JFP **21** (3): 235–286, 2011. © Cambridge University Press 2011
doi:10.1017/S0956796811000116

Structural recursion with locally scoped names

ANDREW M. PITTS
University of Cambridge Computer Laboratory
Cambridge CB3 0FD, UK
(*e-mail:* Andrew.Pitts@cl.cam.ac.uk)

Abstract

This paper introduces a new recursion principle for inductively defined data modulo α-equivalence of bound names that makes use of Odersky-style local names when recursing over bound names. It is formulated in simply typed λ-calculus extended with names that can be restricted to a lexical scope, tested for equality, explicitly swapped and abstracted. 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 calculus has a simple interpretation in nominal sets equipped with name-restriction operations. It is shown to adequately represent α-structural recursion while avoiding the need to verify freshness side-conditions in definitions and computations. The paper is a revised and expanded version of Pitts (Nominal System T. In *Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 2010 (Madrid, Spain)*. ACM Press, pp. 159–170, 2010).

1 Introduction

1.1 Alpha-structural recursion

When giving the semantics of a programming languages, it is commonplace to rise above details of concrete syntax and work at the level of abstract syntax trees. Indeed, if the language involves binding constructs (as most do), one often 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 the fundamental tools of programming language semantics, namely the definition of functions on syntax by structural recursion and the proof of properties of them by structural induction: the arguments 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. To be specific, consider the simple example of the (possibly open) terms of the untyped λ-calculus (Barendregt, 1984)

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

where a ranges over an infinite set $\mathbb{A}$ of variables and where syntax trees for λ-terms are identified up to the usual notion of α-equivalence for λ-bound variables, $\equiv_\alpha$. When making a structurally recursive definition of a function $f : \Lambda \to X$ in terms of functions $f_1 : \mathbb{A} \to X$, $f_2 : X \times X \to X$ and $f_3 : \mathbb{A} \times X \to X$, one can take

advantage of the identification of terms up to $\equiv_\alpha$ by restricting the applicability of the recursion equation for terms of the form $\lambda a.\, t$. Thus, in the third clause of the recursion scheme

$$\left.\begin{array}{rl} & f\, a = f_1\, a \\ & f(t_1\, t_2) = f_2(f\, t_1, f\, t_2) \\ a \notin \overline{a} \;\Rightarrow\; & f(\lambda a.\, t) = f_3(a, f\, t) \end{array}\right\} \tag{2}$$

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

$$\begin{aligned} f_1\, a &\triangleq \begin{cases} t' & \text{if } a = a' \\ a & \text{if } a \neq a' \end{cases} \\ f_2(t_1, t_2) &\triangleq t_1\, t_2 \\ f_3(a, t) &\triangleq \lambda a.\, t \end{aligned}$$

and $\overline{a}$ to be the finite set consisting of a' and the free variables of t'.

For Equation (2) to specify a well-defined (and total) function on α-equivalence classes of syntax trees for λ-terms, the function f_3 has to satisfy some condition ensuring independence of the definiens $f_3(a, f\, t)$ in the third clause from choice of the bound variable a used to represent the λ-abstraction $\lambda a.\, t$ in the corresponding definiendum. In any particular case, such as in capture-avoiding substitution, formulating and proving such a condition on f_3 may be routine (and hence, in the literature, such proof obligations are often left to the reader). What is less routine is to find a condition that applies to arbitrary X, f_1, f_2, f_3 and $\overline{a}$ that not only ensures the scheme (2) always yields a well-defined and total function f, but also is not too restrictive when it comes to applications; and of course such a recursion scheme should be available not just for the example of λ-terms, but also for a wide class of languages involving binding operations.

Alpha-structural recursion (Pitts, 2006) is such a scheme and it is the starting point for the work described in this paper. It uses the theory of nominal sets and the associated concept of (finite) support, which generalizes the notion 'free name' from finitary syntax to more general mathematical structures (Gabbay & Pitts, 2002; Pitts, 2003). For example, the principle of α-structural recursion, when specialized to the set of α-equivalence classes of syntax trees for λ-terms (1), yields the following result whose proof can be extracted from Theorem 5.4 in Pitts (2006).

Theorem 1.1 (α-*Structural recursion principle for* Λ.)
For each nominal set X, the scheme (2) uniquely defines a function f provided the functions f_1, f_2 and f_3 have finite support contained in $\overline{a}$ and provided f_3 satisfies the following '*freshness condition for binders (FCB)*'

$$a \notin \overline{a} \;\Rightarrow\; (\forall x \in X)\; a \# f_3(a, x) \tag{FCB}$$

(where in general we write $a \# x$ to mean that the name a is not in the support of an element x of the nominal set X.) □

For the example of capture-avoiding substitution mentioned above X is Λ itself, which can be given the structure of a nominal set in which support coincides with the usual set of free variables of a λ-term; hence, FCB holds in this case because $f_3(a, x) = \lambda a.\, x$ and $a \# \lambda a.\, x$, since for any λ-term $x \in \Lambda$, a is not free in $\lambda a.\, x$. The Nominal package for the Isabelle proof assistant implements α-structural recursion (and more) within Isabelle/HOL (Urban & Berghofer, 2006; Urban, 2008). Experience with Nominal Isabelle suggests that, despite the need to prove lemmas about support and to prove FCB, this is a convenient and expressive formalization within higher order logic of structural recursion in the presence of binders (see `http://isabelle.in.tum.de/nominal/`).

However, the 'freshness condition for binders' means that α-structural recursion is not a convenient basis for a purely equational calculus – a desirable precondition for integrating these ideas with pure functional programming and constructive type theory (Nordström *et al.*, 1990), as opposed to higher order predicate logic as in the Isabelle/HOL system. This paper develops such a calculus of total, higher order functions with a form of structural recursion modulo α-equivalence that manages to retain 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. This distinguishes the calculus from early approaches to primitive recursion on data involving binding operations based upon higher order abstract syntax (Schürmann *et al.*, 2001). However, more recent such systems, such as Delphin (Poswolsky & Schürmann, 2008) and the system introduced by Licata *et al.* (2008), have nominal (or at least 'pronominal' Licata & Harper, 2009) features, such as types $\#T$ of names of variables of type T. What really distinguishes the calculus introduced here from these recent systems is the emphasis we place upon locally scoped names, or *name-restriction*, in addition to *name-abstraction*. These are different concepts; for one thing, the first does not change the type of the restricted entity, whereas for the second there is a change of type from entity to name-abstracted entity. We will show that the theory of nominal sets provides a setting in which the semantics of abstraction and restriction can be compared. We next explain why we consider restriction important for structural recursions that involve abstraction.

1.2 Locally scoped names

When defining and computing with recursive functions on syntax modulo α-equivalence, conditions like FCB can be avoided by making use of locally scoped names. Roughly speaking, the recursion scheme (2) is replaced by

$$\left.\begin{array}{r} f\, a = f_1\, a \\ f(t\, t') = f_2(f\, t, f\, t') \\ f(\lambda a.\, t) = \nu a.\, f'_3(a, f\, t) \end{array}\right\} \tag{3}$$

where $\nu a.\,(-)$ is a local scoping construct guaranteeing that a is not in the support of $\nu a.\, f'_3(a, x)$ for any x, thereby trivially satisfying FCB for $f_3(a, x) \triangleq \nu a.\, f'_3(a, x)$. This is not an entirely new idea; for example, in Section 4.1 in Schürmann *et al.* (2001),

the authors make use of an informal notion of local scoping when introducing their examples. What is new here is that we manage to give $va.(-)$ a formal semantics that in combination with some other constructs, such as name-swapping allows our calculus to represent functions defined by α-structural recursion.

What is the meaning of $va.(-)$? By far the most common interpretation of locally scoped names is a stateful one, using *dynamic allocation* of fresh names: $va.e$ is evaluated by augmenting the current state with a fresh name and then evaluating e with a bound to that fresh name. FreshML (Shinwell *et al.*, 2003; Shinwell & Pitts, 2005) uses this mechanism to provide recursion schemes, such as Equation (3), within the context of an ML-like impure functional programming language. 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; for example, function expressions can fail to behave extensionally: see Example 1.2 in Pitts & Stark (1998). Instead of trying to tame dynamic allocation in this context, as in Pottier (2007), here it is avoided altogether by using a version of the stateless, type-directed interpretation of $va.(-)$ from Odersky (1994). In fact, the author rediscovered Odersky's version of locally scoped names while studying properties of nominal sets, which provide a new and rather simple denotational semantics for it. Odersky's theory may seem too simple: compared with the dynamic allocation interpretation, there is no scope extrusion of local names from function arguments and no sharing of local names between components of a tuple. Nevertheless, combined with name-swapping, it produces a calculus that can represent any function defined by α-structural recursion for Λ at least, and potentially for any nominal signature in the sense of Definition 2.1 in Urban *et al.* (2004).

1.3 Structure and contributions of this paper

We give a new (and simple) semantics for Odersky-style local names based upon the notion of a *name-restriction operation* (Definition 2.6) on a nominal set. Section 2 reviews the theory of nominal sets and uses it to give a semantics for locally scoped names satisfying some basic structural properties (α-equivalence, garbage collection of unused names and invariance under reordering consecutive scopes). We show that nominal sets equipped with such a local scoping operation are closed under forming product (Section 2.3.2), function (Section 2.3.4) and name-abstraction (Section 2.3.5) nominal sets. In particular, the operation of 'concreting' a name-abstraction, which is, in general, a partial operation, naturally extends to a totally defined operation in the presence of a name-restriction operation. This opens up the possibility of a λ-calculus of total functions incorporating names and name-abstraction provided one also adds this form of locally scoped name.

This possibility is realized in Section 3, where we extend the simply typed λ-calculus over ground types for Booleans and names with name-abstraction types. The resulting *$\lambda\alpha\nu$-calculus* is one of the main contributions of this paper. We give its intended interpretation in nominal restriction sets (Section 3.2); and we develop

an extension of the usual notion of β-conversion that is sound for this semantics and that largely agrees[1] with the Odersky (1994) functional theory of local names (Section 3.3). In Section 3.4, we prove the existence and uniqueness (up to a structural congruence) of normal forms for $\lambda\alpha\nu$-calculus β-conversion. The proof uses evaluation to weak head normal form combined with a 'readback' operation, inspired by the work of Coquand (1991) and Altenkirch & Chapman (2009).

Based upon this foundation, in Section 4, we study recursion schemes involving locally scoped names. The new primitives of the $\lambda\alpha\nu$-calculus allow us to express the known nominal sets initial-algebra semantics of syntax with binders in the convenient style of FreshML (see Examples 4.3–4.6), while retaining an effect-free calculus. To illustrate this, the $\lambda\alpha\nu$-calculus is extended with ground types for untyped λ-terms and numbers, together with recursion combinators. We develop the normalization properties of the resulting *$\lambda\alpha\nu\delta$-calculus* in Section 4.2. Then, in Section 4.3, we prove that the α-structural recursion principle of Theorem 1.1 can be faithfully represented in the $\lambda\alpha\nu\delta$-calculus (Theorem 4.13).

Finally, in Sections 5 and 6, we compare these results with previous work and draw some conclusions about the use of locally scoped names for expressing recursion in the presence of binding operations.

2 Semantics of scope

Fix a countably infinite set $\mathbb{A}$, whose elements a are called (atomic) *names*. To qualify as a notion of local scoping of names, a syntactic construct $\nu a.(-)$ should have three basic properties:

$$\nu a.e \text{ equals } \nu a'.e[a'/a] \quad \text{if name } a' \text{ is not free in } \nu a.e \qquad (\nu\text{-ALPHA})$$

$$\nu a.e \text{ equals } e \quad \text{if name } a \text{ is not free in } e \qquad (\nu\text{-STRENGTHENING})$$

$$\nu a.\nu a'.e \text{ equals } \nu a'.\nu a.e \qquad (\nu\text{-EXCHANGE})$$

Here, $(-)[a'/a]$ denotes (capture-avoiding) substitution and 'equals' may mean some form of structural congruence that is imposed on the syntax, as is the case for name-restriction in the π-calculus (Milner, 1992) (which has additional 'scope extrusion' properties to do with parallel composition); or it may mean some form of semantic equality such as contextual equivalence, as is the case for the ν-calculus (Pitts & Stark, 1993) and $\lambda\nu$ (Odersky, 1994). The names 'Strengthening' and 'Exchange' used above are adopted from Section 2.3 in Gacek *et al.* (2008), which imposes these properties on the ∇-quantifier of Miller & Tiu (2005) in connection with the study of locally scoped eigenvariables and generic judgements in inductive proofs. Given its equational nature, ν-STRENGTHENING could just as well have been called 'ν-weakening'. Note that the other familiar structural property, contraction, is definitely not a property of local scoping: occurrences of a and a' in an expression e are kept distinct once scoped and we do not expect $\nu a.\nu a'.e$ to equal $\nu a''.e[a''/a, a''/a']$ in general.

[1] $\lambda\alpha\nu$-calculus semantics differs from Odersky's version by taking $\nu a.a$ to denote a value rather than being undefined in the semantics.

The theory of nominal sets provides a setting in which properties like the three above can be expressed independently of any particular syntax for expressions. We outline what we need of that theory in order to apply it to the notion of local scoping; see Section 3 in Pitts (2006) for a more leisurely account. (The original formulation of Gabbay and Pitts, 2002 was in terms of a universe of FM-sets, but subsequent experience shows that it is simpler to work with the category of nominal sets (Pitts, 2003) when possible; the relationship between the two is analogous to that between the von Neumann universe and the category of sets.)

2.1 Nominal sets

The main idea is to formulate a syntax-independent notion of 'free name' entirely in terms of the way permutations of names act on structures. Let $\mathrm{Perm}(\mathbb{A})$ denote the group of finite permutations of the set $\mathbb{A}$. Its elements are bijections $\pi : \mathbb{A} \cong \mathbb{A}$ for which $\{a \in \mathbb{A} \mid \pi(a) \neq a\}$ is finite; the group multiplication is given by function composition ($\circ$), the group identity by the identity function ($\imath$) and inverses by inverse functions (π^{-1}). A *nominal set* is a set X equipped with a $\mathrm{Perm}(\mathbb{A})$-action (written $\pi, x \mapsto \pi \cdot x$)

$$(\forall x \in X)\ \imath \cdot x = x$$
$$(\forall \pi, \pi' \in \mathrm{Perm}(\mathbb{A}), x \in X)\ \pi' \cdot (\pi \cdot x) = (\pi' \circ \pi) \cdot x$$

with respect to which every $x \in X$ is *finitely supported*. By definition this means that given x, there is a finite subset $\overline{a} \subseteq \mathbb{A}$ such that for any $a, a' \in \mathbb{A} - \overline{a}$, the permutation $(a\ a') \in \mathrm{Perm}(\mathbb{A})$ that swaps a and a' leaves x invariant: $(a\ a') \cdot x = x$. When x is finitely supported, the smallest (for inclusion) such $\overline{a}$ always exists and is called the *support* of x, written $supp(x)$.

Complementary to finite support is the notion of *freshness*: we say 'a is fresh for x' and write $a \# x$ if $a \notin supp(x)$, that is if x is supported by some finite set of names not containing a.

Example 2.1 ($\mathbb{A}$ *as a nominal set*.)
Permutations act on names by function application. With respect to this action, each $a \in \mathbb{A}$ has support $supp(a) = \{a\}$. Thus, $a \# a'$ holds iff $a \neq a'$.

Example 2.2 (Λ *as a nominal set*.)
Permutations act on syntax trees involving names by applying the permutation wherever names occur in the tree. In the case of syntax trees for λ-terms, this permutation action respects the usual notion of α-equivalence and so one gets an action on the set Λ in Equation (1). One can show that the elements of Λ are finitely supported with respect to this action. Indeed for each $t \in \Lambda$, $supp(t)$ coincides with the finite set of free variables of t.

Definition 2.3 (*Category of nominal sets*.)
The category *Nom* has nominal sets for objects. Given two such, X and Y, the morphisms $f \in Nom(X, Y)$ are *equivariant* functions, that is functions from X to Y satisfying $f(\pi \cdot x) = \pi \cdot (f\, x)$ for all $\pi \in \mathrm{Perm}(\mathbb{A})$ and $x \in X$. Composition of morphisms is given by the usual composition of functions.

The category *Nom* is known to provide a model of classical higher order logic with many interesting properties. We give the ones we need here.

2.1.1 Discrete nominal sets

Every set I becomes a nominal set if endowed with the trivial Perm($\mathbb{A}$)-action $\pi \cdot i = i$, which gives $supp(i) = \emptyset$. In case $I = 1 = \{0\}$ is a one-element set, this gives the terminal object in *Nom*. In case $I = \mathbb{B} = \{True, False\}$ is a two-element set, this gives the subobject classifier in *Nom* (which is a Boolean topos Johnstone, 2002).

2.1.2 Products of nominal sets

The product of X and Y in *Nom* is given by the Cartesian product $X \times Y = \{(x, y) \mid x \in X \wedge y \in Y\}$ endowed with the componentwise action $\pi \cdot (x, y) = (\pi \cdot x, \pi \cdot y)$, which gives $supp(x, y) = supp(x) \cup supp(y)$.

2.1.3 Coproducts of nominal sets

The coproduct of X and Y in *Nom* is given by the disjoint union $X+Y = \{(0, x) \mid x \in Y\} \cup \{(1, y) \mid y \in Y\}$ endowed with the action: $\pi \cdot (0, x) = (0, \pi \cdot x)$, $\pi \cdot (1, y) = (1, \pi \cdot y)$. This gives $supp(0, x) = supp(x)$ and $supp(1, y) = supp(y)$.

2.1.4 Exponentials of nominal sets

Given $X, Y \in Nom$, their exponential $X \to_{\mathrm{fs}} Y$ consists not of all functions f from X to Y, but just the ones that are finitely supported with respect to the usual action of permutations on functions ($\pi \cdot f = \lambda x \in X.\ \pi \cdot (f(\pi^{-1} \cdot x))$); 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 $f((a\ a') \cdot x) = (a\ a') \cdot (f\ x)$. In particular, the elements of $X \to_{\mathrm{fs}} Y$ supported by the empty set of names are precisely the equivariant functions, that is the elements of $Nom(X, Y)$.

2.2 Name-abstraction

The original motivation for nominal sets was to extend the range of induction and recursion for inductively defined sets to encompass sets quotiented by α-equivalence. Nominal sets like Λ (Example 2.2) are isomorphic in *Nom* to nominal sets inductively defined using products $X \times Y$, coproducts $X + Y$ and a *name-abstraction* construct $[\mathbb{A}]X$ for representing domains of name-binding operations. For example, $\Lambda \cong \mu X.\, \mathbb{A} + (X \times X) + [\mathbb{A}]X$ (where $\mathbb{A}$ is regarded as a nominal set as in Example 2.1).

Definition 2.4 (*The functor* $[\mathbb{A}](-) : Nom \to Nom$.)
Given $X \in Nom$, let $[\mathbb{A}]X$ denote the set of equivalence classes of pairs $(a, x) \in \mathbb{A} \times X$ with respect to the equivalence relation identifying (a, x) with (a', x') iff $(a\ a'') \cdot x = (a'\ a'') \cdot x'$ holds for some $a'' \#\ (a, x, a', x')$. We write $\langle a \rangle x$ for the equivalence class of (a, x). The Perm($\mathbb{A}$)-action on the product $\mathbb{A} \times X$ induces

an action on $[\mathbb{A}]X$ and one can show that each $\langle a\rangle x$ is finitely supported with respect to this action; indeed $supp(\langle a\rangle x) = supp(x) - \{a\}$. So $[\mathbb{A}]X$ is a nominal set and given any equivariant function $f \in Nom(X, Y)$, we get a well-defined equivariant function $[\mathbb{A}]f \in Nom([\mathbb{A}]X, [\mathbb{A}]Y)$ by defining: $([\mathbb{A}]f)(\langle a\rangle x) = \langle a\rangle(f\,x)$. This preserves identity and composition of morphisms and so makes $[\mathbb{A}](-)$ into a functor from *Nom* to itself.

Given that name-binding operations can be modeled by finitely supported functions with domain $[\mathbb{A}]X$, the following proposition explains the origin of conditions like FCB in Theorem 1.1. The proof of the proposition can be extracted from Section 5 of Gabbay & Pitts (2002).

Proposition 2.5
Given $X, Y \in Nom$ and $f \in (\mathbb{A} \times X) \to_{\mathrm{fs}} Y$, there is a unique $\hat{f} \in [\mathbb{A}]X \to_{\mathrm{fs}} Y$ satisfying $(\forall a \in \mathbb{A})\ a \# f \ \Rightarrow\ (\forall x \in X)\ \hat{f}(\langle a\rangle x) = f(a, x)$ iff

$$(\forall a \in \mathbb{A})\ a \# f \ \Rightarrow\ (\forall x \in X)\ a \# f(a, x). \tag{4}$$

In fact, Equation (4) holds of f iff there is at least one $a \in \mathbb{A}$ for which $a \# f$ and $(\forall x \in X)\ a \# f(a, x)$ hold; this fact is an example of the characteristic 'some/any' feature of the theory of nominal sets; see Theorem 3.8 in Pitts (2006).

2.3 Name-restriction

In this section, we develop the notion of a name-restriction operation on a nominal set. It will be used to model locally scoped names in the calculi we consider in Sections 3 and 4.

Definition 2.6 (*Category of nominal restriction sets.*)
A *name-restriction operation* on a nominal set X is an equivariant function in $Nom(\mathbb{A} \times X, X)$, written $(a, x) \mapsto a\backslash x$, satisfying

$$a \# a\backslash x \qquad (\backslash\text{-Alpha})$$

$$a \# x \ \Rightarrow\ a\backslash x = x \qquad (\backslash\text{-Strengthening})$$

$$a\backslash(a'\backslash x) = a'\backslash(a\backslash x) \qquad (\backslash\text{-Exchange})$$

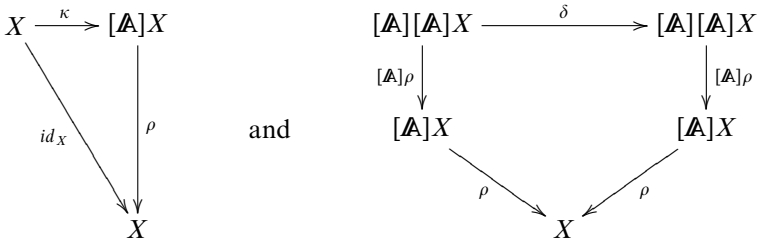
for all $a, a' \in \mathbb{A}$ and $x \in X$. We associate the operation $\_\backslash\_$ to the right and assume it binds less tightly than function application and permutation action; thus, $a\backslash a'\backslash x$ means $a\backslash(a'\backslash x)$, $a\backslash f\,x$ means $a\backslash(f\,x)$ and $a\backslash\pi \cdot x$ means $a\backslash(\pi \cdot x)$. A nominal set equipped with such an operation is called a *nominal restriction set*. We write *Res* for the category whose objects are nominal restriction sets and whose morphisms are equivariant functions preserving name-restriction ($f(a\backslash x) = a\backslash f\,x$).

Since the freshness relation # for a nominal set generalizes the syntactical 'not free in' relation (Example 2.2), it is clear that properties $\backslash$-Strengthening and $\backslash$-Exchange, respectively, model the properties ν-Strengthening and ν-Exchange required of a syntactic notion of local scoping of names; and the discussion in Section 2.2 shows why $\backslash$-Alpha models the α-equivalence property ν-Alpha. In view of Proposition 2.5,

property \-Alpha is equivalent to requiring name-restriction to induce a morphism ρ in *Nom* from $[\mathbb{A}]X$ to X satisfying

$$\rho(\langle a\rangle x) = a\backslash x. \tag{5}$$

Then, properties \-Strengthening and \-Exchange are equivalent to the commutation in *Nom* of



respectively, where $\kappa\, x = \langle a\rangle x$ for some (or indeed any) $a \# x$; and where $\delta(\langle a\rangle\langle a'\rangle x) = \langle a'\rangle\langle a\rangle x$.

