Full Abstraction for Nominal Scott Domains

Steffen Lösch   Andrew M. Pitts

University of Cambridge

{steffen.loesch,andrew.pitts}@cl.cam.ac.uk

Abstract

We develop a domain theory within nominal sets and present programming language constructs and results that can be gained from this approach. The development is based on the concept of *orbit-finite* subset, that is, a subset of a nominal sets that is both finitely supported and contained in finitely many orbits. This concept appears prominently in the recent research programme of Bojańczyk *et al.* on automata over infinite languages, and our results establish a connection between their work and a characterisation of topological compactness discovered, in a quite different setting, by Winskel and Turner as part of a nominal domain theory for concurrency. We use this connection to derive a notion of Scott domain within nominal sets. The functionals for existential quantification over names and 'definite description' over names turn out to be compact in the sense appropriate for nominal Scott domains. Adding them, together with parallel-or, to a programming language for recursively defined higher-order functions with name abstraction and locally scoped names, we prove a full abstraction result for nominal Scott domains analogous to Plotkin's classic result about PCF and conventional Scott domains: two program phrases have the same observable operational behaviour in all contexts if and only if they denote equal elements of the nominal Scott domain model. This is the first full abstraction result we know of for higher-order functions with local names that uses a domain theory based on ordinary extensional functions, rather than using the more intensional approach of game semantics.

Categories and Subject Descriptors F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Lambda calculus and related systems; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Denotational semantics

General Terms Languages, Theory

Keywords Denotational Semantics, Domain Theory, Full Abstraction, Nominal Sets, Symmetry

1. Introduction

Various forms of symmetry are used in many branches of mathematics and computer science. The results in this paper have to do with using symmetry to extend the reach of computation theory from finite data structures and algorithms to ones that, although they are infinite, *become finite when quotiented by a suitable no-*

tion of symmetry. We focus on higher-order functional computation with data that may involve unboundedly many different names and symmetries given by permutations of those names. A simple example of such data is the abstract syntax trees for a language involving binding constructs, such as the λ-calculus with named bound variables: infinitely many abstract syntax trees represent a particular λ-term, modulo permuting their bound names. This way of viewing α-equivalence via symmetry was the initial stimulus for the development of *nominal sets* [14]—a theory for mathematical structures involving atomic names (that is, names whose only attribute is their identity) based on name permutations and the notion of *finite support*; we review this concept in Section 2.

Nominal sets have been used to develop the semantic properties of binders and locally scoped names, with applications to functional and logic programming, to equational logic and rewriting, to type theory and to interactive theorem proving; see the bibliography [1]. The work by Montanari *et al.* [22] on the π-calculus and HD-automata provides a somewhat different application of nominal sets (an independent one, since it uses a notion of 'named set' that only subsequently was shown to be equivalent to nominal sets [15, 37]). The use of symmetries of names (of fresh communication channels in this case) to get finite representations of infinitely many states is at the forefront in their work. It has recently been subsumed and generalised in a programme of what one might call 'orbit-finite' automata theory [7, 8, 13, 23, 42].

In this paper we bring together the 'names and binders' and the 'orbit-finite state space' aspects of nominal sets. We observe that a key concept underlying the automata-theoretic research programme of Bojańczyk *et al.* [6], that of being an *orbit-finite subset*, turns out to subsume a notion of topological compactness introduced, for quite different purposes, by Winskel and Turner in their work on nominal domain theory for concurrency [40]. We explain the connection and use it to develop a version of the classic notion of *Scott domain* within nominal sets. (Previous work on denotational semantics with nominal sets [34, 39] has focussed on simpler notions of domain, analogous either to ω-chain complete posets, or to algebraic lattices.) A domain element is compact if it stands for a finite approximation of a computation. We define this notion for our setting (in Definition 5) and show that the functionals for 'there exists a name such that...' and 'the unique name such that...' are compact. Plotkin [30] famously proves that PCF with parallel-or is *fully abstract* with respect to conventional Scott domains, in the sense that two expressions have equal denotations if and only if they the have the same observable operational behaviour in all contexts. We obtain an analogous result for nominal Scott domains, through adding the above functionals, together with parallel-or, to a programming language for recursively defined higher-order functions with name-abstractions and locally scoped names. Thus this paper makes the following contributions.

- We show (Theorem 8) that a finitely supported subset of a nominal set is compact with respect to unions that are uniform-

directed in the sense of Winskel and Turner if and only if it is orbit-finite.

- We use orbit-finite subsets to generalise the notion of Scott domain from ordinary sets to nominal sets; we prove that the category of nominal Scott domains is cartesian closed, has least fixed points and is closed under forming domains of name abstractions (Sections 5 and 6). Although there are infinitely many names, the nominal Scott domain of names has some strong finiteness properties. In particular, we show that the functionals for existential quantification over names and definite description of names are uniform-compact elements of their function domains (Examples 14 and 15).

- We define a language PNA (Programming with Name Abstractions) that extends Plotkin's PCF language with names that can be locally scoped, swapped, abstracted and concreted. In order to illustrate these facilities for programming with name abstractions, PNA has a nominal algebraic datatype for representing λ-terms. PNA's operational semantics is inspired by [19, 27]; in particular, its method for deconstructing name abstractions makes use of Odersky-style, 'scope-intrusive' local names. We give a simple denotational semantics for PNA using nominal Scott domains and prove that it is computationally adequate (Theorem 27).

- We extend PNA to a language PNA$^+$ with syntax, operational semantics and denotational semantics for parallel-or, for existentially quantifying over names and for definite descriptions of names. We prove that the nominal Scott domain model of PNA$^+$ is fully abstract: any two expressions are contextually equivalent if and only if they have equal denotations in the model (Theorem 30).

