer. Let $\mathcal{R}es$ denote the category whose objects are nominal sets R equipped with a name-restriction operation and whose morphisms $f : R \to R'$ are equivariant functions that preserve name-restriction ($f((\nu a)r) = (\nu a)(f\,r)$). Staton [private communication] has observed the remarkable fact that *$\mathcal{R}es$ is equivalent to the presheaf category $\mathcal{S}et^{p\mathbb{I}}$, where $p\mathbb{I}$ is the category of finite sets and injective partial functions.*

9. Conclusion

Apart from the technical contributions of this paper, the main messages I want to convey are that *name-restriction is just as important as name-abstraction* when computing with binders; and that *name-restriction need not involve a computational effect*. The second is the same point as made by Odersky [21]. The new results about nominal sets (Theorems 2 and 22) lead to a very simple semantics for this form of local name. This in turn suggests combining the characteristic feature of nominal sets—name-permutations—with Odersky-style name-restriction. It might seem that the commutation of this form of local scoping with function abstraction and tupling make it too simple to be useful—I certainly thought so for many years and have vigorously pursued applications of the more common, generative kind of local name. However, as we have seen, the combination of this simple and "pure" form of locally scoped name with name-swapping is very expressive. This paper has used their combination to develop a new form of structural recursion modulo α-equivalence for total functions which has all the expressive convenience of α-structural recursion without the computationally inconvenient freshness conditions on binders. Continuing with total functions, the next obvious step is to try to extend Nominal System T with dependent types. Historically speaking, Gödel's System T was a stepping-stone on the way to Martin Löf's much more expressive treatment of recursion and induction [19]. Gödel's System T is the simply typed kernel of Martin Löf's constructive type theory. It would be interesting to investigate whether the approach introduced here extends to a "nominal Martin-Löf type theory" with Odersky-style local names and name-swapping. The motivation is the search for a logical framework [22] that admits familiar forms of "nominal" specification and formalizes the informal uses of recursion and induction "modulo α" that are common in the practice of programming language semantics.

Acknowledgments

I am very grateful for stimulating conversations on the ideas underlying this paper with James Cheney, Johan Glimming, Bob Harper,

[5] Unfortunately "ν" is used to indicate abstraction rather than restriction in [29, 37].

Paul Blain Levy, and Sam Staton. Andreas Abel and Peter Dybjer gave me good advice about normalization-by-evaluation.

References

[1] A. Abel, T. Coquand, and M. Pagano. A modular type-checking algorithm for type theory with singleton types and proof irrelevance. In P.-L. Curien, editor, *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009, Proceedings*, volume 5608 of *Lecture Notes in Computer Science*, pages 5–19. Springer-Verlag, 2009.

[2] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.

[3] U. Berger, M. Eberl, and H. Schwichtenberg. Term rewriting for normalization by evaluation. *Information and Computation*, 183:19–42, 2003.

[4] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *6th Annual Symposium on Logic in Computer Science*, pages 203–211. IEEE Computer Society Press, Washington, 1991.

[5] J. Cheney. Nominal logic and abstract syntax. *ACM SIGACT News, Logic Column*, 36(4):47–69, Dec. 2005.

[6] J. Cheney. A simple nominal type theory. In *Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP 2008)*, volume 228 of *Electronic Notes in Theoretical Computer Science*, pages 37–52. Elsevier B. V., Jan. 2009.

[7] K. Crary and R. Harper. Higher-order abstract syntax: Setting the record straight. *ACM SIGACT News, Logic Column*, 37(3):93–96, Sept. 2006.

[8] P. Dybjer and A. Filinski. Normalization and partial evaluation. In G. Barthe, P. Dybjer, and J. Saraiva, editors, *Applied Semantics, Advanced Lectures*, volume 2395 of *Lecture Notes in Computer Science, Tutorial*, pages 137–192. Springer-Verlag, 2002. ISBN 3-540-44044-5. International Summer School, APPSEM 2000, Caminha, Portugal, September 9–15, 2000.

[9] M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In *14th Annual Symposium on Logic in Computer Science*, pages 193–202. IEEE Computer Society Press, Washington, 1999.