Example 2.7 (*Exceptional name-restriction.*)
The monad $(-)+1 : Nom \to Nom$ (corresponding to the 'notion of computation' of Moggi, 1991 consisting of a single global exception) maps nominal sets to nominal restriction sets. Given $X \in Nom$, writing $X+1$ as $\{Some(x) \mid x \in X\} \cup \{None\}$, it is easy to see that we get a name-restriction operation on it by defining

$$a\backslash Some(x) = \begin{cases} Some(x) & \text{if } a \# x \\ None & \text{otherwise} \end{cases}$$
$$a\backslash None = None.$$

Although this might seem a rather trivial notion of name-restriction it appears to be the one that arises most often in informal practice to do with the manipulation of syntax involving binders, *viz.* a locally scoped name used in an expression whose value does not contain that name in its support. See also the '$\mathsf{export}\_$' function in the possible-world-based analysis of programming with names and binders of Pouillard & Pottier (2010).

Remark 2.8 (*Free nominal restriction sets.*)
For each nominal set X, there is a freely generated nominal restriction set $F(X)$; in other words, the forgetful functor $U : Res \to Nom$ has a left adjoint $F : Nom \to Res$. This can be described concretely as follows. Let $Fin(\mathbb{A})$ denote the set of finite subsets $\overline{a}$ of $\mathbb{A}$. It becomes a nominal set via the $\mathrm{Perm}(\mathbb{A})$-action $\pi \cdot \overline{a} = \{\pi(a) \mid a \in \overline{a}\}$, which gives $supp(\overline{a}) = \overline{a}$. Then, $F(X)$ is the set of equivalence classes of pairs $(\overline{a}, x) \in Fin(\mathbb{A}) \times X$ with respect to the equivalence relation identifying $(\overline{a}, x)$ with $(\overline{a}', x')$ if

$$supp(x) - \overline{a} = supp(x') - \overline{a}' \wedge (\exists \pi \in \mathrm{Perm}(\mathbb{A}))\ \pi \cdot x = x' \wedge (\forall a \in supp(x) - \overline{a})\ \pi(a) = a.$$

Write $(\overline{a})x$ for the equivalence class of the pair $(\overline{a}, x)$. The $\mathrm{Perm}(\mathbb{A})$-action on the product $Fin(\mathbb{A}) \times X$ induces an action on $F(X)$ and this gives a nominal set (with

$supp((\overline{a})x) = supp(x) - \overline{a}$). The name-restriction operation on $F(X)$ is given by: $a\backslash(\overline{a})x = (\{a\} \cup \overline{a})x$. There is an equivariant function $\eta \in Nom(X, F(X))$ given by $\eta(x) = (\emptyset)x$ and this has the required universal property for the free nominal restriction set on X. Since we do not need the construction in this paper, we omit further details. However, it is worth noting that whereas $F(X)$ is in general different from the nominal restriction set $X + 1$ from Example 2.7, one does have $F(\mathbb{A}) \cong \mathbb{A} + 1$.

Remark 2.9 (*Res is a topos.*)
The nominal sets model of names and binding has close connections with the use of certain presheaf categories to model binding (Fiore *et al.*, 1999). Indeed, *Nom* is equivalent to a sheaf subcategory of the presheaf category $Set^{\mathbf{I}}$ of functors to the category of sets from the category $\mathbf{I}$ of finite sets and injective functions. The use of name-restriction operations on nominal sets brings the connection with presheaf categories even closer. Staton (private communication) has observed the fact that *Res is equivalent to the presheaf category* $Set^{\mathbf{pI}}$, *where* $\mathbf{pI}$ *is the category of finite sets and injective partial functions.* It is interesting to note that the use of injective partial functions between finite sets of names already occurs in early work on logical relations for locally scoped names; see Section 3 in Pitts & Stark (1993), where the term 'partial bijection' is used for what we here call an injective partial function. Being equivalent to a presheaf category, *Res* is, in particular, a topos (Johnstone, 2002), although not a Boolean one; so it is a model of intuitionistic extensional higher order logic. However, in this paper, we will not be concerned so much with *Res* as with a related category, namely the one whose objects are nominal restriction sets, but whose morphisms are merely equivariant functions (not necessarily preserving name-restriction). This is because we will be modelling-typed functional calculi, where there is an operation of restricting a name to a local scope for each type of the calculus, but where not all functions described by the calculus commute with these operations.

2.3.1 Discrete nominal restriction sets

Every discrete nominal set (2.1.1) possesses a unique name-restriction operation, given by $a\backslash i = i$ (since $supp(i) = \emptyset$ and hence $a \# i$).

2.3.2 Products of nominal restriction sets

Given $X, Y \in Res$, the Cartesian product $X \times Y$ (2.1.2) can be given a name-restriction operation by defining

$$a\backslash(x, y) = (a\backslash x, a\backslash y).$$

(This gives the categorical product of X and Y in *Res*.)

2.3.3 Coproducts of nominal restriction sets

Given $X, Y \in \mathit{Res}$, the disjoint union $X + Y$ (2.1.3) can be given a name-restriction operation by defining

$$a\backslash(0, x) = (0, a\backslash x) \quad \text{and} \quad a\backslash(1, y) = (1, a\backslash y).$$

(This gives the categorical coproduct of X and Y in Res.)

2.3.4 Nominal restriction function sets

Theorem 2.10
For each $X \in \mathit{Nom}$ and $Y \in \mathit{Res}$, the nominal set $X \to_{\mathrm{fs}} Y$ of finitely supported functions possesses a name-restriction operation $a, f \mapsto a\backslash f$ satisfying

$$a \# x \;\Rightarrow\; (a\backslash f)\, x = a\backslash f\, x \tag{6}$$

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

Proof
We get a well-defined function $a\backslash f$ with the required properties by mapping each $x \in X$ to $a'\backslash((a\ a') \cdot f)\, x$ for some/any $a' \# (f, x)$. The proof of this involves the kind of reasoning with permutations and finite support that is typical of the theory of nominal sets; we give the details in Appendix A. □

Remark 2.11 (*Exponentials in Res*.)
We remarked above that Res is known to be a topos, and hence, in particular, it is a Cartesian closed category. If $X, Y \in \mathit{Res}$, then the name-restriction operation on $X \to_{\mathrm{fs}} Y$ given in the above theorem does not, in general, make $X \to_{\mathrm{fs}} Y$ the exponential of X and Y in Res, because function application does not commute with name-restriction, in general. In fact, one can show that the exponential in Res is given by the subobject of $X \to_{\mathrm{fs}} Y$ consisting of those finitely supported functions $f \in X \to_{\mathrm{fs}} Y$ satisfying $(\forall a \in \mathbb{A}, x \in X)\ a\backslash f\, x = a\backslash f\, (a\backslash x)$.

2.3.5 Name-abstraction for nominal restriction sets

Ever since the introduction of the name-abstraction construct for nominal sets $[\mathbb{A}]X$ (2.2), it has been known that the elements of $[\mathbb{A}]X$ have a dual nature: see Section 5 in Gabbay & Pitts (2002). On one hand, they are 'abstractions-as-pairs', with the identity of the 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 each equivalence class $\langle a\rangle x \in [\mathbb{A}]X$ is actually a partial function from $\mathbb{A}$ to X (because $\langle a\rangle x = \langle a\rangle x' \;\Rightarrow\; x = x'$) whose domain of definition is $\{a' \mid a' \# \langle a\rangle x\}$. So we get:

Definition 2.12 (*Concretion.*)
Given $X \in \mathit{Nom}$, for each $p \in [\mathbb{A}]X$ and $a \in \mathbb{A}$ with $a \# p$, there is a unique element $p @ a \in X$ satisfying $p = \langle a\rangle(p @ a)$ and called the *concretion of p at a.*

Thus concretion satisfies

$$(\langle a'\rangle x) @ a \;=\; \begin{cases} x & \text{if } a = a' \\ (a'\ a)\cdot x & \text{if } a \neq a' \text{ and } a \# x \\ \text{undefined} & \text{otherwise.} \end{cases} \tag{7}$$

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 kind of problems as does the FCB mentioned in Section 1 when it comes to formulating a typed λ-calculus with this form of name-abstraction. However, if we restrict attention to nominal sets equipped with a name-restriction operation, concretion extends to a well-behaved total operation via the following result.

Theorem 2.13 (*Name-abstractions as total functions.*)
For each $X \in Res$, the nominal set $[\mathbb{A}]X$ possesses a name-restriction operation satisfying

$$a \neq a' \;\Rightarrow\; a\backslash(\langle a'\rangle x) = \langle a'\rangle(a\backslash x) \tag{8}$$

for all $a, a' \in \mathbb{A}$ and $x \in X$. In this case, there are morphisms $m \in Res([\mathbb{A}]X, \mathbb{A}\to_{\mathrm{fs}} X)$ and $e \in Res(\mathbb{A} \to_{\mathrm{fs}} X, [\mathbb{A}]X)$ satisfying $e \circ m = id_{[\mathbb{A}]X}$. (In other words, $[\mathbb{A}]X$ is a retract of $\mathbb{A} \to_{\mathrm{fs}} X$ in the category Res when X has a name-restriction operation.)

Proof
We give the proof in Appendix B. □

Corollary 2.14
If $X \in Res$, then the partial operation of concretion (Definition 2.12) extends to a total function $\_ @ \_ \in Nom([\mathbb{A}]X \times \mathbb{A}, X)$ that corresponds to function evaluation $ev \in Nom((\mathbb{A} \to_{\mathrm{fs}} X) \times \mathbb{A}, X)$ under the monomorphism m of Theorem 2.13.

$$\begin{array}{ccc} [\mathbb{A}]X \times \mathbb{A} & & \\ \Big\downarrow{\scriptstyle m\times id} & \searrow^{\_@\_} & \\ & & X \\ & \nearrow_{ev} & \\ (\mathbb{A} \to_{\mathrm{fs}} X) \times \mathbb{A} & & \end{array} \tag{9}$$

Proof
For each $a \in \mathbb{A}$, consider $c_a \in (\mathbb{A} \times X) \to_{\mathrm{fs}} X$ given by

$$c_a(a', x) \triangleq \begin{cases} x & \text{if } a = a' \\ a'\backslash(a\ a')\cdot x & \text{if } a \neq a'. \end{cases}$$

Note that $supp(c_a) = \{a\}$ so that $a' \# c_a \;\Rightarrow\; a' \neq a \;\Rightarrow\; a' \# a'\backslash(a\ a')\cdot x = c_a(a', x)$. So we get $\hat{c}_a \in [\mathbb{A}]X \to_{\mathrm{fs}} X$ as in Proposition 2.5. Writing $p @ a$ for $\hat{c}_a\, p$, we thus have an equivariant function $\_ @ \_ \in Nom([\mathbb{A}]X \times \mathbb{A}, X)$ satisfying

$$(\langle a'\rangle x) @ a = \begin{cases} x & \text{if } a = a' \\ a'\backslash(a\ a')\cdot x & \text{if } a \neq a'. \end{cases} \tag{10}$$

Note that this agrees with Equation (7) when $a \# \langle a' \rangle x$. This definition makes Equation (9) commute, since for any $(p, a) \in [\mathbb{A}]X \times \mathbb{A}$, choosing a representative $p = \langle a' \rangle x$ with $a' \neq a$ we have

$$\begin{aligned}
& ev(m\, p, a) \\
= \quad & \{\text{definition of } ev \text{ and } m\} \\
& (a' \backslash (\lambda a'' \in \mathbb{A}.\, (a'\ a'') \cdot x))\, a \\
= \quad & \{\text{by (6), since } a \neq a'\} \\
& a' \backslash (a'\ a) \cdot x \\
= \quad & \{\text{by (10), since } a \neq a'\} \\
& p \mathbin{@} a. \quad \square
\end{aligned}$$

Example 2.15
Consider $[\mathbb{A}](X \times X)$ with X the nominal restriction set $\mathbb{A} + 1$ (Example 2.7). If $a, a' \in \mathbb{A}$ are distinct names, then the element

$$\langle a \rangle (Some(a), Some(a')) \in [\mathbb{A}](X \times X)$$

contains a' in its support, and therefore, its concretion at a' is undefined for the original notion in Definition 2.12. For the extended notion of concretion in Corollary 2.14, we have $(\langle a \rangle (Some(a), Some(a'))) \mathbin{@} a' = (Some(a'), None)$.

3 Typed λ-calculus with abstractable names

This section defines the $\lambda\alpha\nu$-calculus, a calculus for total, higher order functions with names and name-abstraction. By also including syntax for name-restriction, we are able to unbind name-abstractions using the totally defined version of concretion described in the previous section. As a result, the calculus is a straightforward extension of simply typed λ-calculus and does not need the bunched contexts with freshness assumptions used by Schöpp & Stark (2004) or Cheney (2009). The calculus provides a foundation for the form of structural recursion with locally scoped names introduced in Section 4.

3.1 Syntax of the $\lambda\alpha\nu$-calculus

The types and expressions of the $\lambda\alpha\nu$-calculus are given in Figure 1. Types are built up from ground types for Booleans (Bool) and names (Name), by forming finite product types ($T_1 \times \cdots \times T_n$), function types ($T_1 \to T_2$) and name-abstraction types ($\mathsf{Name}\ .\ T$). Expressions may involve two different kinds of identifier: *atomic names* $a \in \mathbb{A}$ and *variables* $x \in \mathbb{V}$, where $\mathbb{V}$ is a countably infinite set disjoint from $\mathbb{A}$. Variables stand for unknown expressions, whereas atomic names just stand for themselves. Both kinds of identifier may be bound; the binding forms are as follows.

- *Function abstraction*: free occurrences of x in e are bound in $\lambda x \to e$.
- *Locally scoped names*: free occurrences of a in e are bound in $\nu a.\, e$.
- *Name-abstraction*: free occurrences of a in e are bound in $\alpha a.\, e$. (N.B. many calculi based on nominal sets use a non-binding form of name-abstraction;

$T \in Typ$	$::=$	$T \to T$	function type
	$\mid$	$T \times \cdots \times T$	product type
	$\mid$	$\mathsf{Name}.T$	name-abstraction type
	$\mid$	G	ground types
G	$::=$	Bool	booleans
	$\mid$	Name	atomic names
$e \in Exp$	$::=$	x	variable ($x \in \mathbb{V}$)
	$\mid$	a	atomic name ($a \in \mathbb{A}$)
	$\mid$	$\nu a.e$	locally scoped name
	$\mid$	$\lambda x \to e$	function abstraction
	$\mid$	$e\,e$	function application
	$\mid$	$(e,\ldots,e)$	tuple
	$\mid$	$\mathsf{pr}_i\,e$	projection ($i \in \mathbb{N}$)
	$\mid$	$\alpha a.e$	name abstraction
	$\mid$	$\mathsf{let}\,\langle a\rangle x = e\,\mathsf{in}\,e$	unbinding
	$\mid$	True	truth
	$\mid$	False	falsity
	$\mid$	$e = e$	name equality
	$\mid$	$\mathsf{if}\,e\,\mathsf{then}\,e\,\mathsf{else}\,e$	conditional
	$\mid$	$(a\,\wr\,a)e$	name swapping

Derived forms:

Anon	$\triangleq$	$\nu a.a$	
$e\,@\,a'$	$\triangleq$	$\mathsf{let}\langle a\rangle x = e\,\mathsf{in}\,(a\,\wr\,a')x$	where $a \neq a'$
$\langle a\rangle e$	$\triangleq$	$\alpha a'.(a\,\wr\,a')e$	where $a' \notin fn(e)$
$\lambda\langle a\rangle x \to e$	$\triangleq$	$\lambda y \to \mathsf{let}\,\langle a\rangle x = y\,\mathsf{in}\,e$	where $y \notin fv(e)$
$\lambda(x_1,\ldots,x_m) \to e$	$\triangleq$	$\lambda y \to e[(\mathsf{pr}_1\,y)/x_1,\ldots,(\mathsf{pr}_m\,y)/x_m]$	where $y \notin fv(e)$

Fig. 1. $\lambda\alpha\nu$-Calculus types T and expressions e.

we will derive this from the binding form $\alpha a.e$ together with name-swapping in Definition 3.5.)

- *Unbinding*: free occurrences of a and x in e' are bound in $\mathsf{let}\,\langle a\rangle x = e\,\mathsf{in}\,e'$.

Following the usual informal practice *expressions are implicitly identified up to* α*-equivalence of these bound identifiers.* We also use 'Church-style' explicitly typed variables in order to simplify the presentation of the syntax.[2] Thus, we assume the countably infinite set $\mathbb{V}$ is partitioned into disjoint, countably infinite subsets $\mathbb{V}(T)$ as T ranges over Typ; 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 $T \in Typ$. Even though each $x \in \mathbb{V}$ has a unique type, for clarity we often write $\lambda x \to e$ as $\lambda x : T \to e$ when $x \in \mathbb{V}(T)$.

Notation 3.1

The finite sets of free variables and free names of an expression e are denoted $fv(e)$ and $fn(e)$, respectively. The result (well defined up to α-equivalence) of capture-avoiding substitution of e for all free occurrences of x in e' is denoted $e'[e/x]$; and

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

$$\frac{x \in \mathbb{V}(T)}{x \in Exp(T)} \qquad \frac{a \in \mathbb{A}}{a \in Exp(\mathsf{Name})} \qquad \frac{a \in \mathbb{A} \quad e \in Exp(T)}{\nu a.\, e \in Exp(T)} \qquad \frac{x \in \mathbb{V}(T_1) \quad e \in Exp(T_2)}{\lambda x \to e \in Exp(T_1 \to T_2)}$$

$$\frac{e \in Exp(T_1 \to T_2) \quad e' \in Exp(T_1)}{e\, e' \in Exp(T_2)} \qquad \frac{e_1 \in Exp(T_1) \;\cdots\; e_m \in Exp(T_m)}{(e_1, \ldots, e_m) \in Exp(T_1 \times \cdots \times T_m)} \qquad \frac{e \in Exp(T_1 \times \cdots \times T_m) \quad i \in \{1..m\}}{\mathsf{pr}_i\, e \in Exp(T_i)}$$

$$\frac{a \in \mathbb{A} \quad e \in Exp(T)}{\alpha a.\, e \in Exp(\mathsf{Name}\,.\,T)} \qquad \frac{a \in \mathbb{A} \quad x \in \mathbb{V}(T) \quad e \in Exp(\mathsf{Name}\,.\,T) \quad e' \in Exp(T')}{\mathsf{let}\,\langle a \rangle x = e\,\mathsf{in}\,e' \in Exp(T')} \qquad \frac{b \in \{\mathsf{True}, \mathsf{False}\}}{b \in Exp(\mathsf{Bool})}$$

$$\frac{e, e' \in Exp(\mathsf{Name})}{e = e' \in Exp(\mathsf{Bool})} \qquad \frac{e \in Exp(\mathsf{Bool}) \quad e', e'' \in Exp(T)}{\mathsf{if}\,e\,\mathsf{then}\,e'\,\mathsf{else}\,e'' \in Exp(T)} \qquad \frac{a, a' \in \mathbb{A} \quad e \in Exp(T)}{(a \wr a')e \in Exp(T)}$$

Fig. 2. Well-typed $\lambda\alpha\nu$ expressions, $e \in Exp(T)$ ($T \in Typ$).

similarly for simultaneous substitutions, $e'[e_1/x_1, \ldots, e_m/x_m]$. Note that if $x \in \mathbb{V}(T)$, $e \in Exp(T)$ and $e' \in Exp(T')$, then $e'[e/x] \in Exp(T')$.

Remark 3.2 (*Exp as a nominal set*.)
The usual action of name-permutations $\pi \in \mathrm{Perm}(\mathbb{A})$ on the abstract syntax trees of $\lambda\alpha\nu$-calculus expressions respects α-equivalence of bound names and so induces a $\mathrm{Perm}(\mathbb{A})$-action on Exp. Just as in Example 2.2, this makes Exp into a nominal set in which the support of each $e \in Exp$ is the finite set $fn(e)$ of free names of e. It is not hard to see that if $e \in Exp(T)$, then $\pi \cdot e \in Exp(T)$. Thus, for each $T \in Typ$, $Exp(T)$ is a nominal subset of Exp.

Note that a name-swapping expression $(a_1 \wr a_2)e$ is, in general, different from the expression $(a_1\ a_2) \cdot e$ obtained by letting the transposition $(a_1\ a_2) \in \mathrm{Perm}(\mathbb{A})$ act on the expression e. For example, when $e = x$ is a variable, $(a_1\ a_2) \cdot x = x \neq (a_1 \wr a_2)x$.

Definition 3.3 (*The 'anonymous name'*.)
The expression $\nu a.\, a \in Exp(\mathsf{Name})$ plays an important role in the $\lambda\alpha\nu$-calculus, so we give it a special name:

$$\mathsf{Anon} \triangleq \nu a.\, a. \tag{11}$$

Remark 3.4 (*Concretion versus unbinding*.)
We have chosen to use 'unbinding' $\mathsf{let}\,\langle a \rangle x = e\,\mathsf{in}\,e'$ as the elimination form for name-abstraction types. The operation of concretion (corresponding to the operation in Corollary 2.14) can be defined in terms of it, using name-swapping:

$$e \circledcirc a \triangleq \mathsf{let}\,\langle a' \rangle x = e\,\mathsf{in}\,(a' \wr a)x \quad \text{where } a' \neq a. \tag{12}$$

Thus, $e \circledcirc a \in Exp(T)$ if $e \in Exp(\mathsf{Name}\,.\,T)$. The reason for this choice is that unbinding has better properties with respect to the weak notion of expression equality (β-conversion, defined in Section 3.3) we use later in the paper when representing structural recursion with locally scoped names. However, up to the

stronger version of equality induced by the denotational semantics of Section 3.2, the two notions are interdefinable: concretion is obtained from unbinding as above and conversely, $\mathsf{let}\,\langle a\rangle x = e\,\mathsf{in}\,e'$ is equal in denotation to $\nu a.(e'[(e\,@\,a)/x])$ (where $a \notin fn(e)$). In other words, the denotation of e is matched to the pattern $\langle a\rangle x$ by concreting it at a locally scoped name; this is comparable to the use of such patterns in FreshML (see Section 3.5 in Shinwell & Pitts, 2005) except that here local scoping is not interpreted via dynamic allocation of names (see the discussion after the proof of Theorem 3.10 below).