There are full abstraction results for higher-order functions with local names using the intensional approach of game semantics [3, 18, 41], but our Theorem 30 is the first such result we know of that is based on ordinary extensional functions. There is no similar result known for FreshML [35], which uses generative rather than Odersky-style local names to implement the features that PNA provides for programming with name abstractions; and yet PNA (extended with recursive types) is in principle as expressive as FreshML, in view of Lösch and Pitts [19]. Our proof of Theorem 30 seems novel compared with other proofs of full abstraction results in the literature. We only sketch it here—the details will be published elsewhere. On the other hand it gives rise to some open problems that we discuss in Section 10, together with a number of possibilities for future work exploiting the use of orbit-finite subsets within nominal domain theory.

2. Finite support

We are interested in the denotational semantics of programs written in languages featuring binding constructs involving names that can be tested for equality. To take symmetry into account, we fix some countably infinite set $\mathbb{A}$ (of 'atomic names') and consider finite permutations of $\mathbb{A}$, that is, bijections $\pi : \mathbb{A} \cong \mathbb{A}$ with the property that $\pi\, a = a$ holds for all but finitely many $a \in \mathbb{A}$. Recall that an *action* of such permutations on a set X is a binary operation $\_ \cdot \_$ satisfying $\mathrm{id} \cdot x = x$ (where id is the identity permutation) and $\pi' \cdot (\pi \cdot x) = (\pi' \circ \pi) \cdot x$ (where $\_ \circ \_$ is composition).

Actions of finite permutations of $\mathbb{A}$ on sets X and Y can be extended to their cartesian product $X \times Y$ by defining for each $x \in X$ and $y \in Y$

$$\pi \cdot (x, y) \triangleq (\pi \cdot x, \pi \cdot y). \tag{1}$$

More interestingly, given actions on X and Y, we get an action on the set of functions Y^X by defining for each $f \in Y^X$

$$\pi \cdot f \triangleq \lambda x \in X \to \pi \cdot (f(\pi^{-1} \cdot x)) \tag{2}$$

where π^{-1} is the inverse of the permutation π. In particular, taking $Y = 2 = \{\mathrm{true}, \mathrm{false}\}$, a two-element set with trivial action ($\pi \cdot \mathrm{true} = \mathrm{true}$, $\pi \cdot \mathrm{false} = \mathrm{false}$), we get an action on 2^X and hence on subsets of X: for each $S \subseteq X$

$$\pi \cdot S \triangleq \{\pi \cdot x \mid x \in S\}. \tag{3}$$

Programs, being finite syntactic objects, only involve finitely many atomic names in their construction; whereas the elements of a set X used to denote program behaviours may well be infinite mathematical objects. We wish to limit our attention to infinite behaviours that depend only upon finitely many atomic names, as doing so yields a richer and better behaved theory. We can make precise what it means to 'only depend upon finitely many atomic names' entirely in terms of symmetry, that is, in terms of the permutation action. An element $x \in X$ is *supported* by a set $A \subseteq \mathbb{A}$ of atomic names if every permutation π satisfying $(\forall a \in A)\, \pi\, a = a$ also satisfies $\pi \cdot x = x$. That is, permutations that preserve A also preserve x. We say that a set X equipped with an action of finite permutations of $\mathbb{A}$ is a *nominal set* if each of its elements is supported by some finite set of atomic names. In this case one can show that for each $x \in X$ there is a smallest finite subset of $\mathbb{A}$ supporting x, which we write as $\mathrm{supp}\, x$ [14]. We also write $a \# x$ to mean $a \notin \mathrm{supp}\, x$. Note that since $\mathrm{supp}\, x$ is a finite set and $\mathbb{A}$ is not, given x we can always find some $a \in \mathbb{A}$ satisfying $a \# x$.

Given a nominal set X, the subsets that possess a finite support with respect to the action in (3) are called *finitely supported subsets* of X. Not every subset is finitely supported. For example, when $X = \mathbb{A}$ (with action $\pi \cdot a = \pi\, a$), the only finitely supported subsets are those $S \subseteq \mathbb{A}$ for which either S, or $\mathbb{A} - S$ is finite. We write $\mathrm{P}_{\mathrm{fs}} X$ for the collection of all finitely supported subsets of X, and with the action in (3) this is a nominal set. Indeed it is the power object (in the sense of topos theory [17]) for a model of higher-order logic based on nominal sets. The main difference between this model and the classical one is that it fails to satisfy choice principles.[1] As we discuss next, this difference causes nominal domain theory to be something more than just 'classical domain theory carried out in the nominal model of higher-order logic'.

3. Uniform-directed joins

A key idea behind domain theory [5] is to give denotations to programs with potentially infinite behaviour as a limit of approximations. For domain theory based on approximation via a partial order (rather than a metric), limits are joins of chains (linearly ordered subsets), or more generally, joins of directed subsets (where every finite set of elements has an upper bound in the subset). So long as one considers chains of arbitrary (ordinal) length, classically there is no difference between using joins of chains and using directed joins [20]. However, the equivalence of the two approaches relies on the Axiom of Choice and, as we noted above, that fails to hold for nominal sets. Therefore, in the nominal version of domain theory, formulating limits in terms of joins of chains is more restrictive than using joins of arbitrary directed subsets. (Of course, both the chains and the directed subsets should be finitely supported, to make sense nominally.) Winskel and Turner [40] provide a compelling reason for restricting attention to joins of chains. They show that a key notion provided by the nominal approach, the operation

---

[1] The associated model of set theory goes back to work in the 1930s by Fraenkel and Mostowski, who devised it specifically to negate the Axiom of Choice [see 12, Remark 2.22].

of name abstraction, preserves joins of chains, but does not necessarily preserve joins of directed subsets in general. We give below a simplified version of the Winskel-Turner example of the failure of name abstraction to preserve joins of all finitely supported, directed subsets.