[10] M. J. Gabbay. *A Theory of Inductive Definitions with α-Equivalence: Semantics, Implementation, Programming Language*. PhD thesis, University of Cambridge, 2000.

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

[12] A. Gacek, D. Miller, and G. Nadathur. Combining generic judgments with recursive definitions. In *23rd IEEE Symposium on Logic in Computer Science (LICS 2008)*, pages 33–44. IEEE Computer Society Press, June 2008.

[13] K. Gödel. Über eine bisher noch nicht benütze Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958.

[14] A. D. Gordon and T. Melham. Five axioms of alpha-conversion. In *Theorem Proving in Higher Order Logics, 9th International Conference*, volume 1125 of *Lecture Notes in Computer Science*, pages 173–191. Springer-Verlag, 1996.

[15] D. R. Licata and R. Harper. A universe of binding and computation. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009)*, pages 123–134. ACM Press, 2009.

[16] D. R. Licata, N. Zeilberger, and R. Harper. Focusing on binding and computation. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*, pages 241–252. IEEE Computer Society, 2008.

[17] D. A. Miller. An extension to ML to handle bound variables in data structures, in the proceedings of the logical frameworks bra workshop. Technical Report MS-CIS-90-59, University of Pennsylvania, May 1990.

[18] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(02):119–141, June 1992.

[19] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory*. Oxford University Press, 1990.

[20] M. Norrish. Recursive function definition for types with binders. In *Theorem Proving in Higher Order Logics, 17th International Conference*, volume 3223 of *Lecture Notes in Computer Science*, pages 241–256. Springer-Verlag, 2004.

[21] M. Odersky. A functional theory of local names. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 48–59. ACM Press, 1994.

[22] F. Pfenning. Logical frameworks. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 17, pages 1063–1147. Elsevier Science and MIT Press, 2001.

[23] F. Pfenning and C. Elliott. Higher-order abstract syntax. In *Proc. ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988.

[24] B. Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 371–382. ACM Press, 2008.

[25] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[26] A. M. Pitts. Alpha-structural recursion and induction. *Journal of the ACM*, 53:459–506, 2006.

[27] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Mathematical Foundations of Computer Science, Proc. 18th Int. Symp., Gdańsk, 1993*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, Berlin, 1993.

[28] A. M. Pitts and I. D. B. Stark. Operational reasoning for functions with local state. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 227–273. Cambridge University Press, 1998.

[29] A. Poswolsky and C. Schürmann. Practical programming with higher-order encodings and dependent types. In *European Symposium on Programming (ESOP 2008)*, volume 4960 of *Lecture Notes in Computer Science*, pages 93–107. Springer-Verlag, 2008.

[30] F. Pottier. Static name control for FreshML. In *Twenty-Second Annual IEEE Symposium on Logic In Computer Science (LICS'07)*, pages 356–365, Wroclaw, Poland, July 2007. IEEE Computer Society Press.

[31] M. R. Shinwell and A. M. Pitts. Fresh Objective Caml user manual. Technical Report UCAM-CL-TR-621, University of Cambridge Computer Laboratory, Feb. 2005.

[32] M. R. Shinwell and A. M. Pitts. On a monadic semantics for freshness. *Theoretical Computer Science*, 342:28–55, 2005.

[33] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 263–274. ACM Press, Aug. 2003.

[34] W. W. Tait. Intensional interpretation of functionals of finite type, I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

[35] C. Urban and S. Berghofer. A recursion combinator for nominal datatypes implemented in Isabelle/HOL. In *3rd International Joint Conference on Automated Reasoning (IJCAR 2006), Seattle, USA*, volume 4130 of *Lecture Notes in Computer Science*, pages 498–512. Springer-Verlag, 2006.

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

[37] E. Westbrook, A. Stump, and E. Austin. The calculus of nominal inductive constructions: an intensional approach to encoding name-bindings. In *Proceedings of the Fourth International Workshop on Logical Frameworks and Meta-languages: Theory and Practice (LFMTP 2009), Montreal, Canada*, ACM International Conference Proceeding Series, pages 74–83. ACM Press, Aug. 2009.