As well as allowing us to define concretion in the $\lambda\alpha\nu$-calculus, the explicit name-swapping operation $(a\,\wr\,a')(\_)$ also gives name-abstraction subtle expressive power:

Definition 3.5 (*Non-binding name-abstraction.*)
For each $a \in \mathbb{A}$ and $e \in Exp(T)$, define $\langle a\rangle e \in Exp(\mathsf{Name}\,.\,T)$ by

$$\langle a\rangle e \;\triangleq\; \alpha a'.(a\,\wr\,a')e \qquad \text{where } a' \notin fn(e). \tag{13}$$

This is a non-binding form of name-abstraction; unlike for $\alpha a.e$, the name a occurs free in $\langle a\rangle e$. Examples 3.6 and 3.7 illustrate its use. Before giving them, we introduce some syntactic sugar that makes it easier to specify $\lambda\alpha\nu$-calculus function expressions. For each $a \in \mathbb{A}$, $x \in \mathbb{V}(T)$ and $e \in Exp(T')$, define $\lambda\langle a\rangle x \to e \in Exp((\mathsf{Name}\,.\,T) \to T')$ by

$$\lambda\langle a\rangle x \to e \;\triangleq\; \lambda y \to \mathsf{let}\,\langle a\rangle x = y\,\mathsf{in}\,e \quad \text{where } y \notin fv(e). \tag{14}$$

Given distinct variables $x_1 \in \mathbb{V}(T_1)$, …, $x_m \in \mathbb{V}(T_m)$ and $e \in Exp(T)$, define $\lambda(x_1,\ldots,x_m) \to e \in Exp((T_1 \times \cdots \times T_m) \to T)$ by

$$\lambda(x_1,\ldots,x_m) \to e \;\triangleq\; \lambda y \to e[(\mathsf{pr}_1\,y)/x_1,\ldots,(\mathsf{pr}_m\,y)/x_m] \quad \text{where } y \notin fv(e). \tag{15}$$

Example 3.6 (*Mapping over name-abstractions.*)
We will see in the next section that the intended interpretation of $\lambda\alpha\nu$-calculus types is nominal restriction sets. The action of the name-abstraction functor $[\mathbb{A}](\_)$ (Definition 2.4), when confined to nominal restriction sets, can be expressed in the $\lambda\alpha\nu$-calculus as follows:

$$\begin{aligned} mapAbs \triangleq\; & \lambda f \to \lambda\langle a\rangle x \to \langle a\rangle(f\,x) \\ & \in Exp((T \to T') \to (\mathsf{Name}\,.\,T) \to (\mathsf{Name}\,.\,T')). \end{aligned} \tag{16}$$

Example 3.7 (*'Shocking' isomorphisms.*)
The name-abstraction functor $[\mathbb{A}](\_)$ (Definition 2.4) commutes with many other nominal set constructs. For example, $[\mathbb{A}](X_1 \times X_2) \cong ([\mathbb{A}]X_1) \times ([\mathbb{A}]X_2)$ and $[\mathbb{A}](X_1 \to_{\mathrm{fs}} X_2) \cong ([\mathbb{A}]X_1) \to_{\mathrm{fs}} ([\mathbb{A}]X_2)$. (See Section 2.5 in Licata *et al.*, 2008 for the analogy to Girard's 'shocking equalities'.) The second isomorphism is quite surprising and was noted by Gabbay (2000, Corollary 9.6.9). If we confine ourselves to nominal restriction sets, then the functions that give these isomorphisms can be expressed (via the denotational semantics given in the next section) in the $\lambda\alpha\nu$-calculus as

$$\begin{array}{rcl} [\![T_1 \to T_2]\!] & = & [\![T_1]\!] \to_{\mathrm{fs}} [\![T_2]\!] \\ [\![T_1 \times \cdots \times T_m]\!] & = & [\![T_1]\!] \times \cdots \times [\![T_m]\!] \\ [\![\mathsf{Name}\,.\,T]\!] & = & [\mathbb{A}][\![T]\!] \\ [\![\mathsf{Bool}]\!] & = & \mathbb{B} \\ [\![\mathsf{Name}]\!] & = & \mathbb{A}+1 \end{array}$$

Fig. 3. Denotation of $\lambda\alpha v$ types, $[\![T]\!] \in \mathit{Res}$ ($T \in \mathit{Typ}$).

follows.

$$i \triangleq \lambda\langle a\rangle(x_1, x_2) \to (\langle a\rangle x_1, \langle a\rangle x_2) \tag{17}$$
$$\in \mathit{Exp}(\mathsf{Name}\,.\,(T_1 \times T_2) \to (\mathsf{Name}\,.\,T_1) \times (\mathsf{Name}\,.\,T_2))$$
$$j \triangleq \lambda(y_1, y_2) \to \alpha a.\,(y_1 @ a, y_2 @ a) \tag{18}$$
$$\in \mathit{Exp}((\mathsf{Name}\,.\,T_1) \times (\mathsf{Name}\,.\,T_2) \to \mathsf{Name}\,.\,(T_1 \times T_2))$$
$$k \triangleq \lambda y \to \lambda y_1 \to \alpha a.\,(y @ a)(y_1 @ a) \tag{19}$$
$$\in \mathit{Exp}(\mathsf{Name}\,.\,(T_1 \to T_2) \to (\mathsf{Name}\,.\,T_1) \to (\mathsf{Name}\,.\,T_2))$$
$$l \triangleq \lambda f \to \alpha a.\,\lambda x_1 \to (f(\langle a\rangle x_1)) @ a \tag{20}$$
$$\in \mathit{Exp}(((\mathsf{Name}\,.\,T_1) \to (\mathsf{Name}\,.\,T_2)) \to \mathsf{Name}\,.\,(T_1 \to T_2))$$

3.2 Semantics of the $\lambda\alpha v$-calculus

The types of the $\lambda\alpha v$-calculus are intended to denote nominal restriction sets (Definition 2.6). Figure 3 gives the definition of this denotational semantics. The ground type Bool stands for the discrete two-element nominal restriction set $\mathbb{B} = \{\mathit{True}, \mathit{False}\}$ (Section 2.3.1). The ground type Name stands for the nominal restriction set $\mathbb{A}+1 = \{\mathit{Some}(a) \mid a \in \mathbb{A}\} \cup \{\mathit{None}\}$ obtained from the nominal set of names as in Example 2.7. The interpretation of $\mathsf{Name}\,.\,T$ uses Theorem 2.13 to lift the name-restriction operation for $[\![T]\!]$ to the nominal set $[\mathbb{A}][\![T]\!]$ of name-abstractions. The interpretation of $T_1 \times \cdots \times T_m$ uses the name-restriction operations on each $[\![T_i]\!]$ to get one of the product $[\![T_1]\!] \times \cdots \times [\![T_m]\!]$ as in Section 2.3.2. The interpretation of $T_1 \to T_2$ uses Theorem 2.10 to lift the name-restriction operation for $[\![T_2]\!]$ to the nominal set of finitely supported functions $[\![T_1]\!] \to_{\mathrm{fs}} [\![T_2]\!]$.

The expressions of the $\lambda\alpha v$-calculus are interpreted as finitely supported functions of valuations of the following kind.

Definition 3.8 (*Valuations.*)
The nominal set *Val* consists of functions ρ on $\mathbb{V}$ satisfying

$$(\forall T \in \mathit{Typ}, x \in \mathbb{V}(T))\ \rho\, x \in [\![T]\!]$$

and which are finitely supported with respect to the usual $\mathrm{Perm}(\mathbb{A})$-action on functions. There are many such functions because each $[\![T]\!]$ contains elements with empty support (to which all variables of that type may be mapped to get an emptily supported valuation) and because valuations can be updated: if $\rho \in \mathit{Val}$, $x \in \mathbb{V}(T)$ and $d \in [\![T]\!]$, then the *updated* valuation $\rho[x \mapsto d] \in \mathit{Val}$ maps x to d and otherwise acts like ρ.

$[\![e]\!] \in Val \to_{\mathrm{fs}} [\![T]\!]$ $[T \in Typ, e \in Exp(T)]$ is defined by

$$\begin{array}{rcl}
[\![x]\!] & = & \lambda\rho \in Val.\, \rho\, x \\
[\![a]\!] & = & \lambda\rho \in Val.\, Some(a) \\
[\![\nu a.\, e]\!] & = & a\backslash[\![e]\!] \\
[\![\lambda x : T \to e]\!] & = & \lambda\rho \in Val.\, \lambda d \in [\![T]\!].\, [\![e]\!](\rho[x \mapsto d]) \\
[\![e\, e']\!] & = & \lambda\rho \in Val.\, [\![e]\!]\,\rho\,([\![e']\!]\,\rho) \\
[\![(e_1 \ldots, e_m)]\!] & = & \lambda\rho \in Val.\, ([\![e_1]\!]\,\rho, \ldots, [\![e_m]\!]\,\rho) \\
[\![\mathsf{pr}_i\, e]\!] & = & \lambda\rho \in Val.\, pr_i([\![e]\!]\,\rho) \\
[\![\alpha a.\, e]\!] & = & a\backslash(\lambda\rho \in Val.\, \langle a\rangle([\![e]\!]\rho)) \\
[\![\mathsf{let}\,\langle a\rangle x = e\,\mathsf{in}\,e']\!] & = & a\backslash(\lambda\rho \in Val.\, [\![e']\!](\rho[x \mapsto ([\![e]\!]\,\rho)\mathbin{@} a])) \\
[\![\mathsf{True}]\!] & = & \lambda\rho \in Val.\, True \\
[\![\mathsf{False}]\!] & = & \lambda\rho \in Val.\, False \\
[\![\mathsf{if}\, e\, \mathsf{then}\, e'\, \mathsf{else}\, e'']\!] & = & \lambda\rho \in Val.\, cond([\![e]\!]\,\rho, [\![e']\!]\,\rho, [\![e'']\!]\,\rho) \\
[\![e = e']\!] & = & \lambda\rho \in Val.\, eq([\![e]\!]\,\rho, [\![e']\!]\,\rho) \\
[\![(a \wr a')e]\!] & = & \lambda\rho \in Val.\, (a\ a') \cdot ([\![e]\!]\,\rho)
\end{array}$$

where

$$pr_i \in X_1 \times \cdots \times X_m \to_{\mathrm{fs}} X_i \quad \text{is} \quad pr_i(x_1, \ldots, x_m) \triangleq x_i$$

$$cond \in \mathbb{B} \times X \times X \to_{\mathrm{fs}} X \quad \text{is} \quad cond(b, x, x') \triangleq \begin{cases} x & \text{if } b = True \\ x' & \text{if } b = False \end{cases}$$

$$eq \in (\mathbb{A} + 1) \times (\mathbb{A} + 1) \to_{\mathrm{fs}} \mathbb{B} \quad \text{is} \quad eq(x, x') \triangleq \begin{cases} True & \text{if } x = x' \\ False & \text{if } x \neq x' \end{cases}$$

Fig. 4. Denotation of $\lambda\alpha\nu$ expressions.

Each $e \in Exp(T)$ determines a finitely supported function $[\![e]\!] \in Val \to_{\mathrm{fs}} [\![T]\!]$ as specified in Figure 4. Since we identify expressions up to α-equivalence, one has to check that the clauses in Figure 4 involving binding operations are independent of the choice of bound identifier. For binding atomic names ($\nu a.\,(\_)$, $\alpha a.\,(\_)$ and $\mathsf{let}\,\langle a\rangle x = e\,\mathsf{in}\,(\_)$), this follows from property $\backslash$-Alpha of name-restriction in Definition 2.6. For binding variables ($\lambda x \to (\_)$ and $\mathsf{let}\,\langle a\rangle x = e\,\mathsf{in}\,(\_)$), one can argue as on pp. 492–493 in Pitts (2006), by establishing the following properties simultaneously with the definition:

$$((\forall x \in fv(e))\ \rho(x) = \rho'(x)) \ \Rightarrow\ [\![e]\!]\rho = [\![e]\!]\rho' \tag{21}$$

$$[\![(x\ x') \cdot e]\!]\rho = [\![e]\!](\rho \circ (x\ x')) \tag{22}$$

(where $(x\ x') \cdot e$ is $e[x'/x, x/x']$ and $\rho \circ (x\ x')$ is $\rho[x \mapsto \rho(x')][x' \mapsto \rho(x)]$).

The definition of the denotation of variables, function abstraction, application, tuples, projection, Booleans, conditionals and equality test is entirely standard. The definitions of $[\![\nu a.\, e]\!]$ and $[\![\alpha a.\, e]\!]$ make use of the name-restriction operations on $Val \to_{\mathrm{fs}} [\![T]\!]$ and $Val \to_{\mathrm{fs}} [\mathbb{A}][\![T]\!]$ that they have by virtue of Theorems 2.10 and 2.13

(since each $[\![T]\!]$ is in *Res*, as we saw above). Note that from Equation (6) we have

$$[\![\nu a.\, e]\!]\,\rho = a\backslash([\![e]\!]\,\rho) \quad \text{if } a \# \rho \tag{23}$$

and also

$$[\![\alpha a.\, e]\!]\,\rho = \langle a\rangle([\![e]\!]\,\rho) \quad \text{if } a \# \rho \tag{24}$$

where we have used $\backslash$-STRENGTHENING from Definition 2.6 to simplify $a\backslash\langle a\rangle([\![e]\!]\,\rho)$. Note that given $\rho \in \mathit{Val}$, one can always satisfy the condition $a \# \rho$ on the above equations up to α-equivalence of $\nu a.\, e$ and $\alpha a.\, e$. It is instructive to compare the denotation of $\alpha a.\, e$ with the derived one for the non-binding form of name-abstraction (13); applying the definitions in Figure 4 one finds that

$$[\![\langle a\rangle e]\!]\,\rho = \langle a\rangle([\![e]\!]\,\rho) \tag{25}$$

for all ρ, whether or not $a \# \rho$ holds.

The denotation of name-swapping $(a\wr a')e$ is given by the action of the permutation $(a\ a')$ on the nominal sets $[\![T]\!]$. The denotation of unbinding makes use of the totally defined concretion operation $\_ @ a$ from Corollary 2.14. Atomic names a are interpreted as elements $\mathit{Some}(a)$ in the left-hand summand of $[\![\mathsf{Name}]\!] = \mathbb{A} + 1$. On the other hand, the unique element *None* of the right-hand summand is the denotation of the 'anonymous name' (11):

$$[\![\mathsf{Anon}]\!]\,\rho = [\![\nu a.\, a]\!]\,\rho = a\backslash \mathit{Some}(a) = \mathit{None}. \tag{26}$$

Recall from Remark 3.2 that each $\mathit{Exp}(T)$ is a nominal set. Since all the constructs involved in the definition in Figure 4 are equivariant, $[\![\_]\!]$ is itself equivariant, that is

$$\pi \cdot [\![e]\!] = [\![\pi \cdot e]\!]. \tag{27}$$

In other words, $[\![\_]\!]$ is a morphism in $\mathit{Nom}(\mathit{Exp}(T), \mathit{Val} \to_{\mathrm{fs}} [\![T]\!])$. The denotational semantics also has the usual property with respect to substitution (proved by α-structural recursion for $\lambda\alpha\nu$ expressions):

$$[\![e'[e/x]]\!]\rho = [\![e']\!](\rho[x{\to}[\![e]\!]\rho]). \tag{28}$$

3.3 $\lambda\alpha\nu$-Calculus conversion

This section gives a notion of equality for $\lambda\alpha\nu$-calculus expressions generalizing the usual notion of β-conversion for the simply typed λ-calculus and containing the structural congruences mentioned at the beginning of Section 2. The relation of *β-conversion* for the $\lambda\alpha\nu$-calculus

$$e =_\beta e' \ (T \in \mathit{Typ}, e, e' \in \mathit{Exp}(T))$$

is by definition the smallest congruence (that is equivalence relation respected by the syntactic constructs of the $\lambda\alpha\nu$-calculus) on well-typed expressions containing the basic conversions listed in Figure 5.

Structural conversions

$$\nu a.e =_\beta e \quad \text{if } a \notin fn(e) \qquad \text{(Conv-}\nu\text{-Strengthening)}$$
$$\nu a.\nu a'.e =_\beta \nu a'.\nu a.e \qquad \text{(Conv-}\nu\text{-Exchange)}$$

Scope reductions

$$\nu a.\lambda x \to e =_\beta \lambda x \to \nu a.e \qquad \text{(Conv-}\nu\text{-Fun)}$$
$$\nu a.(e_1,\ldots,e_m) =_\beta (\nu a.e_1,\ldots,\nu a.e_m) \qquad \text{(Conv-}\nu\text{-Prod)}$$
$$\nu a.\alpha a'.e =_\beta \alpha a'.\nu a.e \quad \text{if } a \neq a' \qquad \text{(Conv-}\nu\text{-Abs)}$$

Conditional reductions

$$\mathsf{if\ True\ then}\ e\ \mathsf{else}\ e' =_\beta e \qquad \text{(Conv-if-True)}$$
$$\mathsf{if\ False\ then}\ e\ \mathsf{else}\ e' =_\beta e' \qquad \text{(Conv-if-False)}$$

Equality reductions

$$n = n' =_\beta \begin{cases} \mathsf{True} & \text{if } n = n' \\ \mathsf{False} & \text{if } n \neq n' \end{cases} \quad (n,n' \in \mathbb{A} \cup \{\mathsf{Anon}\}) \qquad \text{(Conv-Eqn)}$$

Swapping reductions

$$(a_1 \wr a_2)\lambda x \to e =_\beta \lambda x \to (a_1 \wr a_2)(e[(a_1 \wr a_2)x/x]) \qquad \text{(Conv-}\pi\text{-Fun)}$$
$$(a_1 \wr a_2)(e_1,\ldots,e_m) =_\beta ((a_1 \wr a_2)e_1,\ldots,(a_1 \wr a_2)e_m) \qquad \text{(Conv-}\pi\text{-Prod)}$$
$$(a_1 \wr a_2)\alpha a.e =_\beta \alpha a.(a_1 \wr a_2)e \quad \text{if } a \neq a_1, a_2 \qquad \text{(Conv-}\pi\text{-Abs)}$$
$$(a_1 \wr a_2)n =_\beta (a_1\ a_2)\cdot n \quad \text{if } n \in \mathbb{A} \cup \{\mathsf{Anon}, \mathsf{True}, \mathsf{False}\} \qquad \text{(Conv-}\pi\text{-Gnd)}$$

β-Reductions

$$(\lambda x \to e)e' =_\beta e[e'/x] \qquad \text{(Conv-}\beta\text{-Fun)}$$
$$\mathsf{pr}_i(e_1,\ldots,e_m) =_\beta e_i \quad \text{if } i \in \{1..n\} \qquad \text{(Conv-}\beta\text{-Prod)}$$
$$\mathsf{let}\ \langle a \rangle x = \alpha a.e\ \mathsf{in}\ e' =_\beta \nu a.(e'[e/x]) \qquad \text{(Conv-}\beta\text{-Abs)}$$

Fig. 5. $\lambda\alpha\nu$-Calculus β-conversion, $e =_\beta e'$ $(T \in Typ, e, e' \in Exp(T))$.

Example 3.9

If x and x' are distinct variables of type Name, then the Boolean expression $x = x'$ is not subject to any conversion. (It is in fact a 'neutral' expression in the terminology of the next section.) By contrast if $a, a' \in \mathbb{A}$ are distinct atomic names, then by Conv-Eqn we have $(a = a') =_\beta \mathsf{False}$. Since β-conversion is a congruence, this particular conversion holds even if a and a' occur within the scope of constructs that bind atomic names, such as $\alpha a.(-)$ and $\nu a.(-)$. For example, the closed expression $\alpha a.\alpha a'.a = a'$ is β-convertible to $\alpha a.\alpha a'.\mathsf{False}$; and using Conv-$\nu$-Strengthening, we have:

$$\nu a.\nu a'.(a = a') =_\beta \mathsf{False}.$$

Contrast this with

$$((\nu a.\nu a'.a) = (\nu a.\nu a'.a')) =_\beta \mathsf{True}$$

which is also obtained from Conv-ν-Strengthening and Conv-Eqn.

Theorem 3.10 (*Soundness of β-conversion.*)
If $e =_\beta e'$, then $[\![e]\!] = [\![e']\!]$.

Proof
It follows from the form of the definition in Figure 4 that the relation $[\![\_]\!] = [\![\_]\!]$ is a congruence. So to prove that it contains $=_\beta$, it suffices to show that each of the conversions in Figure 5 is satisfied by the denotational semantics.

Structural conversions. These are satisfied because of properties \-Strengthening and \-Exchange for the nominal restriction set $Val \to_{\mathrm{fs}} [\![T]\!]$.

Scope reductions. For Conv-ν-Fun, given $\rho \in Val$ and $d \in [\![T]\!]$, up to α-equivalence, we can assume that the bound name a in both the expressions $\nu a.\, \lambda x : T \to e$ and $\lambda x : T \to \nu a.\, e$ (is the same and) satisfies $a \# (\rho, d)$; therefore

$$
\begin{aligned}
& [\![\nu a.\, \lambda x : T \to e]\!]\, \rho\, d \\
= \quad & \{\text{by (23), since } a \# \rho\} \\
& (a \backslash [\![\lambda x : T \to e]\!]\, \rho) d \\
= \quad & \{\text{by (6), since } a \# d\} \\
& a \backslash [\![\lambda x : T \to e]\!]\, \rho\, d \\
= \quad & \{\text{by definition of } [\![\_]\!]\} \\
& a \backslash [\![e]\!](\rho[x \mapsto d]) \\
= \quad & \{\text{by (23), since } a \# \rho[x \mapsto d]\} \\
& [\![\nu a.\, e]\!](\rho[x \mapsto d]) \\
= \quad & \{\text{by definition of } [\![\_]\!]\} \\
& [\![\lambda x : T \to \nu a.\, e]\!]\, \rho\, d.
\end{aligned}
$$

So we do indeed have $[\![\nu a.\, \lambda x : T \to e]\!] = [\![\lambda x : T \to \nu a.\, e]\!]$. Similarly, satisfaction of Conv-ν-Prod follows by combining Equation (23) with the way restriction is defined on products in Section 2.3.2; and Conv-ν-Abs is satisfied because of Equations (23), (24) and (8).

Equality and condition reductions. The satisfaction of these is immediate from the definition of $[\![\_]\!]$.

Swapping reductions. Satisfaction of Conv-π-Fun follows from Equation (28) and the action of permutations on functions (Section 2.1.4); Conv-π-Prod follows from the action of permutations on tuples (Section 2.1.2); Conv-π-Abs from Equation (25) and the action of permutations on name-abstractions (Definition 2.4) and Conv-π-Gnd from Equation (26) and the action of permutations on atomic names and Booleans.

β-Reductions. Satisfaction of Conv-β-Fun is a consequence of the substitution property (28); Conv-β-Prod is immediate from the definition of $[\![\_]\!]$ and Conv-β-Abs follows from Equations (23) and (24), the substitution property (28) and the property (10) of concretion for nominal restriction sets. □

Remark 3.11 (*Comparison with Odersky's λν.*)
The most subtle part of the above proof is the soundness of Conv-ν-Fun. This conversion and Conv-ν-Prod are characteristic features of the functional theory of local names of Odersky (1994); the first corresponds to his ν_λ reduction and the second to his ν_p reduction. The $\lambda\alpha\nu$-calculus was designed to fit the semantics of locally scoped names in nominal restriction sets, but turns out to agree with Odersky's

notion of local name to a large extent. However, there are differences. Nominal restriction takes the structural conversions Conv-ν-Strengthening and Conv-ν-Exchange as fundamental; whereas they are not explicit in Odersky's system, but are valid up to contextual equivalence: see Proposition 5.2 in Odersky (1994). Also, here we take a rather more 'total' 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 $\mathsf{Anon} \triangleq \nu a.\, a$ turns out to be a normal form and $\mathsf{Anon}{=}\mathsf{Anon}$ is convertible to True by the conversion Conv-Eqn. Both behaviours are different from the more common one (in Scheme, ML, Haskell, ...) based on dynamic allocation of globally fresh names; for example, the OCaml (`http://caml.inria.fr/ocaml`) version of $\mathsf{Anon} = \mathsf{Anon}$ is `ref()==ref()`, which evaluates to `false` by dynamically allocating two different names on the left- and right-hand sides of the 'shallow' equality test ==.

Remark 3.12 (*'Anti-Barendregt' convention.*)
In the expression $\mathsf{let}\,\langle a_1\rangle x = \alpha a_2.\, e_2\,\mathsf{in}\,e_1$, the scopes of the two binding atomic names a_1 and a_2 are disjoint, being e_1 and e_2, respectively. Therefore, up to α-equivalence, we may assume that a_1 and a_2 are equal. We have taken advantage of this observation when stating the conversion rule for unbinding a name-abstraction, Conv-β-Abs in Figure 5. Note that this is an instance where it is helpful to not follow the 'Barendregt Variable Convention' (Barendregt, 1984) that bound identifiers be distinct.