Definition 1 (name abstraction). A *nominal poset* is simply a nominal set D equipped with a partial order $\sqsubseteq$ that is respected by the action of permutations: $d \sqsubseteq d' \Rightarrow \pi \cdot d \sqsubseteq \pi \cdot d'$. Given such a D, we get a pre-order on $\mathbb{A} \times D$ by defining $(a, d) \sqsubseteq (a', d')$ to hold whenever we have $(a\ a'') \cdot d \sqsubseteq (a'\ a'') \cdot d'$ in D for some (or indeed, for any) $a'' \# a, a', d, d'$. (As usual, $(a\ a')$ denotes the permutation that swaps a and a', leaving all other atomic names fixed.) We write $[\mathbb{A}]D$ for the poset obtained by quotienting $\mathbb{A} \times D$ by the equivalence relation associated with this pre-order, and $\langle a \rangle d$ for the equivalence class of (a, d). Defining a permutation action by $\pi \cdot \langle a \rangle d = \langle \pi a \rangle (\pi \cdot d)$, one can show that $[\mathbb{A}]D$ is also a nominal poset, with $\mathrm{supp} \langle a \rangle d = (\mathrm{supp}\, d) - \{a\}$. An element of $[\mathbb{A}]D$ is an abstract form of α-equivalence class for elements of D. It is abstract, because D itself may not consist of syntactic data—we just need to know how name permutations act on its elements.

Example 2. For any nominal set X, partially ordering the elements of the nominal set $\mathrm{P_{fs}}X$ of finitely supported subsets of X by inclusion, we get a nominal poset. Consider the case when $X = \mathbb{A}$. Given $a \in \mathbb{A}$, the name abstraction function $\mathrm{P_{fs}}\mathbb{A} \to [\mathbb{A}](\mathrm{P_{fs}}\mathbb{A})$ mapping each $S \in \mathrm{P_{fs}}\mathbb{A}$ to $\langle a \rangle S$ does not preserve all joins of finitely supported, directed subsets. For example, consider the directed subset $\mathcal{F} \in \mathrm{P_{fs}}(\mathrm{P_{fs}}\mathbb{A})$ consisting of all finite sets of atomic names. $\mathcal{F}$ has empty support and its join $\bigsqcup \mathcal{F}$ is equal to $\mathbb{A}$. However, fixing upon $a \neq a'$ in $\mathbb{A}$, one has $\langle a \rangle \mathbb{A} = \langle a' \rangle \mathbb{A} \neq \langle a' \rangle (\mathbb{A} - \{a\})$ and one can check that $\bigsqcup \{\langle a \rangle F \mid F \in \mathcal{F}\} \sqsubseteq \langle a' \rangle (\mathbb{A} - \{a\})$. So $\langle a \rangle (\bigsqcup \mathcal{F}) \neq \bigsqcup \{\langle a \rangle F \mid F \in \mathcal{F}\}$.

A motivation for using nominal rather than ordinary domains to do denotational semantics is precisely to gain access to this operation of name abstraction, which can be used to model language features involving binders. So finding a setting in which name abstraction preserves limits of approximations is crucial. It turns out that the problem with the directed subset $\mathcal{F}$ in the above example is its lack of what Winskel and Turner call *uniformity*: each $F \in \mathcal{F}$ is a finite set of atomic names and hence is supported by F itself. Thus, there is no single finite support for all the elements of $\mathcal{F}$.

Definition 3 (udcpos). Given a nominal set X, call a subset $S \subseteq X$ *uniformly supported* if there is a finite set $A \subseteq_{\mathrm{f}} \mathbb{A}$ that supports each $x \in S$. A *uniform-directed* subset of a nominal poset is a subset S that is both uniformly supported and directed. A *uniform-directed complete partial order* (*udcpo*) is a nominal poset that has joins $\bigsqcup S$ for all uniform-directed subsets S.

Lemma 4. *In a nominal poset D, every finitely supported chain C is necessarily uniformly supported. In particular, each $d \in C$ is supported by $\mathrm{supp}\, C$.* $\quad\square$

For a proof, see Turner [39, Lemma 3.4.2.1]. As Turner points out, using this lemma, the classic result of Markowsky [20] can be extended to show that *a nominal poset is a udcpo if and only if it has joins for all finitely supported chains.* So in effect udcpos give us a domain theory within the higher-order logic of nominal sets based on chain-completeness. As we will see in Section 6, they also give us access to the name-abstraction construct.

We model potentially infinite program behaviours in languages with names using denotations that are uniform-directed joins of approximations to the behaviour. Each approximation should be finite in a suitable sense. For classical domain theory this amounts to being compact (also known as 'isolated') with respect to directed joins. By analogy, we make the following definition.

Definition 5 (algebraic udcpo). An element $u \in D$ of a udcpo D is *uniform-compact* if for all uniform-directed subsets $S \subseteq D$, $u \sqsubseteq \bigsqcup S \Rightarrow (\exists d \in S)\ u \sqsubseteq d$. We write $\mathrm{K}D$ for the set of uniform-compact elements of D. We say that D is an *algebraic* udcpo if each of its elements is the join of a uniform-directed subset of $\mathrm{K}D$. D is ω-*algebraic* if in addition the underlying set of $\mathrm{K}D$ is countable.

Recall that a subset of a set is compact with respect to directed joins (unions) of subsets if and only if it is a finite set. Here we are restricting attention to a smaller class of joins, the uniform-directed ones. Therefore, we expect uniform-compactness to be a more liberal notion of finiteness. We show in the next section that it corresponds precisely to the notion of orbit-finite subset introduced by Bojańczyk *et al.*[2]

4. Orbit-finite subsets

The action of finite permutations of $\mathbb{A}$ on the elements of a nominal set X partitions it into *orbits*: two elements x and x' are in the same orbit if $x' = \pi \cdot x$ for some finite permutation π. For example $\mathbb{A}$ itself has just one orbit. $\mathbb{A} \times \mathbb{A}$ has two, namely $\{(a, a) \mid a \in \mathbb{A}\}$ and $\{(a, a') \in \mathbb{A}^2 \mid a \neq a'\}$, and in general $\mathbb{A}^n$ has always finitely many orbits. In contrast, the nominal set $\mathbb{A}^*$ of finite lists of atomic names has infinitely many orbits (since lists of different length cannot be in the same orbit).