Remark 3.13 (*Incompleteness.*)
The definition of conversion in Figure 5 was chosen to be as weak as possible subject to the criteria that it be decidable, have relatively simple normal forms (see Section 3.4) and adequately represent α-structural recursion when extended as in Section 4. The converse of Theorem 3.10 certainly does not hold. For one thing, we have left out η-expansions:

$$[\![\lambda x \to (e\,x)]\!] = [\![e]\!] \in [\![T \to T']\!] \qquad \text{if } x \notin fv(e) \tag{29}$$

$$[\![(\mathsf{pr}_1\, e, \ldots, \mathsf{pr}_m\, e)]\!] = [\![e]\!] \in [\![T_1 \times \cdots \times T_m]\!] \tag{30}$$

$$[\![\alpha a.\, (e \mathbin{@} a)]\!] = [\![e]\!] \in [\![\mathsf{Name}\,.\,T]\!] \qquad \text{if } a \notin fn(e). \tag{31}$$

Also, we have left out identities that hold because of elementary properties of permutations; for example, $[\![\lambda x \to (a \leftrightarrow a)x]\!] = [\![\lambda x \to x]\!]$. A more interesting example of incompleteness is the fact that $[\![\nu a.\, \nu a'.\, (a \leftrightarrow a')e]\!] = [\![\nu a.\, \nu a'.\, e]\!]$, whereas $\nu a.\, \nu a'.\, (a \leftrightarrow a')e$ is not in general convertible to $\nu a.\, \nu a'.\, e$ using the definition in Figure 5. This suggests the following question.

Open Problem 3.14
Friedman (1975) proved that simply typed λ-terms have equal denotations in the full function hierarchy over a countably infinite set iff they are $\beta\eta$-convertible. Is there an analogue of this for $\lambda\alpha\nu$-calculus and its interpretation in nominal restriction sets? In other words, can one find a finite axiomatization of the relation $[\![\_]\!] = [\![\_]\!]$ of denotational equality between $\lambda\alpha\nu$ expressions. Indeed, is it a decidable relation?

$$
\begin{array}{rcl@{\qquad}rcl}
n \in Nf & ::= & \lambda x \to n & u \in Neu & ::= & x \\
& | & (n, \ldots, n) & & | & \nu a.\, u \\
& | & \alpha a.\, n & & | & u\, n \\
& | & \mathsf{True} & & | & \mathsf{pr}_i\, u \\
& | & \mathsf{False} & & | & \mathsf{let}\, \langle a \rangle x = u\, \mathsf{in}\, n \\
& | & a & & | & \mathsf{if}\, u\, \mathsf{then}\, n\, \mathsf{else}\, n \\
& | & \mathsf{Anon} & & | & u = n \\
& | & u & & | & n = u \\
& & & & | & (a \wr a) u
\end{array}
$$

$$
\begin{array}{rcll}
Nf(T) & \triangleq & \{n \in Nf \mid n \in Exp(T)\} & \\
Neu(T) & \triangleq & \{u \in Neu \mid u \in Exp(T)\} & (T \in Typ)
\end{array}
$$

Fig. 6. $\lambda\alpha\nu$-Calculus normal and neutral forms.

3.4 Normalization for $\lambda\alpha\nu$-calculus

Figure 6 defines a notion of normal form for $\lambda\alpha\nu$-calculus β-conversion. We call the elements of $Nf(T)$ the *normal forms* of type T; and the elements of the auxiliary subset $Neu(T) \subseteq Nf(T)$ are called the *neutral forms* of type T.

If $a \in \mathbb{A}$ does not occur free in $u \in Neu(T)$, then u and $\nu a.\, u$ are different elements of $Exp(T)$ that are convertible by Conv-ν-Strengthening. Similarly, so long as $a \neq a'$ then $\nu a.\, \nu a'.\, u$ and $\nu a'.\, \nu a.\, u$ are different elements of $Exp(T)$ that are convertible by Conv-ν-Exchange. 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 3.16 below that conversion restricted to normal forms coincides with the following simple notion of structural congruence.

Definition 3.15 (*Structural congruence.*)
Let $\equiv$ be the congruence on normal and neutral forms generated by

$$\nu a.\, u \;\equiv\; u \quad \text{if } a \notin fn(u) \tag{32}$$

$$\nu a.\, \nu a'.\, u \;\equiv\; \nu a'.\, \nu a.\, u \tag{33}$$

for all $a, a' \in \mathbb{A}$, $u \in Neu(T)$ and $T \in Typ$.

Theorem 3.16 ($\lambda\alpha\nu$-*Calculus normalization.*)
Each typeable $\lambda\alpha\nu$ expression is β-convertible to a normal form, which is unique up to structural congruence:

$$(\forall T \in Typ, e \in Exp(T))(\exists n \in Nf(T))\; e =_\beta n \tag{34}$$

$$(\forall T \in Typ, n, n' \in Nf(T))\; n =_\beta n' \;\Rightarrow\; n \equiv n'. \tag{35}$$

The proof of the theorem occupies the rest of this section and Appendix C. It shows that the normal form of each well-typed $\lambda\alpha\nu$ expression can be computed;

$$
\begin{array}{rcl@{\qquad}rcl}
w \in \mathit{Wnf} & ::= & \lambda x \to e & v \in \mathit{Wneu} & ::= & x \\
& | & (e, \ldots, e) & & | & \nu a.\, v \\
& | & \alpha a.\, e & & | & v\, e \\
& | & \mathsf{True} & & | & \mathsf{pr}_i v \\
& | & \mathsf{False} & & | & \mathsf{let}\, \langle a \rangle x = v\, \mathsf{in}\, e \\
& | & a & & | & \mathsf{if}\, v\, \mathsf{then}\, e\, \mathsf{else}\, e \\
& | & \mathsf{Anon} & & | & v = e \\
& | & v & & | & n = v \quad (n \in \mathbb{A} \cup \{\mathsf{Anon}\}) \\
& & & & | & (a \wr a) v
\end{array}
$$

$$
\begin{array}{rcl@{\qquad}l}
\mathit{Wnf}(T) & \triangleq & \{w \in \mathit{Wnf} \mid w \in \mathit{Exp}(T)\} & (T \in \mathit{Typ}) \\
\mathit{Wneu}(T) & \triangleq & \{v \in \mathit{Wneu} \mid v \in \mathit{Exp}(T)\} &
\end{array}
$$

Fig. 7. Weak head normal and weak neutral forms.

and since it is not hard to see that structural congruence $\equiv$ is a decidable relation, it follows that the β-conversion relation for the $\lambda\alpha\nu$-calculus is decidable too. The traditional route to such a result is via Church–Rosser and Strong Normalization properties for an oriented version of conversion. It may be that Theorem 3.16 can be proved in that way although the presence of a structural congruence is an added complication. However, here we use a method that gives a more direct 'big step', syntax-directed definition of the β-normal forms of $\lambda\alpha\nu$ expressions, inspired by Coquand (1991), Harper & Pfenning (2005) and Altenkirch & Chapman (2009).

The rules in Figure 9 inductively define a relation $e \Downarrow n$ for evaluating $\lambda\alpha\nu$ expressions e to normal forms n in two big steps. The first step $\Downarrow_{\mathrm{w}}$ finds a *weak head normal form* for expressions and the second step $\Downarrow_{\mathrm{n}}$ 'reads back' (Grégoire & Leroy, 2002) normal forms for weak head normal forms; the relation $\Downarrow$ is then the composition of $\Downarrow_{\mathrm{w}}$ with $\Downarrow_{\mathrm{n}}$. Weak head normal forms for $\lambda\alpha\nu$-calculus are defined in Figure 7 along with an associated notion of *weak neutral* form. The definition of evaluation in Figure 9 makes use of two auxiliary functions on weak head normal forms, $w \mapsto a \mathbin{\diagdown_{\mathrm{w}}} w$ and $w \mapsto (a_1\ a_2)_{\mathrm{w}} w$, that are defined in Figure 8. Both are well-defined total functions, since we identify expressions up to α-equivalence.

It is not hard to see that evaluation preserves typing:

$$e \in \mathit{Exp}(T) \wedge e \Downarrow_{\mathrm{w}} w \;\Rightarrow\; w \in \mathit{Wnf}(T) \tag{36}$$

$$w \in \mathit{Wnf}(T) \wedge w \Downarrow_{\mathrm{n}} n \;\Rightarrow\; n \in \mathit{Nf}(T). \tag{37}$$

Also, the read back relation restricts to neutral forms:

$$v \in \mathit{Wneu}(T) \wedge v \Downarrow_{\mathrm{n}} n \;\Rightarrow\; n \in \mathit{Neu}(T). \tag{38}$$

Lemma 3.17
For all $T \in \mathit{Typ}$, $e \in \mathit{Exp}(T)$ and $n, n' \in \mathit{Nf}(T)$

$$n \Downarrow n \tag{39}$$

$$e \Downarrow n \wedge e \Downarrow n' \;\Rightarrow\; n = n'. \tag{40}$$

$$a \diagdown_{\mathrm{w}} w \triangleq \begin{cases} \lambda x \to \nu a.\, e & \text{if } w = \lambda x \to e \\ (\nu a.\, e_1, \ldots, \nu a.\, e_m) & \text{if } w = (e_1, \ldots, e_m) \\ \alpha a'.\, \nu a.\, e & \text{if } w = \alpha a'.\, e \text{ where } a' \neq a \\ \mathsf{Anon} & \text{if } w = a \\ w & \text{if } w \in (\mathbb{A} - \{a\}) \cup \{\mathsf{Anon}, \mathsf{True}, \mathsf{False}\} \\ \nu a.\, v & \text{if } w = v \in \mathit{Wneu} \end{cases}$$

$$(a_1\ a_2)_{\mathrm{w}} w \triangleq \begin{cases} \lambda x \to (a_1 \wr a_2) e[(a_1 \wr a_2) x / x] & \text{if } w = \lambda x \to e \\ ((a_1 \wr a_2) e_1, \ldots, (a_1 \wr a_2) e_m) & \text{if } w = (e_1, \ldots, e_m) \\ \alpha a.\, (a_1 \wr a_2) e & \text{if } w = \alpha a.\, e \text{ where } a \neq a_1, a_2 \\ a_2 & \text{if } w = a_1 \\ a_1 & \text{if } w = a_2 \\ w & \text{if } w \in (\mathbb{A} - \{a_1, a_2\}) \cup \{\mathsf{Anon}, \mathsf{True}, \mathsf{False}\} \\ (a_1 \wr a_2) v & \text{if } w = v \in \mathit{Wneu} \end{cases}$$

Fig. 8. Auxiliary functions on weak head normal forms.

Proof

Property (39) follows from $n \Downarrow_{\mathrm{n}} n$, which is proved by induction on the structure of n. For property (40), we prove

$$\left.\begin{array}{lcl} e \Downarrow_{\mathrm{w}} w \ \wedge\ e \Downarrow_{\mathrm{w}} w' & \Rightarrow & w = w' \\ w \Downarrow_{\mathrm{n}} n \ \wedge\ w \Downarrow_{\mathrm{n}} n' & \Rightarrow & n = n' \end{array}\right\} \tag{41}$$

by induction on the derivation of $e \Downarrow_{\mathrm{w}} w$ and $w \Downarrow_{\mathrm{n}} n$ from the rules in Figure 9. The only difficulty is to deal with α-equivalence: for the rules involving binding operations, one has to show that the choice of bound identifiers does not affect the results of evaluation up to α-equivalence. First note that the rules in Figure 9 are preserved by permutations of atomic names and type-preserving permutations of variables. It follows that evaluation is preserved by such permutations. This allows one to permute bound identifiers in an induction hypothesis to sufficiently fresh ones as necessary (cf. Section 2 in Pitts, 2003). One then uses

$$\left.\begin{array}{lcl} e \Downarrow_{\mathrm{w}} w & \Rightarrow & \mathit{fn}(w) \subseteq \mathit{fn}(e) \wedge \mathit{fv}(w) \subseteq \mathit{fv}(e) \\ w \Downarrow_{\mathrm{n}} n & \Rightarrow & \mathit{fn}(n) \subseteq \mathit{fn}(w) \wedge \mathit{fv}(n) \subseteq \mathit{fv}(w) \end{array}\right\} \tag{42}$$

whose proof uses the easily verified fact that $\mathit{fn}(a \diagdown_{\mathrm{w}} w) = \mathit{fn}(w) - \{a\}$. □

Lemma 3.18

If $e \Downarrow n$, then $e =_\beta n$.

Proof

The result follows from

$$\left.\begin{array}{lcl} e \Downarrow_{\mathrm{w}} w & \Rightarrow & e =_\beta w \\ w \Downarrow_{\mathrm{n}} n & \Rightarrow & w =_\beta n \end{array}\right\} \tag{43}$$

which is proved by induction on the derivation of $e \Downarrow_{\mathrm{w}} w$ and $w \Downarrow_{\mathrm{n}} n$ using the easily verified facts that $a \diagdown_{\mathrm{w}} w =_\beta \nu a.\, w$ and $(a_1\ a_2)_{\mathrm{w}} w =_\beta (a_1 \wr a_2) w$. □

$$\frac{e \Downarrow_{\mathrm{w}} w \qquad w \Downarrow_{\mathrm{n}} n}{e \Downarrow n}$$

$$\frac{}{w \Downarrow_{\mathrm{w}} w} \qquad \frac{e \Downarrow_{\mathrm{w}} w}{\nu a.\, e \Downarrow_{\mathrm{w}} a \mathbin{\backslash}_{\mathrm{w}} w}(*) \qquad \frac{e_1 \Downarrow_{\mathrm{w}} \lambda x \to e \quad e[e_2/x] \Downarrow_{\mathrm{w}} w}{e_1\, e_2 \Downarrow_{\mathrm{w}} w} \qquad \frac{e_1 \Downarrow_{\mathrm{w}} v}{e_1\, e_2 \Downarrow_{\mathrm{w}} v\, e_2} \qquad \frac{e \Downarrow_{\mathrm{w}} (e_1, \ldots, e_m) \quad i \in \{1..m\} \quad e_i \Downarrow_{\mathrm{w}} w}{\mathsf{pr}_i\, e \Downarrow_{\mathrm{w}} w}$$

$$\frac{e \Downarrow_{\mathrm{w}} v}{\mathsf{pr}_i\, e \Downarrow_{\mathrm{w}} \mathsf{pr}_i\, v} \qquad \frac{a \notin fn(e_1) \quad x \notin fv(e_1) \quad e_1 \Downarrow_{\mathrm{w}} \alpha a.\, e \quad e_2[e/x] \Downarrow_{\mathrm{w}} w}{\mathsf{let}\ \langle a \rangle x = e_1\ \mathsf{in}\ e_2 \Downarrow_{\mathrm{w}} a \mathbin{\backslash}_{\mathrm{w}} w}(*) \qquad \frac{e_1 \Downarrow_{\mathrm{w}} v}{\mathsf{let}\ \langle a \rangle x = e_1\ \mathsf{in}\ e_2 \Downarrow_{\mathrm{w}} \mathsf{let}\ \langle a \rangle x = v\ \mathsf{in}\ e_2}$$

$$\frac{e_1 \Downarrow_{\mathrm{w}} \mathsf{True} \qquad e_2 \Downarrow_{\mathrm{w}} w}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow_{\mathrm{w}} w} \qquad \frac{e_1 \Downarrow_{\mathrm{w}} \mathsf{False} \qquad e_3 \Downarrow_{\mathrm{w}} w}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow_{\mathrm{w}} w}$$

$$\frac{e_1 \Downarrow_{\mathrm{w}} v}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow_{\mathrm{w}} \mathsf{if}\ v\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3} \qquad \frac{e_1 \Downarrow_{\mathrm{w}} n \in \mathbb{A} \cup \{\mathsf{Anon}\} \quad e_2 \Downarrow n}{e_1 = e_2 \Downarrow_{\mathrm{w}} \mathsf{True}} \qquad \frac{e_1 \Downarrow_{\mathrm{w}} n \in \mathbb{A} \cup \{\mathsf{Anon}\} \quad e_2 \Downarrow n' \in \mathbb{A} \cup \{\mathsf{Anon}\} \quad n \neq n'}{e_1 = e_2 \Downarrow_{\mathrm{w}} \mathsf{False}}$$

$$\frac{e_1 \Downarrow_{\mathrm{w}} n \in \mathbb{A} \cup \{\mathsf{Anon}\} \quad e_2 \Downarrow v}{e_1 = e_2 \Downarrow_{\mathrm{w}} n = v} \qquad \frac{e_1 \Downarrow_{\mathrm{w}} v}{e_1 = e_2 \Downarrow_{\mathrm{w}} v = e_2} \qquad \frac{e \Downarrow_{\mathrm{w}} w}{(a_1 \wr a_2) e \Downarrow_{\mathrm{w}} (a_1\, a_2)_{\mathrm{w}} w}(*)$$

$$\frac{e \Downarrow n}{\lambda x \to e \Downarrow_{\mathrm{n}} \lambda x \to n} \qquad \frac{e_1 \Downarrow n_1 \quad \cdots \quad e_m \Downarrow n_m}{(e_1, \ldots, e_m) \Downarrow_{\mathrm{n}} (n_1, \ldots, n_m)} \qquad \frac{e \Downarrow n}{\alpha a.\, e \Downarrow_{\mathrm{n}} \alpha a.\, n}$$

$$\frac{n \in \mathbb{V} \cup \mathbb{A} \cup \{\mathsf{Anon}, \mathsf{True}, \mathsf{False}\}}{n \Downarrow_{\mathrm{n}} n} \qquad \frac{v \Downarrow_{\mathrm{n}} u}{\nu a.\, v \Downarrow_{\mathrm{n}} \nu a.\, u} \qquad \frac{v \Downarrow_{\mathrm{n}} u \quad e \Downarrow n}{v\, e \Downarrow_{\mathrm{n}} u\, n} \qquad \frac{v \Downarrow_{\mathrm{n}} u}{\mathsf{pr}_i\, v \Downarrow_{\mathrm{n}} \mathsf{pr}_i\, u}$$

$$\frac{v \Downarrow_{\mathrm{n}} u \qquad e \Downarrow n}{\mathsf{let}\ \langle a \rangle x = v\ \mathsf{in}\ e \Downarrow_{\mathrm{n}} \mathsf{let}\ \langle a \rangle x = u\ \mathsf{in}\ n} \qquad \frac{v \Downarrow_{\mathrm{n}} u \qquad e \Downarrow n \qquad e' \Downarrow n'}{\mathsf{if}\ v\ \mathsf{then}\ e\ \mathsf{else}\ e' \Downarrow_{\mathrm{n}} \mathsf{if}\ u\ \mathsf{then}\ n\ \mathsf{else}\ n'} \qquad \frac{v \Downarrow_{\mathrm{n}} u \quad e \Downarrow n}{v = e \Downarrow_{\mathrm{n}} u = n}$$

$$\frac{n \in \mathbb{A} \cup \{\mathsf{Anon}\} \quad v \Downarrow_{\mathrm{n}} u}{n = v \Downarrow_{\mathrm{n}} n = u} \qquad \frac{v \Downarrow_{\mathrm{n}} u}{(a_1 \wr a_2) v \Downarrow_{\mathrm{n}} (a_1 \wr a_2) u}$$

(∗) these rules use the auxiliary functions on weak head normal forms defined in Fig. 8.

Fig. 9. Evaluation to normal form, $e \Downarrow n$.

Proposition 3.19
Define

$$e \Downarrow\!\equiv\!\Downarrow e' \triangleq (\exists n, n')\ e \Downarrow n \ \wedge\ e' \Downarrow n' \ \wedge\ n \equiv n'. \tag{44}$$

Then, for all $T \in Typ$ and $e, e' \in Exp(T)$, $e =_\beta e'$ implies $e \Downarrow\!\equiv\!\Downarrow e'$.

Proof
The proof uses a suitable logical relation and is given in Appendix C. □

We can now complete the proof of $\lambda\alpha\nu$-calculus normalization.

Proof of Theorem 3.16.
Since $=_\beta$ is by definition a reflexive relation, for each $e \in Exp(T)$ by Proposition 3.19, we have that $e \Downarrow n$ holds for some n; and so by Lemma 3.18, property (34) holds. For property (35), if $n, n' \in Nf(T)$ satisfy $n =_\beta n'$, then by Proposition 3.19, we have $n \Downarrow\equiv\Downarrow n'$; but by Lemma 3.17, this is equivalent to $n \equiv n'$. □

Remark 3.20 (*Machine-checked proof.*)
The various lemmas and propositions in this section and Appendix C give the proof of Theorem 3.16 in enough detail to convince the reader of its truth, one hopes. Nevertheless, the nature of what the theorem asserts depends very delicately upon the particular definitions in Figures 5–8. Mistakes in, or changes to (cf. Remark 3.13), those definitions could easily invalidate either the theorem, or some details of our proof of it, in a way that might well be hard to spot without checking all the details that are elided here. Furthermore, in the next section, we extend the $\lambda\alpha v$-calculus with a form of structural recursion and will have to replay the proof of normalization for this extended calculus.

For all these reasons, it would be very desirable to have a fully formalized and machine-checked proof of the results in this section in a form that is amenable to extensions. Of the interactive theorem proving systems currently available, Nominal Isabelle (`http://isabelle.in.tum.de/nominal/`) seems most suited to this task. For one thing, Isabelle's Nominal package provides support for the kind of reasoning about permutations and freshness of atomic names in inductive definitions that we used in the proof. Furthermore, Isabelle itself admits a maintainable style of structured proof via its Isar mode. Systems, such as Nominal Isabelle, have yet to reach a level of usability to make the task of formalizing a proof of Theorem 3.16 either simple or quick and no such development has yet been carried out. However, work, such as Urban *et al.* (2011), shows both the feasibility and usefulness of doing so.

4 Recursion with locally scoped names

In this section, we extend $\lambda\alpha v$-calculus with expressions for recursively defined functions on the terms (modulo α-equivalence) of languages with binding operations. The aim is to use the local scoping construct $va.(\_)$ to give a syntactic version of α-structural recursion in a simply typed λ-calculus, without the need for explicit freshness judgements. Pitts (2006) develops a semantic α-structural recursion principle for a wide class of languages involving binding operations, namely those that can be specified via a 'nominal signature'; see Definition 2.1 in Urban *et al.* (2004). We can express any such signature in the $\lambda\alpha v$-calculus using a combination of ground, product and name-abstraction types for the signature's sorts.[3] However, for the sake of simplicity, we just consider one such signature, for the untyped λ-calculus, plus the usual signature for natural numbers.

[3] More precisely, one can express any nominal signature with a single sort of names; but the extension of $\lambda\alpha v$-calculus with many sorts of name is straightforward.

Types and expressions:

$$\begin{array}{lcll} G & ::= & \cdots & \text{ground types} \\ & | & \mathsf{Lam} & \text{object-level } \lambda\text{-terms modulo } \alpha\text{-equivalence} \\ & | & \mathsf{Nat} & \text{natural numbers} \\ e & ::= & \cdots & \text{expressions} \\ & | & K\,e & \text{data constructs } (K \in \{\mathsf{V},\mathsf{A},\mathsf{L},\mathsf{Z},\mathsf{S}\}) \\ & | & \mathsf{lrec}\,e\,e\,e\,e & \mathsf{Lam} \text{ recursion} \\ & | & \mathsf{nrec}\,e\,e\,e & \mathsf{Nat} \text{ recursion} \end{array}$$

Signature:

$$\begin{array}{ll} \mathsf{V} : \mathsf{Name} \to \mathsf{Lam} & \text{object-level variables} \\ \mathsf{A} : (\mathsf{Lam} \times \mathsf{Lam}) \to \mathsf{Lam} & \text{object-level application} \\ \mathsf{L} : (\mathsf{Name}\,.\,\mathsf{Lam}) \to \mathsf{Lam} & \text{object-level } \lambda\text{-abstraction} \\ \mathsf{Z} : 1 \to \mathsf{Nat} & \text{zero} \\ \mathsf{S} : \mathsf{Nat} \to \mathsf{Nat} & \text{successor} \end{array}$$

Typing:

$$\frac{K : T \to G \quad e \in \mathit{Exp}(T)}{K\,e \in \mathit{Exp}(G)} \qquad \frac{\begin{array}{c} e_1 \in \mathit{Exp}(\mathsf{Name} \to T) \\ e_2 \in \mathit{Exp}((T \times T) \to T) \\ e_3 \in \mathit{Exp}((\mathsf{Name}\,.\,T) \to T) \\ e \in \mathit{Exp}(\mathsf{Lam}) \end{array}}{\mathsf{lrec}\,e_1\,e_2\,e_3\,e \in \mathit{Exp}(T)} \qquad \frac{\begin{array}{c} e_1 \in \mathit{Exp}(1 \to T) \\ e_2 \in \mathit{Exp}(T \to T) \\ e \in \mathit{Exp}(\mathsf{Nat}) \end{array}}{\mathsf{nrec}\,e_1\,e_2\,e \in \mathit{Exp}(T)}$$

Conversion:

Scope reductions:

$$\nu a.\,(K\,e) \;=_\beta\; K(\nu a.\,e) \qquad \text{(Conv-}\nu\text{-Data)}$$

Swapping reductions:

$$(a_1 \wr a_2)(K\,e) \;=_\beta\; K((a_1 \wr a_2)e) \qquad \text{(Conv-}\pi\text{-Data)}$$

δ-*Reductions*:

$$\begin{array}{rcll} \mathsf{lrec}\,e_1\,e_2\,e_3\,(\mathsf{V}\,e) & =_\beta & e_1\,e & \text{(Conv-}\delta\text{-V)} \\ \mathsf{lrec}\,e_1\,e_2\,e_3\,(\mathsf{A}(e,e')) & =_\beta & e_2\,(\mathsf{lrec}\,e_1\,e_2\,e_3\,e, \mathsf{lrec}\,e_1\,e_2\,e_3\,e') & \text{(Conv-}\delta\text{-A)} \\ \mathsf{lrec}\,e_1\,e_2\,e_3\,(\mathsf{L}\alpha a.\,e) & =_\beta & e_3\,\alpha a.\,\mathsf{lrec}\,e_1\,e_2\,e_3\,e \quad \text{if } a \notin \mathit{fn}(e_1,e_2,e_3) & \text{(Conv-}\delta\text{-L)} \\ \mathsf{nrec}\,e_1\,e_2\,(\mathsf{Z}()) & =_\beta & e_1\,() & \text{(Conv-}\delta\text{-Z)} \\ \mathsf{nrec}\,e_1\,e_2\,(\mathsf{S}\,e) & =_\beta & e_2\,(\mathsf{nrec}\,e_1\,e_2\,e) & \text{(Conv-}\delta\text{-S)} \end{array}$$

Fig. 10. Extending $\lambda\alpha\nu$-calculus with λ-terms and numbers.

4.1 $\lambda\alpha\nu\delta$*-Calculus*

The $\lambda\alpha\nu\delta$*-calculus* extends the calculus of Section 3 with data for untyped λ-terms and numbers. Its syntax, typing and conversions are given in Figure 10. (In the figure, $1 \in \mathit{Typ}$ denotes the $m = 0$ case of $T_1 \times \cdots \times T_m$.)

To keep things simple, Figure 10 only specifies combinators lrec and nrec for a simple, iterative form of recursion, rather than full primitive recursion for the ground types Lam and Nat. The conversion Conv-δ-L for iterating under an object-level

λ-abstraction makes implicit use of locally scoped names because of the conversion rule for name-abstractions, Conv-β-Abs in Figure 5. This becomes clearer once we introduce some useful syntactic sugar for iteratively defined functions on Lam, making use of the name-abstraction and tuple patterns introduced in Equations (14) and (15). Experience with FreshML (Shinwell *et al.*, 2003) shows that the use of name-abstraction patterns is very convenient for expressing recursion under binders.

Definition 4.1 (*Syntactic sugar for* Lam*-recursions.*)

$$\mathsf{rec}\begin{pmatrix} f : \mathsf{Lam} \to T \\ f(\mathsf{V}\,x) = e_1 \\ f(\mathsf{A}(y_1\,y_2)) = e_2 \\ f(\mathsf{L}(\langle a\rangle y)) = e_3 \end{pmatrix} \triangleq \lambda y' \to \mathsf{lrec}\,(\lambda x \to e_1)\,(\lambda(y_1, y_2) \to e_2')\,(\lambda\langle a\rangle y \to e_3')\,y' \quad (45)$$

where f, x, y_1, y_2, y, y' are distinct variables, $y' \notin fv(e_1, e_2, e_3)$ and

- e_1 contains no free occurrence of f;
- e_2 only contains free occurrences of f in subexpressions of the form $f\,y_1$ and $f\,y_2$; and e_2' is the result of replacing those subexpressions with y_1 and y_2, respectively;
- e_3 only contains free occurrences of f in subexpressions of the form $f\,y$; and e_3' is the result of replacing those subexpressions with y.

Using the typing and conversion rules in Figures 2, 5 and 10, we get the following result.

Theorem 4.2 (*Recursion with locally scoped names.*)
Suppose $f \in \mathbb{V}(\mathsf{Lam} \to T)$, $x \in \mathbb{V}(\mathsf{Name})$, $y_1, y_2, y, y' \in \mathbb{V}(\mathsf{Lam})$, $a \in \mathbb{A}$ and $e_1, e_2, e_3 \in Exp(\mathsf{Lam})$ satisfy the conditions given in Definition 4.1. Writing r for the $\lambda\alpha\nu\delta$ expression defined in Equation (45), then $r \in Exp(\mathsf{Lam} \to T)$ and for all $e \in Exp(\mathsf{Name})$, $e', e'' \in Exp(\mathsf{Lam})$ and $a' \in \mathbb{A}$, we have

$$\left.\begin{array}{rll} r\,(\mathsf{V}\,e) & =_\beta & e_1[e/x] \\ r\,(\mathsf{A}(e', e'')) & =_\beta & e_2'[r\,e/y_1,\, r\,e''/y_2] \\ r\,(\mathsf{L}\alpha a'.\,e') & =_\beta & \nu a'.\,e_3'[r\,e'/y] \qquad \text{if } a' \notin fn(e_1, e_2). \end{array}\right\} \quad (46)$$

This is precisely the kind of recursion scheme alluded to in Section 1.2 that uses locally scoped names when recursing under a binder. It is interesting to compare it with the informal use of ν in Section 4.1 in Schürmann *et al.* (2001) when motivating specific examples of that paper's recursion scheme for higher order abstract syntax; we give our version of some of their examples below.

Example 4.3 (*Substitution.*)
Define

$$sub \triangleq \lambda x \to \lambda y \to \mathsf{rec}\begin{pmatrix} s : \mathsf{Lam} \to \mathsf{Lam} \\ s(\mathsf{V}\,x') = \mathsf{if}\ x' = x\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{V}\,x' \\ s(\mathsf{A}(y_1, y_2)) = \mathsf{A}(s\,y_1, s\,y_2) \\ s(\mathsf{L}(\langle a\rangle y_1)) = \mathsf{L}(\langle a\rangle(s\,y_1)) \end{pmatrix}$$

$$\in Exp(\mathsf{Name} \to \mathsf{Lam} \to \mathsf{Lam} \to \mathsf{Lam}).$$

Here, $\langle a\rangle(s\,y_1) \triangleq \alpha a'.(a \between a')(s\,y_1)$ is an instance of the non-binding form of name-abstraction introduced in Definition 3.5. From Equation (46), we have that *sub* satisfies

$$sub\ a\ e\ (\mathsf{V}\,e') =_\beta \text{if } e' = a \text{ then } e \text{ else } \mathsf{V}\,e' \tag{47}$$

$$sub\ a\ e\ (\mathsf{A}(e_1, e_2)) =_\beta \mathsf{A}(sub\ a\ e\ e_1, sub\ a\ e\ e_2) \tag{48}$$

$$sub\ a\ e\ (\mathsf{L}(\alpha a_1.\, e_1)) =_\beta \nu a_1.\, \mathsf{L}(\langle a_1\rangle(sub\ a\ e\ e_1)) \qquad \text{if } a_1 \notin fn(a, e) \tag{49}$$

We claim that $sub\ a\ e\ e'$ represents the capture-avoiding substitution of the λ-term represented by e for free occurrences of the object-level variable $\mathsf{V}\,a$ in the λ-term represented by e'. For example, $\mathsf{L}(\alpha a'.\,\mathsf{V}\,a)$ represents the λ-term $\lambda a'.\,a$; so, assuming that a and a' are distinct atomic names, $sub\ a\,(\mathsf{V}\,a')\,(\mathsf{L}(\alpha a'.\,\mathsf{V}\,a))$ should represent the λ-term $(\lambda a'.\,a)[a'/a]$, that is $\lambda a''.\,a'$, where $a'' \neq a'$. Indeed, one can use the conversion equations in Figures 5 and 10 to calculate that

$$\begin{aligned}
& sub\ a\,(\mathsf{V}\,a')\,(\mathsf{L}(\alpha a'.\,\mathsf{V}\,a)) \\
= \quad & \{\text{by } \alpha\text{-equivalence for } \lambda\alpha\nu\delta \text{ expressions, where } a_1 \neq a, a'\} \\
& sub\ a\,(\mathsf{V}\,a')\,(\mathsf{L}(\alpha a_1.\,\mathsf{V}\,a)) \\
=_\beta \quad & \{\text{by (49)}\} \\
& \nu a_1.\,\mathsf{L}(\langle a_1\rangle(sub\ a\,(\mathsf{V}\,a')\,(\mathsf{V}\,a))) \\
=_\beta \quad & \{\text{by (47), Conv-Eqn and Conv-if-True}\} \\
& \nu a_1.\,\mathsf{L}(\langle a_1\rangle(\mathsf{V}\,a')) \\
\triangleq \quad & \{\text{where } a'' \neq a'\} \\
& \nu a_1.\,\mathsf{L}(\alpha a''.\,(a_1 \between a'')(\mathsf{V}\,a')) \\
=_\beta \quad & \{\text{by Conv-}\pi\text{-Data and Conv-}\pi\text{-Gnd}\} \\
& \nu a_1.\,\mathsf{L}(\alpha a''.\,\mathsf{V}\,a') \\
=_\beta \quad & \{\text{by Conv-}\nu\text{-Strengthening}\} \\
& \mathsf{L}(\alpha a''.\,\mathsf{V}\,a')
\end{aligned}$$

which does indeed represent the λ-term $\lambda a''.\,a'$. The claim that *sub* correctly represents capture-avoiding substitution for λ-terms, in general, is substantiated in Example 4.14 in Section 4.3.

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 in the $\lambda\alpha\nu\delta$-calculus by the open expression $\mathsf{L}(\langle a\rangle x) = \mathsf{L}(\alpha a'.\,(a \between a')x)$. Filling the hole in $\lambda a.[-]$ with a gives $\lambda a.\,a$; and correspondingly, the substituted expression $\mathsf{L}(\langle a\rangle x)[\mathsf{V}\,a/x]$ is indeed convertible to $\mathsf{L}(\alpha a.\,\mathsf{V}\,a)$.

Example 4.4 (*Counting occurrences of λ-abstractions.*)

Consider the function $|-|$ from λ-terms to numbers satisfying $|a| = 0$, $|t\,t'| = |t| + |t'|$ and $|\lambda a.\,t| = |t| + 1$. What could be simpler? And yet formal recursion schemes for λ-terms have found it tricky: see Section 3.3 in Gordin & Melham (1996) and Section 3 in Norrish (2004). We can represent this function in the $\lambda\alpha\nu\delta$-calculus

more or less directly by

$$cntlam \triangleq \mathsf{rec}\left(\begin{array}{l} c : \mathsf{Lam} \to \mathsf{Nat} \\ c(\mathsf{V}\,x) = zero \\ c(\mathsf{A}(y_1, y_2)) = plus\,(c\,y_1)\,(c\,y_2) \\ c(\mathsf{L}(\langle a \rangle y)) = \mathsf{S}\,(c\,y) \end{array}\right)$$

where $zero \triangleq \mathsf{Z}\,()$ and $plus \triangleq \lambda z_1 \to \lambda z_2 \to \mathsf{nrec}\,(\lambda() \to z_1)\,(\lambda z \to \mathsf{S}\,z)\,z_2$ is addition. From Equation (46), we have that *cntlam* satisfies

$$cntlam\,(\mathsf{V}\,e) =_\beta zero \tag{50}$$

$$cntlam\,(\mathsf{A}(e_1, e_2)) =_\beta plus\,(cntlam\,e_1)\,(cntlam\,e_2) \tag{51}$$

$$cntlam\,(\mathsf{L}(\alpha a.\, e)) =_\beta va.\,\mathsf{S}(cntlam\,e) \tag{52}$$

(Compare this with Example 4.4 in Schürmann *et al.*, 2001.) For example, $|\lambda a.\, a| = |a| + 1 = 0 + 1 = 1$; and correspondingly, $cntlam\,(\mathsf{L}(\alpha a.\, \mathsf{V}\,a)) =_\beta va.\,\mathsf{S}(cntlam(\mathsf{V}\,a)) =_\beta va.\,\mathsf{S}\,zero =_\beta \mathsf{S}\,zero$, by Equations (52) and (50) and Conv-v-Strengthening.

Example 4.5 (*Counting bound variable occurrences.*)
Define

$$cntvar \triangleq \lambda y \to \mathsf{rec}\left(\begin{array}{l} f : \mathsf{Lam} \to (\mathsf{Name} \to \mathsf{Nat}) \to \mathsf{Nat} \\ f(\mathsf{V}\,x) = \lambda b \to b\,x \\ f(\mathsf{A}(y_1, y_2)) = \lambda b \to plus\,(f\,y_1\,b)\,(f\,y_2\,b) \\ f(\mathsf{L}(\langle a \rangle y)) = \lambda b \to f\,y\,(\lambda x \to \\ \qquad\qquad \mathsf{if}\ x = a\ \mathsf{then}\ one\ \mathsf{else}\ b\,x) \end{array}\right) y\,(\lambda x \to zero)$$

$$\in Exp(\mathsf{Lam} \to \mathsf{Nat})$$

where $one \triangleq \mathsf{S}\,zero$. The idea is to count occurrences of bound variables in a λ-term. The iteration is parameterized by a function $b : \mathsf{Name} \to \mathsf{Nat}$ initially set to $\lambda x \to zero$ and updated by mapping a to *one* when passing under a λ-abstraction that binds the name a. This example should be compared with Example 4.3 in Schürmann *et al.* (2001).

Example 4.6 (*Listing binding variables.*)
The previous example shows that we can compute with object-level 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 binding variables in a λ-term

$$bv \triangleq \mathsf{rec}\left(\begin{array}{l} f : \mathsf{Lam} \to \mathsf{Lam} \\ f(\mathsf{V}\,x) = nil \\ f(\mathsf{A}(y_1, y_2)) = append\,(f\,y_1)\,(f\,y_2) \\ f(\mathsf{L}(\langle a \rangle y)) = cons(a, f\,y) \end{array}\right)$$

where we encode lists of atomic names as certain expressions of type Lam and $nil : \mathsf{Lam}$, $cons : (\mathsf{Name} \times \mathsf{Lam}) \to \mathsf{Lam}$ and $append : \mathsf{Lam} \to \mathsf{Lam} \to \mathsf{Lam}$ are suitable encodings of nil and cons constructors and an append operation for such lists. In fact, all that bv computes is a list of Anon's whose length is the number of occurrences of λ-binders in the λ-term. For example, $bv(\mathsf{L}(\alpha a.\, \mathsf{L}(\alpha a'.\, \mathsf{V}\,a))) =_\beta cons(\mathsf{Anon}, cons(\mathsf{Anon}, nil))$ because:

$$
\begin{aligned}
& bv(\mathsf{L}(\alpha a.\, \mathsf{L}(\alpha a'.\, \mathsf{V}\, a))) \\
=_\beta \quad & \{\text{by (46)}\} \\
& va.\, cons(a, va'.\, cons(a', nil)) \\
=_\beta \quad & \{\text{by Conv-}v\text{-Data and Conv-}v\text{-Prod}\} \\
& cons(va.\, a, cons(va.\, va'.\, a', va.\, va'.\, nil)) \\
=_\beta \quad & \{\text{by Conv-}v\text{-Strengthening}\} \\
& cons(va.\, a, cons(va'.\, a', nil)).
\end{aligned}
$$

Compare this with the Fresh Objective Caml function `listBvars` on p. 15 in Shinwell & Pitts (2005).

4.2 *Normalization for $\lambda\alpha v\delta$-calculus*

We extend the notion of normal and neutral forms for the $\lambda\alpha v$-calculus (Figure 6) to the $\lambda\alpha v\delta$-calculus as follows:

$$
\begin{array}{rcl@{\qquad}rcl}
n \in Nf & ::= & \cdots & u \in Neu & ::= & \cdots \\
& | & \mathsf{V}\, n & & | & \mathsf{A}\, u \\
& | & \mathsf{A}(n, n) & & | & \mathsf{L}\, u \\
& | & \mathsf{L}(\alpha a.\, n) & & | & \mathsf{Z}\, u \\
& | & \mathsf{Z}() & & | & \mathsf{Irec}\, n\, n\, n\, u \\
& | & \mathsf{S}\, n & & | & \mathsf{nrec}\, n\, n\, u
\end{array}
\tag{53}
$$

The neutral forms are slightly more complicated than might be expected, for example compared with Figure 4 in Pitts (2010), because we have chosen to make all constructors unary, via the use of product and name-abstraction types.

The notion of structural congruence between normal forms remains unchanged (Definition 3.15), as does the statement of the normalization theorem:

Theorem 4.7 ($\lambda\alpha v\delta$-*Calculus normalization.*)
Each typeable $\lambda\alpha v\delta$ expression is β-convertible to a normal form, which is unique up to structural congruence. (That is, properties (34) and (35) hold for the $\lambda\alpha v\delta$-calculus.)

Proof
The proof is an extension of that given in Section 3.4. The definition of weak head normal forms and weak neutral forms in Figure 7 is extended as follows:

$$
\begin{array}{rcl@{\qquad}rcl}
w \in Wnf & ::= & \cdots & v \in Wneu & ::= & \cdots \\
& | & \mathsf{V}\, e & & | & \mathsf{A}\, v \\
& | & \mathsf{A}(e, e) & & | & \mathsf{L}\, v \\
& | & \mathsf{L}(\alpha a.\, e) & & | & \mathsf{Z}\, v \\
& | & \mathsf{Z}() & & | & \mathsf{Irec}\, e\, e\, e\, v \\
& | & \mathsf{S}\, e & & | & \mathsf{nrec}\, e\, e\, v.
\end{array}
\tag{54}
$$

The relations $e \Downarrow n$, $e \Downarrow_{\mathrm{w}} w$ and $w \Downarrow_{\mathrm{n}} n$ defined in Figure 9 are extended by adding the rules in Figure 11. The auxiliary functions $w \mapsto a \setminus_{\mathrm{w}} w$ and $w \mapsto (a_1\ a_2)_{\mathrm{w}} w$ defined

$e \Downarrow n$, $e \Downarrow_{\mathrm{w}} w$ and $w \Downarrow_{\mathrm{n}} n$ are defined by the rules in Fig. 9 plus:

$$\frac{e \Downarrow_{\mathrm{w}} w \qquad K \in \{\mathsf{A},\mathsf{L},\mathsf{Z}\}}{K\,e \Downarrow_{\mathrm{w}} K\,w} \qquad \frac{e \Downarrow_{\mathrm{w}} \mathsf{V}\,e' \quad e_1\,e' \Downarrow_{\mathrm{w}} w}{\mathsf{Irec}\,e_1\,e_2\,e_3\,e \Downarrow_{\mathrm{w}} w} \qquad \frac{e \Downarrow_{\mathrm{w}} \mathsf{A}(e',e'') \quad e_2(\mathsf{Irec}\,e_1\,e_2\,e_3\,e', \mathsf{Irec}\,e_1\,e_2\,e_3\,e'') \Downarrow_{\mathrm{w}} w}{\mathsf{Irec}\,e_1\,e_2\,e_3\,e \Downarrow_{\mathrm{w}} w}$$

$$\frac{e \Downarrow_{\mathrm{w}} \mathsf{L}(\alpha a.\,e') \qquad a \notin \mathit{fn}(e_1,e_2,e_3) \quad e_3(\alpha a.\,\mathsf{Irec}\,e_1\,e_2\,e_3\,e') \Downarrow_{\mathrm{w}} w}{\mathsf{Irec}\,e_1\,e_2\,e_3\,e \Downarrow_{\mathrm{w}} w} \qquad \frac{e \Downarrow_{\mathrm{w}} v}{\mathsf{Irec}\,e_1\,e_2\,e_3\,e \Downarrow_{\mathrm{w}} \mathsf{Irec}\,e_1\,e_2\,e_3\,v} \qquad \frac{e \Downarrow_{\mathrm{w}} \mathsf{Z}() \quad e_1() \Downarrow_{\mathrm{w}} w}{\mathsf{nrec}\,e_1\,e_2\,e \Downarrow_{\mathrm{w}} w}$$

$$\frac{e \Downarrow_{\mathrm{w}} \mathsf{S}\,e' \qquad e_2(\mathsf{nrec}\,e_1\,e_2\,e') \Downarrow_{\mathrm{w}} w}{\mathsf{nrec}\,e_1\,e_2\,e \Downarrow_{\mathrm{w}} w} \qquad \frac{e \Downarrow_{\mathrm{w}} v}{\mathsf{nrec}\,e_1\,e_2\,e \Downarrow_{\mathrm{w}} \mathsf{nrec}\,e_1\,e_2\,v} \qquad \frac{e \Downarrow n}{\mathsf{V}\,e \Downarrow_{\mathrm{n}} \mathsf{V}\,n}$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow_{\mathrm{n}} n_2}{\mathsf{A}(e_1,e_2) \Downarrow_{\mathrm{n}} \mathsf{A}(n_1,n_2)} \qquad \frac{v \Downarrow_{\mathrm{n}} u}{\mathsf{A}\,v \Downarrow_{\mathrm{n}} \mathsf{A}\,u} \qquad \frac{e \Downarrow n}{\mathsf{L}(\alpha a.\,e) \Downarrow_{\mathrm{n}} \mathsf{L}(\alpha a.\,n)} \qquad \frac{v \Downarrow_{\mathrm{n}} u}{\mathsf{L}\,v \Downarrow_{\mathrm{n}} \mathsf{L}\,u} \qquad \frac{}{\mathsf{Z}() \Downarrow_{\mathrm{n}} \mathsf{Z}()}$$

$$\frac{v \Downarrow_{\mathrm{n}} u}{\mathsf{Z}\,v \Downarrow_{\mathrm{n}} \mathsf{Z}\,u} \qquad \frac{e \Downarrow n}{\mathsf{S}\,e \Downarrow_{\mathrm{n}} \mathsf{S}\,n} \qquad \frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad e_3 \Downarrow n_3 \quad v \Downarrow_{\mathrm{n}} u}{\mathsf{Irec}\,e_1\,e_2\,e_3\,v \Downarrow_{\mathrm{n}} \mathsf{Irec}\,n_1\,n_2\,n_3\,u} \qquad \frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad v \Downarrow_{\mathrm{n}} u}{\mathsf{nrec}\,e_1\,e_2\,v \Downarrow_{\mathrm{n}} \mathsf{nrec}\,n_1\,n_2\,u}$$

Fig. 11. Evaluation to normal form for $\lambda\alpha\nu\delta$-calculus.

in Figure 8 are extended as follows:

$$a \leftthreetimes_{\mathrm{w}} w \triangleq \begin{cases} \cdots & \\ \mathsf{V}(\nu a.\,e) & \text{if } w = \mathsf{V}\,e \\ \mathsf{A}(\nu a.\,e_1, \nu a.\,e_2) & \text{if } w = \mathsf{A}(e_1,e_2) \\ \mathsf{L}(\alpha a'.\,\nu a.\,e) & \text{if } w = \mathsf{L}(\alpha a'.\,e) \text{ where } a' \neq a \\ w & \text{if } w = \mathsf{Z}() \\ \mathsf{S}(\nu a.\,e) & \text{if } w = \mathsf{S}\,e \end{cases} \tag{55}$$

$$(a_1\ a_2) \cdot_{\mathrm{w}} w \triangleq \begin{cases} \cdots & \\ \mathsf{V}((a_1 \leftrightarrow a_2)e) & \text{if } w = \mathsf{V}\,e \\ \mathsf{A}((a_1 \leftrightarrow a_2)e_1, (a_1 \leftrightarrow a_2)e_2) & \text{if } w = \mathsf{A}(e_1,e_2) \\ \mathsf{L}(\alpha a'.\,(a_1 \leftrightarrow a_2)e) & \text{if } w = \mathsf{L}(\alpha a'.\,e) \text{ with } a' \neq a_1, a_2 \\ w & \text{if } w = \mathsf{Z}() \\ \mathsf{S}((a_1 \leftrightarrow a_2)e) & \text{if } w = \mathsf{S}\,e. \end{cases} \tag{56}$$

These extensions do not affect the proof of Lemma 3.18. So, one can prove the normalization theorem by proving the property in Proposition 3.19; this is done, as before, using a logical relation; the details are in Appendix C. □

4.3 Representational adequacy

In this section, we show that λ-terms and functions on them defined using the α-structural recursion principle (Theorem 1.1) can be faithfully represented in the $\lambda\alpha\nu\delta$-calculus.