Definition 6 (orbit-finite subsets). A finitely supported subset $S \in \mathrm{P_{fs}}X$ of a nominal set X is said to be *orbit-finite* if it is contained in the union of finitely many orbits of X.

Bojańczyk *et al.* investigate orbit-finite data structures and algorithms (for a generalised version of nominal sets over any 'Fraïssé symmetry'). Note that an orbit-finite subset may well have infinitely many different elements. For example, $\mathbb{A}$ is an orbit-finite subset of itself. In order to compute with orbit-finite subsets one needs an effective presentation of them and their operations. The following notion turns out to give an alternative characterisation of orbit-finite subsets that is suitable for calculation. It was introduced independently by Turner [39, Definition 3.4.3.2], Gabbay [11, Section 3.3; 12, Definition 3.1; 13, Definition 3.1] and Bojańczyk *et al.* [6, Section 8], whose 'hull' terminology we adopt here. (See also Ciancia and Montanari [9, Definition 6.10], whose 'closures' are hulls of the form $\mathrm{hull}_{\mathrm{supp}\, x - \{a\}} \{x\}$.)

Definition 7 (orbit-finite hulls). Let X be a nominal set. Given finite subsets $A \subseteq_{\mathrm{f}} \mathbb{A}$ and $F \subseteq_{\mathrm{f}} X$, define $\mathrm{hull}_A F \triangleq \{\pi \cdot x \mid \pi \# A \wedge x \in F\}$, where $\pi \# A$ means that π is a finite permutation of $\mathbb{A}$ that fixes each $a \in A$.

It is not hard to see that $\mathrm{hull}_A F$ is supported by A and contained in a finite union of orbits of X (namely the orbits of each $x \in F$). What is less obvious is that every orbit-finite subset is of this form. This follows from a key technical property of hulls, proved independently by Turner [39, Lemma 3.4.3.5] and Bojańczyk *et al.* [6, Lemma 3]:

$$(\forall A, A' \subseteq_{\mathrm{f}} \mathbb{A}, F \subseteq_{\mathrm{f}} X)\ A \subseteq A' \Rightarrow$$
$$(\exists F' \subseteq_{\mathrm{f}} X)\ \mathrm{hull}_A F = \mathrm{hull}_{A'} F'. \quad (4)$$

This property can be used to prove the following theorem that makes the connection between orbit-finite subsets and the notion of uniform-compactness from the previous section. Consider the nominal poset $\mathrm{P_{fs}}X$ of finitely supported subsets of a nominal set X, the partial order being subset inclusion. It possesses joins for all

---

[2] Bojańczyk et al. [6, Section 3] use the term 'finitary subset' for what we call an orbit-finite subset.

finitely supported subsets, given by union, and hence in particular it is a udcpo.

Theorem 8. *An element of the udcpo $\mathrm{P_{fs}}X$ is uniform-compact if and only if it is an orbit-finite subset of X; and it is an orbit-finite subset of X if and only if it is equal to $\mathrm{hull}_A F$ for some $A \subseteq_{\mathrm{f}} \mathbb{A}$ and $F \subseteq_{\mathrm{f}} X$. Every $S \in \mathrm{P_{fs}}X$ is the uniform-directed join of the orbit-finite subsets contained in and with the same support as S. Thus $\mathrm{P_{fs}}X$ is an algebraic udcpo in the sense of Definition 5.* $\square$

So there is the following analogy

$$\frac{\text{finite}}{\text{directed}}\text{sets} \sim \frac{\text{orbit-finite}}{\text{uniform-directed}}\text{nominal sets}$$

which we apply next to the classical notion of Scott domain that arose in the denotational semantics of higher-order functional programming languages [30, Lemma 4.4].

5. Nominal Scott domains

A *nominal Scott domain D* is by definition an ω-algebraic udcpo with a least element and joins for all finitely supported subsets that have upper bounds (or equivalently, by Theorem 8, joins for all orbit-finite subsets that have upper bounds).

Remark 9 (flat domains). If X is a nominal set, the *flat nominal poset $X_\perp$* is given by $X \uplus \{\perp\}$, with partial order $d \sqsubseteq d' \Leftrightarrow d = \perp \vee d = d'$ and permutation action extending that on X by $\pi \cdot \perp = \perp$. It is easily seen to be a nominal Scott domain, with $\mathrm{K}(X_\perp) = X_\perp$.

Definition 10. The category **Nsd** has nominal Scott domains for its objects and for its morphisms the functions $f : D \rightarrow D'$ that are both *equivariant*, that is, $f(\pi \cdot d) = \pi \cdot (f\,d)$ holds for all finite permutations π and all $d \in D$, and *uniform-continuous*, that is, monotone and preserving uniform-directed joins.

Remark 11 (Winskel-Turner domain theory). The 'nominal domain theory for concurrency' of Winskel and Turner [40] introduces the notion of uniform-directed join and contains a characterisation of compact elements in terms of the hull construct from Definition 7. However, their domains are more specific than ours as they are based on path sets (downwards-closed subsets of preorders), which form prime-algebraic complete lattices. Modulo countability, their category $\mathbf{FMCts}_\emptyset$ is a full subcategory of **Nsd**.

Theorem 12. **Nsd** *is cartesian closed.*

Proof. The terminal object is given by the trivial flat domain $\emptyset_\perp$. The product of D_1 and D_2 is given by their cartesian product, with permutation action as in (1) and partial order $(d_1, d_2) \sqsubseteq (d'_1, d'_2) \triangleq d_1 \sqsubseteq d'_1 \wedge d_2 \sqsubseteq d'_2$. Exponentials $D_1 \rightarrow D_2$ have an underlying set consisting of all uniform-continuous functions $f : D_1 \rightarrow D_2$ that are finitely supported with respect to the usual permutation action for functions given in (2). The partial order on such functions is also given as usual, argument-wise: $f \sqsubseteq f' \triangleq (\forall d \in D_1)\ f\,d \sqsubseteq f'd$. For ordinary Scott domains, compact elements of the exponential are given by joins of finite, bounded sets of step-functions. Here, given uniform-compact elements $u_i \in \mathrm{K}D_i$ ($i = 1, 2$), one can show that the *step-function*

$$[u_1 , u_2] \triangleq \lambda d \in D_1 \Rightarrow \text{if } u_1 \sqsubseteq d \text{ then } u_2 \text{ else } \perp \quad (5)$$

is in $\mathrm{K}(D_1 \rightarrow D_2)$; that a typical element of $\mathrm{K}(D_1 \rightarrow D_2)$ is the join of an orbit-finite, bounded set of such step functions (so in view of Theorem 8, $\mathrm{K}(D_1 \rightarrow D_2)$ is countable, because $\mathrm{K}D_1$ and $\mathrm{K}D_2$ are); and that every element of $D_1 \rightarrow D_2$ is a uniform-directed join of elements in $\mathrm{K}(D_1 \rightarrow D_2)$. $\square$

We give some examples of uniform-compact elements of exponentials in **Nsd** associated with the flat domain of atomic names,

$\mathbb{A}_\perp$. The examples show that although $\mathbb{A}_\perp$ has a countably infinite underlying set, it has very different uniform-compactness properties from the flat domain of natural numbers, $\mathbb{N}_\perp$ (e.g. the permutation action on $\mathbb{N}$ is *discrete*: $\pi \cdot n \triangleq n$). These examples will be important for the development in Section 7.

Example 13 (name equality test). Let $2 = \{\text{true}, \text{false}\}$ be a two-element, discrete nominal set. For each atomic name $a \in \mathbb{A}$, consider the function $eq_a : \mathbb{A}_\perp \rightarrow 2_\perp$ given by

$$eq_a\, d \triangleq \begin{cases} \text{true} & \text{if } d = a \\ \text{false} & \text{if } d \in \mathbb{A} - \{a\} \\ \perp & \text{if } d = \perp \end{cases} \quad (6)$$

for each $d \in \mathbb{A}_\perp$. Then using the notation from Definition 7 and (5), one finds that $eq_a = \bigsqcup \mathrm{hull}_{\{a\}}\{[a , \text{true}], [a' , \text{false}]\}$, where a' is any atomic name not equal to a. Thus from the proof of Theorem 12 we have $eq_a \in \mathrm{K}(\mathbb{A}_\perp \rightarrow 2_\perp)$.

Example 14 (exists name). For each $f \in \mathbb{A}_\perp \rightarrow 2_\perp$ define

$$exists_\mathbb{A} f \triangleq \begin{cases} \text{true} & \text{if } (\exists a \in \mathbb{A})\ f\,a = \text{true} \\ \text{false} & \text{if } (\forall a \in \mathbb{A})\ f\,a = \text{false} \\ \perp & \text{otherwise.} \end{cases} \quad (7)$$

Picking any $a \in \mathbb{A}$, one can show that $exists_\mathbb{A}$ is equal to

$$\bigsqcup \mathrm{hull}_\emptyset \{[[a , \text{true}] , \text{true}], [\bigsqcup \mathrm{hull}_\emptyset \{[a , \text{false}]\} , \text{false}]\}$$

and hence that $exists_\mathbb{A} \in \mathrm{K}((\mathbb{A}_\perp \rightarrow 2_\perp) \rightarrow 2_\perp)$.

Example 15 (definite name description). Note that the functions in Example 13 satisfy $eq_a = eq_{a'} \Rightarrow a = a'$. Hence for each $f \in (\mathbb{A}_\perp \rightarrow 2_\perp)$ we can define

$$the_\mathbb{A}\, f \triangleq \begin{cases} a & \text{if } f = eq_a \text{ for some } a \in \mathbb{A} \\ \perp & \text{otherwise.} \end{cases} \quad (8)$$

Then picking any $a \in \mathbb{A}$, one has $the_\mathbb{A} = \bigsqcup \mathrm{hull}_\emptyset \{[eq_a , a]\}$ and hence $the_\mathbb{A} \in \mathrm{K}((\mathbb{A}_\perp \rightarrow 2_\perp) \rightarrow \mathbb{A}_\perp)$.

Remark 16 (least fixed points). As for any cartesian closed category, Theorem 12 allows us to model typed λ-calculus using nominal Scott domains and equivariant, uniform-continuous functions. **Nsd** also supports the usual interpretation of recursively defined terms via least fixed points: for if $D \in \mathbf{Nsd}$ and $f \in (D \rightarrow D)$, then $\{\perp, f \perp, f^2 \perp, \ldots\}$ is a uniform-directed subset of D (each element is supported by $\mathrm{supp}\, f$) whose join is $\mathrm{fix}\, f$, the least fixed point of f, by the usual Tarskian argument. Indeed for each nominal Scott domain D, the function assigning least fixed points gives us a morphism in **Nsd**

$$\mathrm{fix} : (D \rightarrow D) \rightarrow D. \quad (9)$$

It is probably the case that **Nsd** has the 'algebraic compactness' properties (enriched over the category of nominal sets) needed to model recursive definitions at the level of types; *cf.* [5, Section 5] and [34, Section 3]. However, we have yet to check the details of this.

6. Abstraction, concretion and restriction

The following result is the basis for giving denotational semantics using nominal Scott domains to languages with name binding operations, such as the one considered in Section 7.