Definition 4.8 (*Closed expressions and normal forms.*)
For each $T \in Typ$, let $Cexp(T)$ denote the subset of $Exp(T)$ consisting of $\lambda\alpha\nu\delta$ expressions of type T that are *closed*, that is which have no free variables; and let $Cnf(T)$ denote the subset of normal forms with no free variables. Note that if e is a closed expression, then so is $\pi \cdot e$ for any $\pi \in \mathrm{Perm}(\mathbb{A})$; so $Cexp(T)$ is a nominal subset of the nominal set $Exp(T)$. Similarly, $Cnf(T)$ is a nominal subset of $Nf(T)$.

Of course closed expressions may well involve free atomic names – we use the latter to represent free object-level variables, as we shall see for the example of the untyped λ-calculus.

Inspecting the definition of $\lambda\alpha\nu\delta$-calculus normal and neutral forms in Figure 6 and Equation (54), it is apparent that neutral forms always contain at least one free variable and hence that $Cnf(\mathsf{Lam})$ is in bijection with the nominal set

$$\Lambda[\mathsf{Anon}] \triangleq \{t ::= \mathsf{Anon} \mid a \mid t\,t \mid \lambda a.\,t\}/\equiv_\alpha \tag{57}$$

of possibly open untyped λ-terms (modulo α-equivalence, of course) involving a constant symbol Anon. Consequently, we get a simple form of 'representational adequacy' (Pfenning, 2001) result within $\lambda\alpha\nu\delta$-calculus for the object language consisting of the untyped λ-calculus:

Proposition 4.9
With Λ as in Equation (1), the function $\ulcorner - \urcorner : \Lambda \to Cnf(\mathsf{Lam})$ satisfying

$$\left.\begin{aligned} \ulcorner a \urcorner &= \mathsf{V}\,a \\ \ulcorner t\,t' \urcorner &= \mathsf{A}(\ulcorner t \urcorner, \ulcorner t' \urcorner) \\ \ulcorner \lambda a.\,t \urcorner &= \mathsf{L}(\alpha a.\ulcorner t \urcorner) \end{aligned}\right\} \tag{58}$$

(well defined by Theorem 1.1) gives a bijection between Λ and the subset of $Cnf(\mathsf{Lam})$ of closed normal forms not involving Anon (and hence, not involving any use of name restriction, $\nu a.(-)$). □

Remark 4.10 (*The 'anonymous name'.*)
We have seen that admitting locally scoped names at all types leads to a simple equational calculus for name-abstraction and its unbinding destructor. The price we pay for this is the existence of the canonical form Anon (Definition 3.3), and hence, the fact that object-level syntax (such as λ-terms) is injectively, but not bijectively represented by $\lambda\alpha\nu\delta$-calculus normal forms. Nevertheless, the representation (58) is very simple. This is good; the 'coding gap' between object- and metalanguage is very small – we just have to take care with the extra constant Anon when manipulating the object language from within the $\lambda\alpha\nu\delta$-calculus.

Below we will need the following result showing that for closed expressions of ground type the 'non-binding' form of name-abstraction $\langle a \rangle e$ (Definition 3.5) is β-convertible to the binding form $\alpha a.\,e$.

Lemma 4.11
If $G \in \{\mathsf{Name}, \mathsf{Bool}, \mathsf{Lam}, \mathsf{Nat}\}$, $a \in \mathbb{A}$ and $e \in Cexp(G)$, then $\langle a \rangle e =_\beta \alpha a.\,e$.

Proof
If suffices to prove

$$(\forall a_1, a_2 \in \mathbb{A}, e \in Cexp(G))\ (a_1 \wr a_2)e\ =_\beta\ (a_1\ a_2) \cdot e \tag{59}$$

since then, by definition of α-conversion, we have

$$\alpha a.\, e = \alpha a'.\, (a\ a') \cdot e =_\beta \alpha a'.\, (a \wr a')e \triangleq \langle a \rangle e$$

where a' is any atomic name not free in e. By the normalization Theorem 4.7, for Equation (59), it suffices to prove $(a_1 \wr a_2)n =_\beta (a_1\ a_2) \cdot n$ for all $n \in Cnf(G)$. This follows by α-structural induction for n, relying on the fact that closed normal forms of ground type do not involve neutral subexpressions. □

We turn next to the representation of functions on Λ by computing normal forms in $\lambda\alpha\nu\delta$-calculus. First note that our proof of the normalization Theorem 4.7 gives more than just the existence and uniqueness (modulo $\equiv$) of normal forms: we proved that the recursively enumerable evaluation relations $\{(e, n) \in Exp(T) \times Nf(T) \mid e \Downarrow n\}$ are in fact the graphs of total functions that pick out representatives of normal forms of expressions within $\equiv$-equivalence classes.

Definition 4.12 (*Normalization function.*)
For each $T \in Typ$ and $e \in Exp$, write $nf_T(e)$ for the unique $n \in Nf(T)$ satisfying $e \Downarrow n$. Thus, $nf_T(-)$ satisfies:

$$e \in Exp(T)\ \Rightarrow\ e =_\beta nf_T(e) \tag{60}$$

$$n \in Nf(T)\ \Rightarrow\ nf_T(n) = n \tag{61}$$

$$e, e' \in Exp(T)\ \wedge\ e =_\beta e'\ \Rightarrow\ nf_T(e) \equiv nf_T(e'). \tag{62}$$

Note that since $\Downarrow$ is equivariant so is nf_T so that $nf_T \in Nom(Exp(T), Nf(T))$. In view of Equation (42), we also have that nf_T restricts to a function from $Cexp(T)$ to $Cnf(T)$.

Given any type $T \in Typ$, let X denote the quotient nominal set $Cexp(T)/\equiv$ of closed normal forms of type T modulo structural congruence. We write $[n]$ for the equivalence class of $n \in Cnf(T)$. Suppose that the functions

$$f_1 \in \mathbb{A} \to_{\mathrm{fs}} X$$
$$f_2 \in (X \times X) \to_{\mathrm{fs}} X$$
$$f_3 \in (\mathbb{A} \times X) \to_{\mathrm{fs}} X$$

are all supported by the finite subset $\overline{a} \subseteq \mathbb{A}$ and that f_3 satisfies the 'freshness condition for binders'

$$a \notin \overline{a}\ \Rightarrow\ (\forall x \in X)\ a \# f_3(a, x). \tag{FCB}$$

Let $f \in \Lambda \to_{\mathrm{fs}} X$ be the function defined from f_1, f_2 and f_3 by α-structural recursion as in Theorem 1.1. We will show that if f_1, f_2 and f_3 are representable in $\lambda\alpha\nu\delta$-calculus in a suitable sense, then so is f.

Theorem 4.13 (*Representation of* α*-structural recursion.*)
With f_1, f_2, f_3 and $\overline{a}$ as above, suppose the closed expressions $e_1 \in Cexp(\mathsf{Name} \to T)$, $e_2 \in Cexp((T \times T) \to T)$ and $e_3 \in Cexp((\mathsf{Name}\,.\,T) \to T)$ satisfy

$$fn(e_1, e_2, e_3) \subseteq \overline{a} \tag{63}$$

$$f_1\, a \;=\; [nf_T(e_1\, a)] \tag{64}$$

$$f_2([n], [n']) \;=\; [nf_T(e_2\,(n, n'))] \tag{65}$$

$$a \notin \overline{a} \;\Rightarrow\; f_3(a, [n]) \;=\; [nf_T(e_3\, \alpha a.\, n)] \tag{66}$$

for all $a \in \mathbb{A}$ and $n, n' \in Cnf(T)$. Then, the function $f \in \Lambda \to_{\mathrm{fs}} X$ defined by α-structural recursion from (f_1, f_2, f_3) is represented by $\lambda x \to \mathsf{lrec}\, e_1\, e_2\, e_3\, x \in Cexp(\mathsf{Lam} \to T)$ in the sense that for all $t \in \Lambda$

$$f\, t \;=\; [nf_T(\mathsf{lrec}\, e_1\, e_2\, e_3\, \ulcorner t \urcorner)]. \tag{67}$$

Proof

By the uniqueness part of α-structural recursion, to prove Equation (67), it suffices to show that $t \mapsto [nf_T(\mathsf{lrec}\, e_1\, e_2\, e_3\, \ulcorner t \urcorner)]$ satisfies the recursion scheme (2) that defines f; in other words, writing $r\, \ulcorner t \urcorner$ for $\mathsf{lrec}\, e_1\, e_2\, e_3\, \ulcorner t \urcorner$, it suffices to prove:

$$[nf_T(r\, \ulcorner a \urcorner)] = f_1\, a \tag{68}$$

$$[nf_T(r\, \ulcorner t_1\, t_2 \urcorner)] = f_2([nf_T(r\, \ulcorner t_1 \urcorner)], [nf_T(r\, \ulcorner t_2 \urcorner)]) \tag{69}$$

$$a \notin \overline{a} \;\Rightarrow\; [nf_T(r\, \ulcorner \lambda a.\, t \urcorner)] = f_3(a, [nf_T(r\, \ulcorner t \urcorner)]). \tag{70}$$

We give the proof of Equation (70); the proofs of Equations (68) and (69) are similar. Suppose $a \notin \overline{a}$. Then, for any $t \in \Lambda$

$$\begin{aligned}
& r\, \ulcorner \Lambda a.\, t \urcorner \\
= \quad & \{\text{by (58)}\} \\
& r\, (\mathsf{L}(\alpha a.\, \ulcorner t \urcorner)) \\
=_\beta \quad & \{\text{by Conv-}\delta\text{-L in Figure 10, since } a \notin \overline{a} \supseteq fn(e_1, e_2, e_3) \text{ by (63)}\} \\
& e_3\, \alpha a.\, r\, \ulcorner t \urcorner \\
=_\beta \quad & \{\text{by (60)}\} \\
& e_3\, \alpha a.\, nf_T(r\, \ulcorner t \urcorner).
\end{aligned}$$

Therefore, $[nf_T(r\, \ulcorner \Lambda a.\, t \urcorner)] = [nf_T(e_3\, \alpha a.\, nf_T(r\, \ulcorner t \urcorner))]$ and combining this with Equation (66) gives Equation (70). □

Example 4.14

Theorem 4.13 can be used to prove that the expression *sub* defined in Example 4.3 does indeed represent capture-avoiding substitution on λ-terms, in the sense that for all $a \in \mathbb{A}$ and $t, t' \in \Lambda$

$$sub\; a\, \ulcorner t \urcorner\, \ulcorner t' \urcorner \;=_\beta\; \ulcorner t'[t/a] \urcorner. \tag{71}$$

Proof of (71).
Fixing a and t, in the theorem take $T = \mathsf{Lam}$ and (f_1, f_2, f_3) to be the functions well defined by

$$\begin{aligned} f_1\, a' &\triangleq \begin{cases} [\ulcorner t \urcorner] & \text{if } a' = a \\ [\mathsf{V}\, a'] & \text{if } a' \neq a \end{cases} \\ f_2([n_1], [n_2]) &\triangleq [\mathsf{A}(n_1, n_2)] \\ f_3(a', [n]) &\triangleq [\mathsf{L}(\alpha a'.\, n)]. \end{aligned}$$

for all $a' \in \mathbb{A}$ and $n, n' \in Nf(\mathsf{Lam})$. They are supported by the finite set $\overline{a}$ consisting of a and the free atomic names of t; and f_3 satisfies FCB since $a' \notin fn(\mathsf{L}(\alpha a'.\, n))$. Let f be the function defined from them by α-structural recursion as in Theorem 1.1. An easy proof by α-structural induction (Pitts, 2006) shows that $(\forall t' \in \Lambda)\ f\, t' = [\ulcorner t'[t/a] \urcorner]$. So to prove Equation (71), it suffices to show that for all $t' \in \Lambda$

$$f\, t' \;=\; [nf_{\mathsf{Lam}}(sub\; a\, \ulcorner t \urcorner\, \ulcorner t' \urcorner)]. \tag{72}$$

Note that by the definition of *sub* in Example 4.3, we have $sub\; a\, \ulcorner t \urcorner =_\beta \lambda y \to \mathsf{Irec}\, e_1\, e_2\, e_3\, y$ where

$$\begin{aligned} e_1 &\triangleq \lambda x \to \mathsf{if}\ x = a\ \mathsf{then}\ \ulcorner t \urcorner\ \mathsf{else}\ \mathsf{V}\, x \\ e_2 &\triangleq \lambda(y_1, y_2) \to \mathsf{A}(y_1, y_2) \\ e_3 &\triangleq \lambda\langle a' \rangle y \to \mathsf{L}(\langle a' \rangle y). \end{aligned}$$

So, Equation (72) follows from Theorem 4.13 once we verify properties (63)–(66) for e_1, e_2 and e_3. Note that by definition of $\ulcorner t \urcorner$, it has the same free atomic names as t; therefore, we certainly have Equation (63). Property (64) is an immediate consequence of the definitions of f_1 and e_1, together with the definition of β-conversion; similarly, for Equation (65). Finally, for Equation (66), we use Lemma 4.11; if $a' \notin \overline{a} = \{a\} \cup fn(t)$ then

$$\begin{aligned} & e_3\, \alpha a'.\, n \\ =_\beta\ & \{\text{by definition of } e_3\} \\ & \mathsf{let}\ \langle a' \rangle y = \alpha a'.\, n\ \mathsf{in}\ \mathsf{L}(\langle a' \rangle y) \\ =_\beta\ & \{\text{by CONV-}\beta\text{-ABS in Figure 5}\} \\ & \nu a'.\, \mathsf{L}(\langle a' \rangle n) \\ =_\beta\ & \{\text{by Lemma 4.11}\} \\ & \nu a'.\, \mathsf{L}(\alpha a'.\, n) \\ =_\beta\ & \{\text{by CONV-}\nu\text{-STRENGTHENING in Figure 5}\} \\ & \mathsf{L}(\alpha a'.\, n) \end{aligned}$$

and hence by Equations (61) and (62), $f_3(a', [n]) \triangleq [\mathsf{L}(\alpha a'.\, n)] = [nf_{\mathsf{Lam}}(e_3\, \alpha a'.\, n)]$. □

5 Related work

Notions of name-restriction in nominal sets. The notion of nominal sets equipped with a name-restriction operation given in Section 2.3 was discussed by the author in a talk on 'Nominal Semantics of Abstraction and Restriction' at the conference

on Category Theory and Computer Science that took place in Copenhagen in 2004 and in a manuscript privately circulated around that time to do with free nominal restriction sets. However, several other people researching nominal techniques have considered operations, mainly syntactical ones, that have the characteristic properties (\-Alpha), (\-Strengthening) and (\-Exchange). See, for example Cardelli & Gordon (2001), Fernández & Gabbay (2009) and Gabbay & Lengrand (2009). What is new here is the observation that nominal sets equipped with a name-restriction operation are closed under exponentiation by nominal sets (Theorems 2.10) and under taking nominal sets of name-abstractions (Theorem 2.13). This immediately suggests a simple calculus of higher order functions with name-restriction and name-abstraction that turns out to resurrect Odersky's conception of locally scoped name.

Computing with higher order abstract syntax. It has become very common to use typed λ-calculus as a uniform method of representing syntax involving binding (Pfenning & Elliott, 1988). The pros and cons of this *higher order abstract syntax* compared with 'nominal' techniques have been vigorously debated (Cheney, 2005; Crary & Harper, 2006). 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? The primary focus of this paper is on functional computation on representations. So, the analyses of Poswolsky & Schürmann (2008) and Licata *et al.* (2008)(Licata *et al.*, 2008) are pertinent: early uses of typed λ-calculus representations identify 'functions-as-data' with 'functions-as-computation' (Miller, 1990 provides an early exception) and this leads to complications, such as modalities (Schürmann *et al.*, 2001), when trying to develop recursion and induction for higher order abstract syntax. Various authors have advocated separating the two notions of function, leading to forms of locally scoped symbols in Poswolsky & Schürmann (2008), Pientka (2008), Licata *et al.* (2008), Licata & Harper (2009) and Westbrook *et al.* (2009) that are comparable to the notion of name-abstraction $\alpha a.e$ in the $\lambda\alpha v$-calculus used in this paper. This should not be confused with the notion of name-restriction $va.e$ that we are using here.[4] For one thing the latter does not change the type of expressions, whereas the former does. The nominal sets model in Section 2 clarifies the difference between the two notions. Furthermore, Theorem 2.13 provides an interesting semantical insight: in the presence of name-restriction, it is consistent to regard types $\mathsf{Name} \,.\, T$ of 'functions-as-data' as subtypes of types $\mathsf{Name} \to T$ of 'functions-as-computation'.

A characteristic feature of using typed λ-calculus to represent binding is that one gets substitution and β-equivalence 'for free' in addition to renaming and α-equivalence. This is often seen as a strength of the approach (Pientka, 2008), but the author is 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

[4] Unfortunately 'v' is used to indicate abstraction rather than restriction by Poswolsky & Schürmann (2008) and by Westbrook *et al.* (2009).

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 Poswolsky & Schürmann (2008) and Licata & Harper (2009), for example here some things are not automatic. Similarly, Licata & Harper (2009) incorporate types classifying closed object-level expressions, whereas the author would prefer to let the user make inductive definitions of such types. On the other hand, the use of locally scoped symbols of any data type, rather than just at name types like Name as here, seems an interesting feature of Miller (1990), Miller & Tiu (2005), Poswolsky & Schürmann (2008) and Licata *et al.* (2008).

Nominal calculi. Harper has coined the term 'pronominal' for the use of locally scoped symbols as 'pronouns referring to a designated binding site' (Licata & Harper, 2009), contrasting it with the nominal approach to symbols as nouns with independent existence. Is the $\lambda\alpha\nu$-calculus 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 $\lambda\alpha\nu$-calculus, 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 Conv-ν-Fun and Conv-ν-Prod conversions in Figure 5).

However, $\lambda\alpha\nu$-calculus is 'nominal' in the sense of being a language for describing some aspects of the theory of nominal sets. Among previous such languages perhaps the most closely related to $\lambda\alpha\nu$-calculus is the 'simple nominal type theory' (SNTT) of Cheney (2009). The motivation behind both systems is to produce a calculus combining simple type theory (initially, and dependent type theory in the long run) with some of the distinctive features of the nominal sets model of names and binding, particularly inductively defined nominal sets involving the name-abstraction construct. Moreover, both $\lambda\alpha\nu$-calculus and SNTT aim to avoid the need for freshness side conditions while defining and computing in the calculus. However, SNTT's use of bunched contexts containing information about object-level freshness means that the aim of avoiding freshness conditions is only partially met: well typedness of SNTT terms is mediated by freshness conditions in the context. By contrast, $\lambda\alpha\nu$-calculus 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 Pfenning & Elliott, 1988). This is at the expense of introducing terms like $\nu a.\, a$ (*cf.* Remark 4.10), but the fact that SNTT lacks name-restriction $\nu a.\,(-)$ and name-swapping $(a_1 \wr a_2)(-)$ limits its expressivity. Cheney (2009) discusses the limitations caused by lack of a name-restriction construct in Section 4 of that paper. In particular, some of the 'shocking' isomorphisms of Example 3.7 are not expressible in SNTT; see Figure 5 in Cheney (2009).[5] Nevertheless, SNTT is a very interesting system whose metatheory is simpler

[5] Be warned, Cheney (2009) uses the notation $\langle a \rangle(-)$ for the *binding* operation that is denoted here by $\alpha a.\,(-)$.

than the one presented here. It would be interesting to investigate translating it into $\lambda\alpha\nu$-calculus; 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.

Nominal System T. This paper is a revised and expanded version of Pitts (2010). The calculus presented in that paper is called *Nominal System T* because it takes Gödel's System T for primitive recursive functions of higher type (Gödel, 1958), formulated as a typed λ-calculus following Tait (1967) and generalizes from numbers to inductive data modulo α-equivalence of bound names. The $\lambda\alpha\nu$-calculus presented here follows the suggestion in Section 7 of that paper and provides name-abstraction types. As we noted at the beginning of Section 4, this enables any nominal signature of constructors to be added easily to the calculus. In formulating the $\lambda\alpha\nu\delta$-calculus, we added the signatures for λ-terms and numbers and developed iteration combinators for them that correspond to their initial algebra semantics in nominal restriction sets; doubtless more general forms of primitive recursion rather than iteration could be given, along the lines of those in Pitts (2010). That paper develops η-long β-normal forms for Nominal System T (via a normalization-by-evaluation argument), whereas here we have just used β-normal forms – they are sufficient for the results in Section 4.3.

This paper gives (in Section 2) the details of the semantics of locally scoped names in nominal sets alluded to in Pitts (2010). In doing so, a flaw in that paper emerges. Nominal System T admits name-swapping expressions with arbitrary expressions of type Name to be swapped, rather than just atomic names as here. For example, $(\mathsf{Anon} \wr a)e$ is well formed in Nominal System T (with slightly different concrete syntax). We can extend the denotational semantics of Section 3.2 to cope with such expressions. In particular, the denotation of $(\mathsf{Anon} \wr a)e$ is given by restriction, $\lambda\rho \in \mathit{Val}.\, a\backslash([\![e]\!]\rho)$. However, the conversion $(\pi\lambda)$ in Figure 3 of Pitts (2010) is not sound for this semantics. A correct, but more complicated definition of conversion could be given; however, here we have taken the simple way out and made the syntactic restriction that only atomic names can appear in explicit swapping expressions. This has the knock-on effect, via Definition 3.4, of restricting the operation of concreting a name-abstraction to apply only to atomic names. This does restrict the expressivity of $\lambda\alpha\nu$-calculus; for example, one cannot λ-abstract a in $\lambda x \to x \mathbin{@} a \in \mathit{Exp}((\mathsf{Name}\ .\ T) \to T)$ to get a concretion function of type $\mathsf{Name} \to (\mathsf{Name}\,.\,T) \to T$. However, the restriction does not prevent the formulation of structural recursion with locally scoped names in Section 4.