Theorem 17. *If D is a nominal Scott domain, then so is the nominal poset $[\mathbb{A}]D$ from Definition 1. The operation of name abstraction $(a, d) \mapsto \langle a \rangle d$ extends to a morphism $\mathbb{A}_\perp \times D \rightarrow [\mathbb{A}]D$ in **Nsd** once we define $\langle \perp \rangle d \triangleq \perp$.*

Proof. If $S \in \mathrm{P_{fs}}([\mathbb{A}]D)$ is uniform-directed, then so is $\{d \in D \mid \langle a \rangle d \in S\} \in \mathrm{P_{fs}}D$, for any $a \in \mathbb{A}$. The same holds if S is finitely supported and bounded from above. In both cases, picking $a \# S$, one finds that the join of S in $[\mathbb{A}]D$ is $\langle a \rangle (\bigsqcup \{d \in D \mid \langle a \rangle d \in S\})$. Thus, $[\mathbb{A}]D$ has uniform-directed joins and joins of bounded finitely supported subsets, and its least element is $\langle a \rangle \bot$ (for any $a \in \mathbb{A}$). The above description of uniform-directed joins in $[\mathbb{A}]D$ implies that $\langle a \rangle u \in \mathrm{K}(\langle \mathbb{A} \rangle D)$ if and only if $u \in \mathrm{K}D$ and (hence) that $[\mathbb{A}]D$ is ω-algebraic. It also implies that each $\lambda d \in D \to \langle a \rangle d$ is uniform-continuous. $\square$

Algorithms that manipulate binders not only construct name abstractions, they also pull them apart. For example, FreshML [35] supports computation with name abstractions via a convenient form of pattern matching that allows bound entities to be named while guaranteeing invariance under α-equivalence. The mechanism underlying this form of deconstruction of name abstractions is most easily understood in terms of name 'concretion', to which FreshML's pattern matching can be translated. Given $D \in \mathbf{Nsd}$ and $e \in [\mathbb{A}]D$, for each $a \in \mathbb{A}$ with $a \# e$ there is a unique element $e @ a \in D$ satisfying $e = \langle a \rangle (e @ a)$, called the *concretion* of the name abstraction e at the atomic name a [14, Section 5]. Note that this operation is partially defined: to form $e @ a$ we require $a \# e$, meaning that a not in the support of e. For flat domains we will make concretion a total, uniform-continuous function simply by mapping the pairs where concretion is undefined to $\bot \in D$. However, for non-flat domains this is not possible, because in general it does not give a monotone function. For example in $[\mathbb{A}](\mathrm{P_{fs}}\mathbb{A})$, $a' \in \mathrm{supp}\langle a \rangle \{a, a'\}$ (assuming $a \neq a'$), but we cannot define the concretion of $\langle a \rangle \{a, a'\}$ at a' to be the least element $\emptyset$ of $\mathrm{P_{fs}}\mathbb{A}$ since $\langle a \rangle \{a\} \sqsubseteq \langle a \rangle \{a, a'\}$ and $(\langle a \rangle \{a\}) @ a' = \{a'\} \neq \emptyset$.

One way to deal with this partiality of concretion in a programming language is to enhance its type system with 'freshness assumptions' to ensure statically that name abstractions are only concreted at fresh names. This is the solution adopted by the original version of FreshML [28] and is the one chosen by Winskel and Turner in their language Nominal HOPLA [39, 40]. Later versions of FreshML use a conventional type system and enforce freshness conditions dynamically via the use of local names in expressions [35]—at the expense of purity [31]. We do the same with the language introduced in the next section, but achieve purity via the use of Odersky-style [24] local names rather than generative ones. These will be modelled by some extra structure on nominal Scott domains in the form of name restriction operations [27, Section 2.3] that enable us to give morphisms $[\mathbb{A}]D \times \mathbb{A}_\bot \to D$ in $\mathbf{Nsd}$ for the operation of name concretion.

Definition 18 (uniform-continuous name restriction). A *uniform-continuous name restriction operation* on a nominal Scott domain D is a morphism $r : [\mathbb{A}]D \to D$ in $\mathbf{Nsd}$ satisfying the structural properties: $a \# d \Rightarrow r\langle a \rangle d = d$ and $r\langle a \rangle (r\langle a' \rangle d) = r\langle a' \rangle (r\langle a \rangle d)$ for all $a, a' \in \mathbb{A}$ and $d \in D$.

We usually write $r\langle a \rangle d$ as $a \backslash d$ with the particular morphism r understood from context. Using this morphism, as in Pitts [27, Corollary 2.14] we can extend the partial operation of concretion to a total equivariant function $\_ @ \_ : [\mathbb{A}]D \times \mathbb{A}_\bot \to D$ satisfying

$$\left. \begin{array}{rcl} (\langle a \rangle d) @ a & = & d \\ (\langle a \rangle d) @ a' & = & a \backslash (a\ a') \cdot d \quad \text{if } a \neq a' \\ (\langle a \rangle d) @ \bot & = & \bot. \end{array} \right\} \quad (10)$$

The fact that this is uniform-continuous and hence determines a morphism in $\mathbf{Nsd}$ follows from the description of uniform-directed joins in $[\mathbb{A}]D$ in the proof of Theorem 17.

The following result shows that domains arising in the denotational semantics of higher-order functional programming with name abstractions (Section 7) all carry a uniform-continuous name restriction operation. The theorem can be proved by extending the results in Pitts [27, Section 2.3] from nominal sets to nominal Scott domains.

Theorem 19. *Every flat nominal Scott domain $X_\bot$ has a uniform-continuous name restriction operation satisfying*

$$a \backslash d = \begin{cases} d & \text{if } a \# d \\ \bot & \text{if } a \in \mathrm{supp}\, d \end{cases} \quad (11)$$

for all $a \in \mathbb{A}$ and $d \in X_\bot$. If $D_1, D_2 \in \mathbf{Nsd}$ have uniform-continuous name restriction operations, their product $D_1 \times D_2$ has one satisfying

$$a \backslash (d_1, d_2) = (a \backslash d_1, a \backslash d_2) \quad (12)$$

for all $a \in \mathbb{A}$, $d_1 \in D_1$ and $d_2 \in D_2$. If $D_1, D_2 \in \mathbf{Nsd}$ and D_2 has a uniform-continuous name restriction operation, then whether or not D_1 has one as well, the exponential $D_1 \to D_2$ has such an operation, satisfying

$$(a \backslash f)\, d = a \backslash (f\, d) \quad \text{if } a \# d \quad (13)$$

for all $d \in D_1$ and $f \in D_1 \to D_2$. Finally, if $D \in \mathbf{Nsd}$ has a uniform-continuous name restriction operation, then the name abstraction domain $[\mathbb{A}]D$ has one satisfying

$$a \backslash (\langle a' \rangle d) = \langle a' \rangle (a \backslash d) \quad \text{if } a \neq a' \quad (14)$$

for all $a, a' \in \mathbb{A}$ and $d \in D$. $\square$

Remark 20. In the above theorem, the name restriction operation for exponentials is rather subtle. Property (13) at first seems like only a partial specification for the function $a \backslash f$, but in fact determines it uniquely, since it implies that for all $d \in D$

$$(a \backslash f)\, d = a' \backslash ((a\ a') \cdot f)\, d \quad \text{for any } a' \# (f, d) \quad (15)$$

[see 27, Theorem 2.10]. It is easier to see that (14) uniquely defines name restriction for name abstraction domains because given $a \in \mathbb{A}$, we can always choose a representative for the equivalence class $\langle a' \rangle d$ with $a \neq a'$.

Theorem 19 gives operations that adequately model locally scoped names in programming languages. For example, in retrospect one can see that Shinwell and Pitts [34] use the name restriction operation constructed as above for continuation domains of the form $(D \to 1_\bot) \to 1_\bot$ to adequately model the operational semantics of FreshML, which evaluates a local scope by generating a name that is fresh for the current state. In this paper we use Odersky's functional theory of local names [24], which is modelled in $\mathbf{Nsd}$ rather easily in view of the above theorem. The next section introduces a language corresponding to a simply typed fragment of FreshML [35], but with this kind of locally scoped name.

7. PNA: programming with name abstractions

The programming language PNA (Programming with Name Abstractions) is basically Plotkin's PCF [30] with names added. Like PCF, PNA has arithmetic constructs, call-by-name higher-order functions and fixed-point recursion. What distinguishes the two languages is that PNA treats names as first-class citizens and has constructs for locally scoping them, swapping them, testing them for equality, and for name abstraction and concretion. To exercise the use of name abstraction it also features a representative 'nominal algebraic datatype', namely a type for α-equivalence classes of λ-terms. For example, when *subst* is the PNA expression defined below, *subst* $e_1\, a\, e_2$ computes the λ-term obtained by capture-avoiding substitution of the λ-term represented by e_1 for all free occurrences of the variable named a in the λ-term represented by e_2.

$$\tau \in \mathrm{Typ} ::= \qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{types}$$
$$\mathtt{bool} \mid \mathtt{nat} \mid \tau \times \tau \mid \tau \to \tau \mid \mathtt{name} \mid \mathtt{term} \mid \delta\,\tau$$

$e \in \mathrm{Exp} ::=$	*expressions*
x	variable $(x \in \mathbb{V})$
$\mathtt{T}$	truth
$\mathtt{F}$	falsity
$\mathtt{if}\,e\,\mathtt{then}\,e\,\mathtt{else}\,e$	conditional
$\mathtt{O}$	number zero
$\mathtt{S}\,e$	successor
$\mathtt{pred}\,e$	predecessor
$\mathtt{zero}\,e$	zero test
$(e\,,e)$	pair
$\mathtt{fst}\,e$	first projection
$\mathtt{snd}\,e$	second projection
$\lambda x : \tau \to e$	function abstraction
$e\,e$	function application
$\mathtt{fix}\,e$	fixed-point recursion
a	atomic name $(a \in \mathbb{A})$
$\nu a.\,e$	local name
$(e \rightleftharpoons e)\,e$	name swapping
$e = e$	name equality test
$\mathtt{V}\,e$	variable term
$\mathtt{A}\,e\,e$	application term
$\mathtt{L}\,e$	lambda term
$\mathtt{case}\,e\,\mathtt{of}\,(\mathtt{V}\,x \to e \mid \mathtt{A}\,x\,x \to e \mid \mathtt{L}\,x \to e)$	term case
$\alpha a.\,e$	name abstraction
$e\,@\,e$	name concretion

$$c \in \mathrm{Can} ::= \qquad\qquad\qquad\qquad\qquad \textit{canonical forms}$$
$$\mathtt{T} \mid \mathtt{F} \mid \mathtt{O} \mid \mathtt{S}\,c \mid (e\,,e) \mid \lambda x : \tau \to e \mid a \mid \mathtt{V}\,c \mid \mathtt{A}\,c\,c \mid \mathtt{L}\,c \mid \alpha a.\,c$$

Figure 1. Syntax of PNA

$$subst \triangleq \lambda y : \mathtt{term} \to \lambda x : \mathtt{name} \to$$
$$\mathtt{fix}\,(\lambda(f : \mathtt{term} \to \mathtt{term}) \to \lambda y' : \mathtt{term} \to$$
$$\mathtt{case}\,y'\,\mathtt{of}$$
$$\mathtt{V}\,x_1 \to \mathtt{if}\,x_1 = x\,\mathtt{then}\,y\,\mathtt{else}\,y'$$
$$\mid \mathtt{A}\,y_2\,y_2' \to \mathtt{A}\,(f\,y_2)\,(f\,y_2')$$
$$\mid \mathtt{L}\,z \to \mathtt{L}\,(\alpha a.\,f(z\,@\,a))).$$

Figure 1 gives the syntax of PNA. In the grammar for expressions, the part below the dotted line is what is added to PCF. There are two kinds of identifier in the language: *variables* $x, y, z, f, \ldots \in \mathbb{V}$ and *atomic names* $a, b, c, \ldots \in \mathbb{A}$. The sets $\mathbb{V}$ of variables and $\mathbb{A}$ of atomic names are disjoint and countably infinite. Both kinds of identifier may be bound and the language's binding forms are $\lambda x : \tau \to \_$, $\nu a.\_$, $\mathtt{case}\,e\,\mathtt{of}\,(\mathtt{V}\,x \to \_ \mid \mathtt{A}\,x\,x \to \_ \mid \mathtt{L}\,x \to \_)$ and $\alpha a.\_$. We identify expressions up to α-equivalence of bound identifiers. For any expression e, we write $\mathrm{fn}(e)$ for its finite set of free atomic names and $\mathrm{fv}(e)$ for its finite set of free variables.