6 Conclusion

Locally scoped names are an important feature of the informal metatheory of binding operations in programming languages and logics. Nominal sets provide a fruitful mathematical model of name scoping in the presence of recursive definitions. Previous work on formal languages based on this model (Pitts & Gabbay, 2000; Schöpp & Stark, 2004; Cheney, 2009) uses some form of freshness information built

into a type system in order to ensure that an expression using a locally scoped name, when interpreted in nominal sets, does not have the name in its support. In this paper, such 'static freshness inference' is avoided by using an Odersky-style local scoping construct, which is given a new interpretation using nominal-sets-with-restriction, rather than just nominal sets. This results in a more standard form of typed λ-calculus with fewer side conditions than Schöpp and Stark's calculus and greater expressiveness than Cheney's calculus. However, as pointed out in Remark 4.10, our approach does entail dealing with the extra canonical form $\mathsf{Anon} = \nu a.\, a$. The presence of this single nonstandard constant of type Name does not seem quite as bad as the 'exotic' terms that can appear in systems based on weak high-order abstract syntax, but nevertheless, it will complicate any logic based upon the $\lambda\alpha\nu\delta$-calculus. In managing to avoid static freshness inference, perhaps we have traded one form of complexity in reasoning for another?

To investigate this question further, an obvious next step is to try to generalize the use we have made of local names and name-swapping from simple to 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 (Nordström *et al.*, 1990). 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 (Pfenning, 2001) that admits familiar forms of specification using bound names and formalizes the informal uses of 'modulo α' recursion and induction that are common in the practice of programming language semantics. This is certainly the motivation of previous work in this line (Schöpp & Stark, 2004; Westbrook *et al.*, 2009), but the hope is that the use of the form of locally scoped names considered here can lead to a simpler system that is more expressive, or at least which is closer to informal practice in its forms of expression.

A Proof of Theorem 2.10

The proof of Theorem 2.10 depends upon the following result.

Lemma A1
Suppose $X \in \mathit{Nom}$ and $Y \in \mathit{Res}$. For each $f \in X \to_{\mathrm{fs}} Y$ and $a \in \mathbb{A}$, there is a unique $g \in X \to_{\mathrm{fs}} Y$ satisfying

$$a \# g \tag{A1}$$

$$(\forall x \in X)\; a \# x \;\Rightarrow\; g\,x = a\backslash f\,x \tag{A2}$$

Proof
First note that we have

$$(\forall a', a'' \in \mathbb{A})\; a', a'' \# (f, x) \;\Rightarrow\; a'\backslash((a\;a')\cdot f)x = a''\backslash((a\;a'')\cdot f)x. \tag{A3}$$

To see this, suppose $a', a'' \# (f, x)$. If $a = a'$ or $a = a''$, then $a, a', a'' \# (f, f\,x)$ and so both the right- and left-hand sides of the Equation (A3) are equal to $f\,x$ by property $\backslash$-Strengthening. So we may suppose $a' \neq a \neq a''$; but then by property $\backslash$-Alpha,

we have $a'\backslash((a\ a')\cdot f)x = (a'\ a'')\cdot(a'\backslash((a\ a')\cdot f)x)$, from which the desired equality follows by equivariance of $(-)\backslash(-)$.

In view of Equation (A3), we get a well-defined function by mapping each $x \in X$ to

$$g\,x = a'\backslash((a\ a')\cdot f)\,x \quad \text{for } \textit{any } a' \# (f,x). \tag{A4}$$

Note that by equivariance of $\backslash$, $g\,x = (a\ a')\cdot(a\backslash f((a\ a')\cdot x))$. So when $a \# x$ and hence $(a\ a')\cdot x = x$, we get $g\,x = (a\ a')\cdot(a\backslash f\,x) = a\backslash f\,x$ since $a, a' \# a\backslash f\,x$. Thus, Equation (A2) holds for this definition of g.

Since $\backslash$ is equivariant, it follows from its definition that g is supported by the finite set $\mathit{supp}(f)\cup\{a\}$ and hence is in $X \to_{\mathrm{fs}} Y$. In fact, g is supported by $\mathit{supp}(f)-\{a\}$, since choosing any $a' \# (a,f)$, we have for any $x \in X$:

$$\begin{aligned}
&((a\ a')\cdot g)\,x\\
=\ &\{\text{permutation action on functions}\}\\
&(a\ a')\cdot(g((a\ a')\cdot x))\\
=\ &\{\text{by (A4), choosing } a'' \# (f,a,a',x) \text{ and hence with } a'' \# (f,(a\ a')\cdot x)\}\\
&(a\ a')\cdot(a''\backslash((a\ a'')\cdot f)((a\ a')\cdot x))\\
=\ &\{\text{equivariance and } a,a' \notin \mathit{supp}((a\ a'')\cdot f)\}\\
&a''\backslash((a\ a'')\cdot f)\,x\\
=\ &\{\text{by (A4), since } a'' \# (f,x)\}\\
&g\,x.
\end{aligned}$$

Hence, $(a\ a')\cdot g = g$; and since $a' \# (a,f)$, we have $a' \# g$ and thus $a = (a\ a')\cdot a' \# (a\ a')\cdot g = g$. So, g satisfies Equation (A1).

For uniqueness, if $g' \in X \to_{\mathrm{fs}} Y$ also satisfies Equations (A1) and (A2), then for any x, picking $a' \# (g,g',x)$ we have

$$\begin{aligned}
&g'\,x\\
=\ &\{a, a' \# g' \text{ by (A1) and assumption on } a'\}\\
&(a\ a')\cdot g'((a\ a')\cdot x)\\
=\ &\{\text{by (A2) for } g', \text{ since } a \# (a\ a')\cdot x\}\\
&(a\ a')\cdot(a\backslash f((a\ a')\cdot x))\\
=\ &\{\text{by (A2) for } g, \text{ since } a \# (a\ a')\cdot x\}\\
&(a\ a')\cdot g((a\ a')\cdot x)\\
=\ &\{a, a' \# g \text{ by (A1) and assumption on } a'\}\\
&g\,x.
\end{aligned}$$

Thus, $g' = g$. □

Proof of Theorem 2.10.
For each $a \in \mathbb{A}$ and $f \in X \to_{\mathrm{fs}} Y$, let $a\backslash f$ be the unique function $g \in X \to_{\mathrm{fs}} Y$ as in Lemma A1, which indeed has property (6). The fact that $\backslash \in \mathit{Nom}(\mathbb{A}\times(X\to_{\mathrm{fs}} Y), X\to_{\mathrm{fs}} Y)$ follows easily from its definition; so it just remains to check that it has properties $\backslash$-ALPHA, $\backslash$-STRENGTHENING and $\backslash$-EXCHANGE. The first is Equation (A1). For the second, if $a \# f$, then $a\backslash f$ and f are both finitely supported functions satisfying Equatons (A1) and (A2); hence, by the uniqueness part of Lemma A1, $a\backslash f = f$. Finally, for property $\backslash$-EXCHANGE, without loss of generality, we may assume $a \neq a'$ and then for any $x \in X$ we have:

$$
\begin{aligned}
& (a\backslash a'\backslash f)\,x \\
= \quad & \{\text{by (A4), choosing } a'' \# (a, a', f, x) \text{ and hence with } a'' \# (a'\backslash f, x)\} \\
& a''\backslash((a\ a'')\cdot(a'\backslash f))\,x \\
= \quad & \{\text{equivariance of } \backslash \text{ and } a' \# (a, a'')\} \\
& a''\backslash(a'\backslash((a\ a'')\cdot f))\,x \\
= \quad & \{\text{by (A4), choosing } a''' \# (a, a', a'', f, x) \text{ and hence with } a''' \# ((a\ a'')\cdot f, x)\} \\
& a''\backslash a'''\backslash((a'\ a''')(a\ a'')\cdot f)\,x \\
= \quad & \{\text{by } (\backslash\text{-Exchange}) \text{ for } Y \in \mathit{Res} \text{ and } a', a''' \# (a, a'')\} \\
& a'''\backslash a''\backslash((a\ a'')(a'\ a''')\cdot f)\,x \\
= \quad & \{\text{reversing the above steps}\} \\
& (a'\backslash a\backslash f)\,x. \quad \square
\end{aligned}
$$

B Proof of Theorem 2.13

For each $a \in \mathbb{A}$, consider $r_a \in (\mathbb{A} \times X) \to_{\mathrm{fs}} [\mathbb{A}]X$ given by

$$r_a(a', x) \triangleq \begin{cases} \langle a'\rangle x & \text{if } a = a' \\ \langle a'\rangle(a\backslash x) & \text{if } a \neq a'. \end{cases}$$

Note that $a' \# r_a(a', x)$ for all a', x; hence, r_a induces $\hat{r}_a \in [\mathbb{A}]X \to_{\mathrm{fs}} [\mathbb{A}]X$ as in Proposition 2.5. Writing $a\backslash p$ for $\hat{r}_a\, p$, we have

$$a\backslash(\langle a'\rangle x) = \begin{cases} \langle a'\rangle x & \text{if } a = a' \\ \langle a'\rangle(a\backslash x) & \text{if } a \neq a'. \end{cases} \tag{B1}$$

So property (8) holds; and it is easy to see that $\backslash$ is equivariant and satisfies $\backslash$-Alpha, $\backslash$-Strengthening and $\backslash$-Exchange.

Using the name-restriction operation on $\mathbb{A} \to_{\mathrm{fs}} X$ from Theorem 2.10, we get an equivariant function in $(\mathbb{A} \times X) \to_{\mathrm{fs}} (\mathbb{A} \to_{\mathrm{fs}} X)$ by mapping each (a, x) to the finitely supported function $a\backslash(\lambda a' \in \mathbb{A}.\,(a\ a')\cdot x) \in \mathbb{A} \to_{\mathrm{fs}} X$. By Proposition 2.5, this induces a well-defined (and equivariant) function m satisfying

$$m(\langle a\rangle x) = a\backslash(\lambda a' \in \mathbb{A}.\,(a\ a')\cdot x) \tag{B2}$$

for all $a \in \mathbb{A}$ and $x \in X$. It is not hard to see that m preserves name-restriction and hence $m \in \mathit{Res}([\mathbb{A}]X, \mathbb{A} \to_{\mathrm{fs}} X)$. For if $a, a', a'' \in \mathbb{A}$ are mutually distinct, then for any $x \in X$, we have

$$
\begin{aligned}
& (a\backslash m(\langle a'\rangle x))a'' \\
= \quad & \{\text{by (A2), since } a \neq a''\} \\
& a\backslash m(\langle a'\rangle x)a'' \\
= \quad & \{\text{by (B2)}\} \\
& a\backslash(a'\backslash(\lambda b \in \mathbb{A}.\,(a'\ b)\cdot x)a'' \\
= \quad & \{\text{by (A2), since } a' \neq a''\} \\
& a\backslash a'\backslash(a'\ a'')\cdot x \\
= \quad & \{\text{by } \backslash\text{-Exchange, equivariance of } \backslash \text{ and since } a \neq a', a''\} \\
& a'\backslash(a'\ a'')\cdot(a\backslash x) \\
= \quad & \{\text{by (A2), since } a' \neq a''\}
\end{aligned}
$$

$$
\begin{aligned}
& (a'\backslash(\lambda b \in \mathbb{A}.\,(a'\ b)\cdot(a\backslash x))a'' \\
= \quad & \{\text{by (B2)}\} \\
& m(\langle a'\rangle(a\backslash x))a'' \\
= \quad & \{\text{by (B1), since } a \neq a'\} \\
& m(a\backslash\langle a'\rangle x)a''
\end{aligned}
$$

and this suffices to show that $a\backslash m\,y = m(a\backslash y)$ holds for all $a \in \mathbb{A}$ and $y \in [\mathbb{A}]X$.

In the other direction, given $f \in \mathbb{A} \to_{\mathrm{fs}} X$, note that if $a, a' \# f$, then $\langle a\rangle(f\ a) = \langle a'\rangle(f\ a')$; so we get a function e from $\mathbb{A} \to_{\mathrm{fs}} X$ to $[\mathbb{A}]X$ by defining

$$e(f) = \langle a\rangle(f\ a) \quad \text{where } a \# f. \tag{B3}$$

and this gives a morphism in $Res(\mathbb{A} \to_{\mathrm{fs}} X, [\mathbb{A}]X)$. Finally, to see that $e \circ m = id_{[\mathbb{A}]X}$, note that

$$
\begin{aligned}
& e(m(\langle a\rangle x)) \\
= \quad & \{\text{by (B3), where } a' \# (a, x)\} \\
& \langle a'\rangle(m\,(\langle a\rangle x)\,a') \\
= \quad & \{\text{by (B2) and (6), since } a \neq a'\} \\
& \langle a'\rangle(a\backslash(a\ a')\cdot x) \\
= \quad & \{\text{by (B1), since } a \neq a'\} \\
& a\backslash(\langle a'\rangle(a\ a')\cdot x) \\
= \quad & \{\text{by definition of } \langle -\rangle -, \text{ since } a' \# (a, x)\} \\
& a\backslash(\langle a\rangle x) \\
= \quad & \{\text{by } \backslash\text{-ALPHA, since } a \# \langle a\rangle x\} \\
& \langle a\rangle x. \quad \square
\end{aligned}
$$

C Proof of Proposition 3.19

We prove

$$e =_\beta e' \ \Rightarrow\ e \Downarrow\equiv\Downarrow e' \tag{C1}$$

first for the $\lambda\alpha\nu$-calculus and then for its extension to the $\lambda\alpha\nu\delta$-calculus.

C.1 $\lambda\alpha\nu$-Calculus

Property (C1) is proved via the logical relation defined in Figure C1. It is not hard to see that this defines an equivariant partial equivalence relation:

$$e \sim_T e' \ \Rightarrow\ \pi\cdot e \sim_T \pi\cdot e' \quad (\pi \in \mathrm{Perm}(\mathbb{A})) \tag{C2}$$

$$e \sim_T e' \ \Rightarrow\ e' \sim_T e \tag{C3}$$

$$e \sim_T e' \wedge e' \sim_T e'' \ \Rightarrow\ e' \sim_T e''. \tag{C4}$$

The proof of transitivity (C4) is by induction on the structure of types, with the induction step for name-abstraction types relying on a 'some/any' property typical of the theory of nominal sets (cf. Theorem 3.8 of Pitts, 2006); in the clause for $\sim_{\mathsf{Name}.T}$ in Figure C1, if the right-hand side holds for some $a \notin fn(e, e')$ then it holds

$$e \sim_T e' \triangleq e \approx_T e' \vee (\exists v, v')\, e \Downarrow_w v \wedge e' \Downarrow_{\mathrm{w}} v' \wedge v \Downarrow\!\equiv\!\Downarrow v'$$

where $e \approx_T e'$ is defined by recursion on the structure of T:

$$\begin{aligned}
e \approx_{T_1 \to T_2} e' &= (\exists x, e_2, e_2')\, e \Downarrow_{\mathrm{w}} \lambda x \to e_2 \wedge e' \Downarrow_{\mathrm{w}} \lambda x \to e_2' \wedge \\
&\qquad (\forall e_1, e_1')\, e_1 \sim_{T_1} e_1' \Rightarrow e_2[e_1/x] \sim_{T_2} e_2'[e_1'/x] \\
e \approx_{T_1 \times \cdots \times T_m} e' &= (\exists e_1, \ldots, e_m, e_1', \ldots, e_m')\, e \Downarrow_{\mathrm{w}} (e_1, \ldots, e_m) \wedge e' \Downarrow_{\mathrm{w}} (e_1', \ldots, e_m') \wedge \\
&\qquad (\forall i \in \{1..m\})\, e_i \sim_{T_i} e_i' \\
e \approx_{\mathsf{Name}.T} e' &= (\exists a \notin fn(e, e'), e_1, e_1')\, e \Downarrow_{\mathrm{w}} \alpha a.\, e_1 \wedge e' \Downarrow_{\mathrm{w}} \alpha a.\, e_1' \wedge e_1 \sim_T e_1' \\
e \approx_G e' &= e \Downarrow\!\equiv\!\Downarrow e'
\end{aligned}$$

Fig. C1. Logical relation, $e \sim_T e'$ ($T \in Typ, e, e' \in Exp(T)$).

for any such a:

$$e \approx_{\mathsf{Name}.T} e' \Leftrightarrow (\forall a \notin fn(e, e'))(\exists e_1, e_1')\, e \Downarrow_{\mathrm{w}} \alpha a.\, e_1 \wedge e' \Downarrow_{\mathrm{w}} \alpha a.\, e_1' \wedge e_1 \sim_T e_1'. \quad \text{(C5)}$$

Remark C1
In Figure C1, one might expect the definition of $e \approx_{T_1 \to T_2} e'$ to begin $(\exists x \notin fv(e, e'), e_2, e_2') \ldots$, mirroring the occurrence of '$a \notin fn(e, e')$' in the definition of $e \approx_{\mathsf{Name}.T} e'$. It makes no difference to the definition of $\approx$ to add this condition on x, but it is not necessary to do so, because x is not in the support of the rest of the defining clause for functions (unlike the defining clause for name-abstractions, where a may occur in the support of the existentially quantified property).

Write $e \Updownarrow_{\mathrm{w}} e'$ if e and e' have the same behaviour with respect to evaluation to weak head normal form:

$$e \Updownarrow_{\mathrm{w}} e' \triangleq (\forall w)\, e \Downarrow_{\mathrm{w}} w \Leftrightarrow e' \Downarrow_{\mathrm{w}} w. \quad \text{(C6)}$$

The following property of the logical relation follows immediately from its definition.

Lemma C2
If $e \Updownarrow_{\mathrm{w}} e'$ and $e' \sim_T e''$, then $e \sim_T e''$. □

Lemma C3
For all $T \in Typ$, $v, v' \in Neu(T)$ and $e, e' \in Exp(T)$

$$v \Downarrow\!\equiv\!\Downarrow v' \Rightarrow v \sim_T v' \quad \text{(C7)}$$

$$e \sim_T e' \Rightarrow e \Downarrow\!\equiv\!\Downarrow e'. \quad \text{(C8)}$$

Proof
Property (C7) follows immediately from the definition in Figure C1. For property (C8), if $e \sim_T e'$, then either $(\exists v, v')\, e \Downarrow_w v \wedge e' \Downarrow_{\mathrm{w}} v' \wedge v \Downarrow\!\equiv\!\Downarrow v'$, in which case $e \Downarrow\!\equiv\!\Downarrow e'$, or $e \approx_T e'$. So it suffices to prove

$$e \approx_T e' \Rightarrow e \Downarrow\!\equiv\!\Downarrow e' \quad \text{(C9)}$$

and we do this by induction on the structure of T. The base case when $T = G$ is a ground type is immediate from the definition of $\approx_G$. The induction steps for product

and name-abstraction types are straightforward. Finally, in case $T = T_1 \to T_2$, if $e \approx_T e'$ then $e \Downarrow_{\mathrm{w}} \lambda x \to e_2$ and $e' \Downarrow_{\mathrm{w}} \lambda x \to e'_2$ hold for some x, e_2, e'_2 satisfying

$$(\forall e_1, e'_1)\ e_1 \sim_{T_1} e'_1 \ \Rightarrow\ e_2[e_1/x] \sim_{T_2} e'_2[e'_1/x]. \tag{C10}$$

Since $x \Downarrow\equiv\Downarrow x$, by Equation (C7), we have $x \sim_{T_1} x$, and therefore, Equation (C10) gives $e_2 \sim_{T_2} e'_2$. Arguing as above, this means that either $e_2 \Downarrow\equiv\Downarrow e'_2$, or $e_2 \approx_{T_2} e'_2$; and in the latter case, by induction hypothesis, we again have $e_2 \Downarrow\equiv\Downarrow e'_2$. So $e_2 \Downarrow\equiv\Downarrow e'_2$ holds and since $e \Downarrow_{\mathrm{w}} \lambda x \to e_2$ and $e' \Downarrow_{\mathrm{w}} \lambda x \to e'_2$, this implies that $e \Downarrow\equiv\Downarrow e'$, as required. □

We next show that the logical relation is compatible with all the expression-forming constructs of the $\lambda\alpha\nu$-calculus.

Lemma C4

(i) If $e \sim_T e'$, then $\nu a.\, e \sim_T \nu a.\, e'$.

(ii) If $x \in \mathbb{V}(T_1), e, e' \in Exp(T_2)$ and $(\forall e_1, e'_1)\ e_1 \sim_{T_1} e'_1 \ \Rightarrow\ e[e_1/x] \sim_{T_2} e'[e'_1/x]$, then $\lambda x \to e \sim_{T_1 \to T_2} \lambda x \to e'$.

(iii) If $(\forall i \in \{1..m\})\ e_i \sim_{T_i} e'_i$, then $(e_1, \ldots, e_m) \sim_{T_1 \times \cdots \times T_m} (e'_1, \ldots, e'_m)$.

(iv) If $e \sim_T e'$, then $\alpha a.\, e \sim_{\mathsf{Name}.T} \alpha a.\, e'$.

(v) If $e \sim_{T_1 \to T_2} e'$ and $e_1 \sim_{T_1} e'_1$, then $e\, e_1 \sim_{T_2} e'\, e'_1$.

(vi) If $e \sim_{T_1 \times \cdots \times T_m} e'$, then $\mathsf{pr}_i\, e \sim_{T_i} \mathsf{pr}_i\, e'$ for all $i \in \{1..m\}$.

(vii) If $e_1 \sim_{\mathsf{Name}.T} e'_1$, $a \notin fn(e_1, e'_1)$, $x \in \mathbb{V}(T), e_2, e'_2 \in Exp(T')$ and $(\forall e, e')\, e \sim_T e' \Rightarrow e_2[e/x] \sim_{T'} e'_2[e'/x]$, then $\mathsf{let}\,\langle a\rangle x = e_1\ \mathsf{in}\ e_2 \sim_{T'} \mathsf{let}\,\langle a\rangle x = e'_1\ \mathsf{in}\ e'_2$.

(viii) If $e_1 \sim_{\mathsf{Bool}} e'_1$, $e_2 \sim_T e'_2$ and $e_3 \sim_T e'_3$, then $\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \sim_T \mathsf{if}\ e'_1\ \mathsf{then}\ e'_2\ \mathsf{else}\ e'_3$.

(ix) If $e_1 \sim_{\mathsf{Name}} e'_1$ and $e_2 \sim_{\mathsf{Name}} e'_2$, then $e_1 = e_2 \sim_{\mathsf{Bool}} e'_1 = e'_2$.

(x) If $e \sim_T e'$, then $(a_1 \wr a_2)e \sim_T (a_1 \wr a_2)e'$.

Proof

Part (i) is proved by induction on the structure of T. At ground types, the property follows from the definitions of $\Downarrow_{\mathrm{w}}$, $\Downarrow_{\mathrm{n}}$ and $a \setminus_{\mathrm{w}} (\_)$ in Figure 9, using the easily verified fact that

$$w \Downarrow\equiv\Downarrow w' \ \Rightarrow\ a \setminus_{\mathrm{w}} w \Downarrow\equiv\Downarrow a \setminus_{\mathrm{w}} w'. \tag{C11}$$

In particular, $v \Downarrow\equiv\Downarrow v'$ implies $\nu a.\, v \Downarrow\equiv\Downarrow \nu a.\, v'$; so for the induction steps at compound types, we just have to show $e \approx_T e' \ \Rightarrow\ \nu a.\, e \sim_T \nu a.\, e'$. We give the argument in case $T = T_1 \to T_2$; the other cases are easier (using property (C5) for the case of name-abstraction types). Suppose $e \Downarrow_{\mathrm{w}} \lambda x \to e_2$ and $e' \Downarrow_{\mathrm{w}} \lambda x \to e'_2$ hold for some x, e_2, e'_2 satisfying Equation (C10). Then, $\nu a.\, e \Downarrow_{\mathrm{w}} \lambda x \to \nu a.\, e_2$ and $\nu a.\, e' \Downarrow_{\mathrm{w}} \lambda x \to \nu a.\, e'_2$; so to prove $\nu a.\, e \approx_T \nu a.\, e'$, it suffices to show that

$$(\forall e_1, e'_1)\ e_1 \sim_{T_1} e'_1 \ \Rightarrow\ (\nu a.\, e_2)[e_1/x] \sim_{T_2} (\nu a.\, e'_2)[e'_1/x]. \tag{C12}$$

Given $e_1 \sim_{T_1} e'_1$, pick $a' \notin fn(a, e_1, e'_1, e_2, e'_2)$. By Equation (C2), we have $(a\ a') \cdot e_1 \sim_T (a\ a') \cdot e'_1$ and hence by (C10), $e_2[(a\ a') \cdot e_1/x] \sim_{T_2} e'_2[(a\ a') \cdot e'_1/x]$. Thus, by induction hypothesis, $\nu a.\, e_2[(a\ a') \cdot e_1/x] \sim_{T_2} \nu a.\, e'_2[(a\ a') \cdot e'_1/x]$; and by choice of a', this is the same as $(a\ a') \cdot ((\nu a.\, e_2)[e_1/x]) \sim_T (a\ a') \cdot ((\nu a.\, e'_2)[e'_1/x])$. So by Equation (C2) again, $(\nu a.\, e_2)[e_1/x] \sim_T (\nu a.\, e'_2)[e'_1/x]$. So we do indeed have Equation (C12), as required.

Parts (ii)–(iv) are straightforward.

For part (v), if $e \sim_{T_1 \to T_2} e'$ and $e_1 \sim_{T_1} e'_1$, then by definition of $\sim_{T_1 \to T_2}$

- either $e \Downarrow_{\mathrm{w}} \lambda x \to e_2$, $e' \Downarrow_{\mathrm{w}} \lambda x \to e'_2$ and Equation (C10) hold: in this case, $e\, e_1 \Updownarrow_{\mathrm{w}} e_2[e_1/x] \sim_{T_2} e'_2[e'_1/x] \Updownarrow_{\mathrm{w}} e'\, e'_1$ and hence $e\, e_1 \sim_{T_2} e'\, e'_1$ by Lemma C2 (and Equation (C3));
- or $e \Downarrow_{\mathrm{w}} v$, $e' \Downarrow_{\mathrm{w}} v'$ and $v \Downarrow\!\equiv\!\Downarrow v'$ hold: in this case, $e\, e_1 \Downarrow_{\mathrm{w}} v\, e_1$ and $e'\, e'_1 \Downarrow_{\mathrm{w}} v'\, e'_1$; furthermore, by Equation (C8), we have $e_1 \Downarrow\!\equiv\!\Downarrow e'_1$, which together with $v \Downarrow\!\equiv\!\Downarrow v'$ yields $v\, e_1 \Downarrow\!\equiv\!\Downarrow v'\, e'_1$.

For part (vi), if $e \sim_{T_1 \times \cdots \times T_m} e'$ then by definition of $\sim_{T_1 \times \cdots \times T_m}$

- either $e \Downarrow_{\mathrm{w}} (e_1, \ldots, e_m)$ and $e' \Downarrow_{\mathrm{w}} (e'_1, \ldots, e'_m)$ with $(\forall i \in \{1..m\})\ e_i \sim_{T_i} e'_i$: in this case, for each $i \in \{1..m\}$, we have $\mathsf{pr}_i\, e \Updownarrow_{\mathrm{w}} e_i \sim_{T_i} e'_i \Updownarrow_{\mathrm{w}} \mathsf{pr}_i\, e'$ and hence $\mathsf{pr}_i\, e \sim_{T_i} \mathsf{pr}_i\, e'$ by Lemma C2 (and Equation (C3));
- or $e \Downarrow_{\mathrm{w}} v$, $e' \Downarrow_{\mathrm{w}} v'$ and $v \Downarrow\!\equiv\!\Downarrow v'$ hold: in this case, $\mathsf{pr}_i\, e \Downarrow_{\mathrm{w}} \mathsf{pr}_i\, v$, $\mathsf{pr}_i\, e' \Downarrow_{\mathrm{w}} \mathsf{pr}_i\, v'$ and $\mathsf{pr}_i\, v \Downarrow\!\equiv\!\Downarrow \mathsf{pr}_i\, v'$.

For part (vii), suppose $e_1 \sim_{\mathsf{Name}.T} e'_1$, $a \notin fn(e_1, e'_1)$, $x \in \mathbb{V}(T)$, $e_2, e'_2 \in Exp(T')$ and $(\forall e, e')\ e \sim_T e' \Rightarrow e_2[e/x] \sim_{T'} e'_2[e'/x]$. By definition of $\sim_{\mathsf{Name}.T}$

- either $e_1 \approx_{\mathsf{Name}.T} e'_1$ holds: in this case, by Equation (C5), $e_1 \Downarrow_{\mathrm{w}} \nu a.\, e$ and $e'_1 \Downarrow_{\mathrm{w}} \nu a.\, e'$ hold for some e, e' satisfying $e \sim_T e'$ and hence also $\nu a.\, e_2[e/x] \sim_{T'} \nu a.\, e'_2[e'/x]$ by assumption on e_2, e'_2 and using part (i); but $\mathsf{let}\,\langle a\rangle x = e_1\ \mathsf{in}\ e_2 \Updownarrow_{\mathrm{w}} \nu a.\, e_2[e/x]$ and $\mathsf{let}\,\langle a\rangle x = e'_1\ \mathsf{in}\ e'_2 \Updownarrow_{\mathrm{w}} \nu a.\, e'_2[e'/x]$, and therefore, by Lemma C2, $\mathsf{let}\,\langle a\rangle x = e_1\ \mathsf{in}\ e_2 \sim_{T'} \mathsf{let}\,\langle a\rangle x = e'_1\ \mathsf{in}\ e'_2$;
- or $e_1 \Downarrow_{\mathrm{w}} v$, $e_1 \Downarrow_{\mathrm{w}} v'$ and $v \Downarrow\!\equiv\!\Downarrow v'$ hold: in this case, $\mathsf{let}\,\langle a\rangle x = e_1\ \mathsf{in}\ e_2 \Downarrow_{\mathrm{w}} \mathsf{let}\,\langle a\rangle x = v\ \mathsf{in}\ e_2$ and $\mathsf{let}\,\langle a\rangle x = e'_1\ \mathsf{in}\ e'_2 \Downarrow_{\mathrm{w}} \mathsf{let}\,\langle a\rangle x = v'\ \mathsf{in}\ e'_2$; by assumption on e_2, e'_2, we have $e_2 = e_2[x/x] \sim_{T'} e'_2[x/x] = e'_2$ (since $x \sim_T x$ holds by Equation (C7)), hence $e_2 \Downarrow\!\equiv\!\Downarrow e'_2$ by Equation (C8), and therefore, $\mathsf{let}\,\langle a\rangle x = v\ \mathsf{in}\ e_2 \Downarrow\!\equiv\!\Downarrow \mathsf{let}\,\langle a\rangle x = v'\ \mathsf{in}\ e'_2$.

For part (viii), if $e_1 \sim_{\mathsf{Bool}} e'_1$, $e_2 \sim_T e'_2$ and $e_3 \sim_T e'_3$, then by definition of $\sim_{\mathsf{Bool}}$

- either $e_1 \Downarrow_{\mathrm{w}} \mathsf{True}$ and $e'_1 \Downarrow_{\mathrm{w}} \mathsf{True}$: in this case, $\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Updownarrow_{\mathrm{w}} e_2 \sim_T e'_2 \Updownarrow_{\mathrm{w}} \mathsf{if}\ e'_1\ \mathsf{then}\ e'_2\ \mathsf{else}\ e'_3$ and we can apply Lemma C2 to get $\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \sim_T \mathsf{if}\ e'_1\ \mathsf{then}\ e'_2\ \mathsf{else}\ e'_3$;
- or $e_1 \Downarrow_{\mathrm{w}} \mathsf{False}$ and $e'_1 \Downarrow_{\mathrm{w}} \mathsf{False}$: this case is similar to the previous one;
- or $e_1 \Downarrow_{\mathrm{w}} v$, $e'_1 \Downarrow_{\mathrm{w}} v'$ and $v \Downarrow\!\equiv\!\Downarrow v'$ hold: in this case, $\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow_{\mathrm{w}} \mathsf{if}\ v\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$ and $\mathsf{if}\ e'_1\ \mathsf{then}\ e'_2\ \mathsf{else}\ e'_3 \Downarrow_{\mathrm{w}} \mathsf{if}\ v'\ \mathsf{then}\ e'_2\ \mathsf{else}\ e'_3$; and since $e_2 \Downarrow\!\equiv\!\Downarrow e'_2$ and $e_3 \Downarrow\!\equiv\!\Downarrow e'_3$ hold by Equation (C8), we also have $\mathsf{if}\ v\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow\!\equiv\!\Downarrow \mathsf{if}\ v'\ \mathsf{then}\ e'_2\ \mathsf{else}\ e'_3$.

Part (ix) is proved similarly to part (viii).

Finally, part (x) is proved by induction on the structure of T. At ground types, the property follows from the definitions of $\Downarrow_{\mathrm{w}}$, $\Downarrow_{\mathrm{n}}$ and $(a_1\ a_2)_{\mathrm{w}}(\_)$ in Figure 9, using the easily verified fact that

$$w \Downarrow\!\equiv\!\Downarrow w' \Rightarrow (a_1\ a_2)_{\mathrm{w}} w \Downarrow\!\equiv\!\Downarrow (a_1\ a_2)_{\mathrm{w}} w'. \tag{C13}$$

In particular, $v \Downarrow\!\equiv\!\Downarrow v'$ implies $(a_1 \wr a_2)v \Downarrow\!\equiv\!\Downarrow (a_1 \wr a_2)v'$; so for the induction steps at compound types, we just have to show $e \approx_T e' \Rightarrow (a_1 \wr a_2)e \sim_T (a_1 \wr a_2)e'$, the proof of which is straightforward – in the case of name-abstraction types using property (C5) to pick a representative bound name a in $e \Downarrow_{\mathrm{w}} \nu a.\, e_1$ that is not equal to a_1 or a_2 so that $(a_1 \wr a_2)e \Downarrow_{\mathrm{w}} \nu a.\, (a_1 \wr a_2)e_1$ holds (and similarly for e'_1). □

Definition C5 (*Substitutions.*)
A (finite) *substitution* $\sigma \in Sub$ is by definition a function from variables to well-typed $\lambda\alpha v$-expressions that respects typing ($x \in \mathbb{V}(T) \Rightarrow \sigma(x) \in Exp(T)$) and satisfies $\sigma(x) = x$ for all but finitely many variables x. We write $e[\sigma]$ for the result of applying to e the simultaneous substitution specified by σ. Two substitutions are logically related, $\sigma \sim \sigma'$, if for all $T \in Typ$ and $x \in \mathbb{V}(T)$, it is the case that $\sigma(x) \sim_T \sigma'(x)$.

Corollary C6 (*'Fundamental property' of the logical relation.*)
For all $T \in Typ$, $e \in Exp(T)$ and $\sigma, \sigma' \in Sub$

$$\sigma \sim \sigma' \Rightarrow e[\sigma] \sim_T e[\sigma']. \tag{C14}$$

Proof
By α-structural induction (Pitts, 2006) for e. For the base case $e = x \in \mathbb{V}(T)$, we need $x \sim_T x$, which follows from Equation (C7). The other bases cases, $a \sim_{\mathsf{Name}} a$, $\mathsf{True} \sim_{\mathsf{Bool}} \mathsf{True}$ and $\mathsf{False} \sim_{\mathsf{Bool}} \mathsf{False}$ are immediate from the definition in Figure C1. The induction steps follow from Lemma C4. □

Lemma C7
For all $T \in Typ$ and $e, e' \in Exp(T)$, if $e =_\beta e'$ then $e \sim_T e'$.

Proof
It suffices to show that the logical relation is a $\lambda\alpha v$-calculus congruence containing the basic β-conversions in Figure 5. First note that the identity substitution satisfies $\sigma \sim \sigma$ (since we have $x \sim_T x$ for all $x \in \mathbb{V}(T)$, by Equation (C7)). Therefore, Corollary C6 implies that the logical relation is reflexive:

$$(\forall T \in Typ, e, e \in Exp(T))\ e \sim_T e. \tag{C15}$$

We have already noted that it is symmetric (C3) and transitive (C4); and Lemma C4 and Corollary C6 together show that it respects the expression-forming constructs of the $\lambda\alpha v$-calculus (modulo α-equivalence). So the logical relation is indeed a congruence. To see that it contains the conversions in Figure 5, note that (in view of Equation (41)) they are all instances of the $\Updownarrow_{\mathrm{w}}$ relation; but combining Lemma C2 with Equation (C15), we have that $e \Updownarrow_{\mathrm{w}} e'$ implies $e \sim_T e'$. □

Combining property (C8) in Lemma C3 with Lemma C7, we have proved Equation (C1). □

C.2 $\lambda\alpha v\delta$-Caclulus

When passing from the $\lambda\alpha v$-calculus to the $\lambda\alpha v\delta$-calculus, the definition of the logical relation $e \sim_T e'$ (Figure C1) is unchanged, since it is independent of the nature of ground types. However, three new clauses have to be added to Lemma C4:

(xi) *If* $e \sim_T e'$, *then* $K\, e \sim_G K\, e'$ *(where* $K \in \{\mathsf{V}, \mathsf{A}, \mathsf{L}, \mathsf{Z}, \mathsf{S}\}$ *and* $K : T \to G$*).*

(xii) *If* $e_1 \sim_{\mathsf{Name}\to T} e_1'$, $e_2 \sim_{(T\times T)\to T} e_2'$, $e_3 \sim_{(\mathsf{Name}.T)\to T} e_3'$ *and* $e \sim_{\mathsf{Lam}} e'$, *then* $\mathsf{lrec}\, e_1\, e_2\, e_3\, e \sim_T \mathsf{lrec}\, e_1'\, e_2'\, e_3'\, e'$.

(xiii) *If* $e_1 \sim_{1\to T} e_1'$, $e_2 \sim_{T\to T} e_2'$ *and* $e \sim_{\mathsf{Nat}} e'$, *then* $\mathsf{nrec}\, e_1\, e_2\, e \sim_T \mathsf{nrec}\, e_1'\, e_2'\, e'$.

The proof of (xi) is straightforward, using property (C8). However, the proofs of (xii) and (xiii) are more involved because of the inductive nature of normal forms of types Lam and Nat. Recall from Figure C1 that at ground types G, the logical relation coincides with the relation $\Downarrow\equiv\Downarrow$. Write $e \sim_G^{(k)} e'$ if $e \Downarrow_{\mathrm{n}} n$, $e' \Downarrow_{\mathrm{n}} n'$ and $n \equiv n'$ hold for some n, n' with $e \Downarrow_{\mathrm{n}} n$ and $e' \Downarrow_{\mathrm{n}} n'$ having proofs of height $\leqslant k$. Then, one can prove (xii) by proving

$$e \sim_{\mathsf{Lam}}^{(k)} e' \;\Rightarrow\; (\forall e_1, e_1', e_2, e_2', e_3, e_3')\; e_1 \sim_{\mathsf{Name}\to T} e_1' \;\wedge\; e_2 \sim_{(T\times T)\to T} e_2' \;\wedge\; e_3 \sim_{(\mathsf{Name}.T)\to T} e_3' \;\Rightarrow\; \mathsf{lrec}\, e_1\, e_2\, e_3\, e \sim_T \mathsf{lrec}\, e_1'\, e_2'\, e_3'\, e'$$

by induction on $k \in \mathbb{N}$. Similarly, for (xiii).

The extended Lemma C4 gives the fundamental property of the logical relation (Corollary C6) for $\lambda\alpha\nu\delta$-calculus and the fact that it contains β-conversion (Lemma C7). Then, as before, we get Equation (C1) by combining property (C8) and Lemma C7. □

Acknowledgements

Research partially supported by UK EPSRC grant EP/D000459/1.

References

Altenkirch, T. & Chapman, J. (2009) Big-step normalisation. *J. Funct. Prog.* **19**(3–4), 311–333.

Barendregt, H. P. (1984) *The Lambda Calculus: Its Syntax and Semantics*, Revised ed. North-Holland.

Cardelli, L. & Gordon, A. D. (2001) Logical properties of name restriction. In *Proceedings of the 5th International Conference on Typed Lambda Calculus and Applications*, Abramsky, S. (ed), Lecture Notes in Computer Science, vol. 2044. Berlin: Springer-Verlag, pp. 46–60.

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

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

Coquand, T. (1991) An algorithm for testing conversion in type theory., In *Logical Frameworks*, Huet, G. & Plotkin, G. D. (eds). New York, NY, USA: Cambridge University Press, pp. 255–228.

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

Fernández, M. & Gabbay, M. J. (2005) Nominal rewriting with name generation: Abstraction vs. locality. In *Proceedings of the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming PPDP 2005*. ACM Press, pp. 47–58.

Fiore, M. P., Plotkin, G. D. & Turi, D. (1999) Abstract syntax and variable binding. In *Proceedings of the 14th Annual Symposium on Logic in Computer Science*. Washington: IEEE Computer Society Press, pp. 193–202.

Friedman, H. (1975) Equality between functionals. In *Logic Colloquium*, Parikh, R. (ed), Lecture Notes in Mathematics, vol. 453. Berlin/Heidelberg: Springer, pp. 22–37.

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

Gabbay, M. J. & Lengrand, S. (December 2009) The lambda-context calculus (extended version). *Inform. comput.* **207**, 1369–1400.

Gabbay, M. J. & Pitts, A. M. (2002) A new approach to abstract syntax with variable binding. *Form. Asp. Comput.* **13**, 341–363.

Gacek, A., Miller, D. & Nadathur, G. (2008) Combining generic judgments with recursive definitions. In *Proceedings of the 23rd IEEE Symposium on Logic in Computer Science, LICS 2008*. IEEE Computer Society Press, pp. 33–44.

Gödel, K. (1958) Über eine bisher noch nicht benüzte Erweiterung des finiten Standpunktes. *Dialectica* **12**, 280–287.

Gordon, A. D. & Melham, T. (1996) Five axioms of alpha-conversion. In *Proceedings of the 9th International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, vol. 1125. Springer-Verlag, pp. 173–191.

Grégoire, B. & Leroy, X. (2002) A compiled implementation of strong reduction. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (Pittsburgh, PA, USA)*. ACM Press, pp. 235–246.

Harper, R. & Pfenning, F. (2005) On equivalence and canonical forms in the LF type theory. *ACM Trans. Comput. Log.* **6**, 61–101.

Johnstone, P. T. (2002) *Sketches of an Elephant, a Topos Theory Compendium,* Vol. 1–2. Oxford Logic Guides, nos. 43–44. Oxford University Press.

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

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

Miller, D. A. (1990) An extension to ML to handle bound variables in data structures. In Proceedings of the Logical Frameworks BRA Workshop, Technical Report MS-CIS-90-59. University of Pennsylvania.

Miller, D. A. & Tiu, A. (2005) A proof theory for generic judgments. *ACM Trans. Comput. Log.* **6**(4), 749–783.

Milner, R. (1992) Functions as processes. *Math. Struct. Comput. Sci.* **2**(2), 119–141.

Moggi, E. (1991) Notions of computation and monads. *Inf. Comput.*, **93**(1), 55–92.

Nordström, B., Petersson, K. & Smith, J. M. (1990) *Programming in Martin-Löf's Type Theory*. Oxford University Press.

Norrish, M. (2004) Recursive function definition for types with binders. In *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science, vol. 3223. Springer-Verlag, pp. 241–256.

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

Pfenning, F. (2001) Logical frameworks. In *Handbook of Automated Reasoning*, Robinson, A. & Voronkov, A. (eds). Elsevier Science and MIT Press, Chapter 17, pp. 1063–1147.

Pfenning, F. & Elliott, C. (1988) Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, pp. 199–208.

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

Pitts, A. M. (2003) Nominal logic, a first order theory of names and binding. *Inf. Comput.* **186**, 165–193.

Pitts, A. M. (2006) Alpha-structural recursion and induction. *J. ACM* **53**, 459–506.

Pitts, A. M. (2010) Nominal System T. In *Proceedings of the 37th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 2010 (Madrid, Spain)*. ACM Press, pp. 159–170.

Pitts, A. M. & Gabbay, M. J. (2000) A metalanguage for programming with bound names modulo renaming. In *Proceedings of the 5th International Conference on Mathematics of Program Construction, MPC 2000, (Ponte De Lima, Portugal, July 2000)*, Backhouse, R. & Oliveira, J. N. (eds), Lecture Notes in Computer Science, vol. 1837. Heidelberg: Springer-Verlag, pp. 230–255.

Pitts, A. M. & Stark, I. D. B. (1993) Observable properties of higher order functions that dynamically create local names, or: What's new? In *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science (Gdańsk, 1993)*, Lecture Notes in Computer Science, vol. 711. Springer-Verlag, Berlin, pp. 122–141.

Pitts, A. M. & Stark, I. D. B. (1998) Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*, Gordon, A. D. & Pitts, A. M. (eds), Publications of the Newton Institute. Cambridge University Press, pp. 227–273.

Poswolsky, A. & Schürmann, C. (2008) Practical programming with higher-order encodings and dependent types. In *Proceedings of the European Symposium on Programming, ESOP 2008*, Lecture Notes in Computer Science, vol. 4960. Springer-Verlag, pp. 93–107.

Pottier, F. (2007) Static name control for FreshML. In *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science, LICS 2007*. Wroclaw, Poland: IEEE Computer Society Press, pp. 356–365.

Pouillard, N. & Pottier, F. (2010) A fresh look at programming with names and binders. In *Proceedings of the Fifteenth ACM SIGPLAN International Conference on Functional Programming, ICFP 2010*. ACM Press, pp. 217–228.

Schöpp, U. & Stark, I. D. B. (2004) A dependent type theory with names and binding. In *Proceedings of the Computer Science Logic, CSL 2004 (Karpacz, Poland)*, Lecture notes in Computer Science, vol. 3210. Springer-Verlag, pp. 235–249.

Schürmann, C., Despeyroux, J. & Pfenning, F. (2001) Primitive recursion for higher-order abstract syntax. *Theor. Comput. Sci.* **266**, 1–57.

Shinwell, M. R. & Pitts, A. M. (February 2005) *Fresh Objective Caml User Manual*, Techical Report UCAM-CL-TR-621. University of Cambridge Computer Laboratory.

Shinwell, M. R., Pitts, A. M. & Gabbay, M. J. (2003) FreshML: Programming with binders made simple. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003 (Uppsala, Sweden)*. ACM Press, pp. 263–274.

Tait, W. W. (1967) Intensional interpretation of functionals of finite type, I. *J. Symb. Log.* **32**(2), 198–212.

Urban, C. (2008) Nominal reasoning techniques in Isabelle/HOL. *J. Autom. Reason.* **40**(4), 327–356.

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

Urban, C., Cheney, J. & Berghofer, S. (2011) Mechanizing the metatheory of LF. *ACM Trans. Comput. Log.* **12**(15), 1–42.

Urban, C., Pitts, A. M. & Gabbay, M. J. (2004) Nominal unification. *Theor. Comput. Sci.* **323**, 473–497.

Westbrook, E., Stump, A. & Austin, E. (2009) 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. ACM Press, pp. 74–83.