The reason for making a syntactic distinction between variables and atomic names is that they behave differently. Various properties of PNA, such as its typing judgement, are preserved by the operation of substitution of expressions for variables but are only preserved by permutations of atomic names rather than more general forms of substitution for names. The operation of simultaneous substitution of expressions $e_1, .., e_n$ for distinct variables $x_1, .., x_n$ in an expression e is written as $e[e_1/x_1, .., e_n/x_n]$, where the substitution avoids capture of both free variables and free atomic names by the language's binding forms. The operation of applying a finite permutation $\pi : \mathbb{A} \cong \mathbb{A}$ to an expression e is written $\pi \cdot e$. It

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad\qquad \frac{c = \mathtt{T} \mid \mathtt{F}}{\Gamma \vdash c : \mathtt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathtt{if}\,e_1\,\mathtt{then}\,e_2\,\mathtt{else}\,e_3 : \tau} \qquad \frac{}{\Gamma \vdash \mathtt{O} : \mathtt{nat}}$$

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{S}\,e : \mathtt{nat}} \qquad \frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{pred}\,e : \mathtt{nat}} \qquad \frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash \mathtt{zero}\,e : \mathtt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1\,,e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{fst}\,e : \tau_1}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{snd}\,e : \tau_2} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau \to e : \tau \to \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\,e_2 : \tau'} \qquad \frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \mathtt{fix}\,e : \tau}$$

$$\frac{a \in \mathbb{A}}{\Gamma \vdash a : \mathtt{name}} \qquad \frac{a \in \mathbb{A} \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \nu a.\,e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{name} \qquad \Gamma \vdash e_2 : \mathtt{name}}{\Gamma \vdash e_3 : \tau} \qquad \frac{\Gamma \vdash e_1 : \mathtt{name}}{\Gamma \vdash e_2 : \mathtt{name}}$$
$$\frac{}{\Gamma \vdash (e_1 \rightleftharpoons e_2)\,e_3 : \tau} \qquad\qquad \frac{}{\Gamma \vdash e_1 = e_2 : \mathtt{bool}}$$

$$\frac{\Gamma \vdash e : \mathtt{name}}{\Gamma \vdash \mathtt{V}\,e : \mathtt{term}} \qquad \frac{\Gamma \vdash e_1 : \mathtt{term} \qquad \Gamma \vdash e_2 : \mathtt{term}}{\Gamma \vdash \mathtt{A}\,e_1\,e_2 : \mathtt{term}} \qquad \frac{\Gamma \vdash e : \delta\,\mathtt{term}}{\Gamma \vdash \mathtt{L}\,e : \mathtt{term}}$$

$$\frac{\Gamma \vdash e : \mathtt{term} \qquad \Gamma, x_1 : \mathtt{name} \vdash e_1 : \tau}{\Gamma, x_2 : \mathtt{term}, x_2' : \mathtt{term} \vdash e_2 : \tau \qquad \Gamma, x_3 : \delta\,\mathtt{term} \vdash e_3 : \tau}{\Gamma \vdash \mathtt{case}\,e\,\mathtt{of}\,(\mathtt{V}\,x_1 \to e_1 \mid \mathtt{A}\,x_2\,x_2' \to e_2 \mid \mathtt{L}\,x_3 \to e_3) : \tau}$$

$$\frac{a \in \mathbb{A} \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \alpha a.\,e : \delta\,\tau} \qquad \frac{\Gamma \vdash e_1 : \delta\,\tau \qquad \Gamma \vdash e_2 : \mathtt{name}}{\Gamma \vdash e_1\,@\,e_2 : \tau}$$

Figure 2. PNA typing rules

is defined by recursing into all sub-expressions and applying π to occurrences of atomic names. This is an action in the sense of Section 2 and makes the set Exp of PNA expressions into a nominal set. Since the elements of Exp are expressions up to α-equivalence, support in this nominal set is given by the finite set of free names of each expression.

PNA is a simply typed language. The grammar for types (Figure 1) extends that for PCF (in a version with products $\tau_1 \times \tau_2$) with a type $\mathtt{name}$ of names, a type $\mathtt{term}$ of λ-terms modulo α-equivalence, and name abstraction types $\delta\,\tau$. The inductively defined typing judgement $\Gamma \vdash e : \tau$ (read as 'in the environment Γ the expression e has type τ') is defined in Figure 2 by the usual rules for PCF and, below the dotted line, rules concerning names. The typing environments $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ are finite functions from variables to types that track occurrences of free variables in e. Note that because there is only one sort of name, we do not bother to add a component to Γ tracking occurrences of free atomic names in e.

In Figure 3 we extend PCF's usual rules for an inductively defined big-step evaluation relation with the rules below the dotted

$$\frac{c = \mathtt{T} \mid \mathtt{F} \mid \mathtt{0} \mid (e_1 , e_2) \mid \lambda x : \tau \to e}{c \Downarrow c} \qquad \frac{e \Downarrow c}{\mathtt{S}\,e \Downarrow \mathtt{S}\,c}$$

$$\frac{e_1 \Downarrow \mathtt{T} \quad e_2 \Downarrow c}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \Downarrow c} \qquad \frac{e_1 \Downarrow \mathtt{F} \quad e_3 \Downarrow c}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \Downarrow c}$$

$$\frac{e \Downarrow \mathtt{S}\,c}{\mathtt{pred}\,e \Downarrow c} \qquad \frac{e \Downarrow \mathtt{0}}{\mathtt{zero}\,e \Downarrow \mathtt{T}} \qquad \frac{e \Downarrow \mathtt{S}\,c}{\mathtt{zero}\,e \Downarrow \mathtt{F}}$$

$$\frac{e \Downarrow (e_1 , e_2) \quad e_1 \Downarrow c}{\mathtt{fst}\,e \Downarrow c} \qquad \frac{e \Downarrow (e_1 , e_2) \quad e_2 \Downarrow c}{\mathtt{snd}\,e \Downarrow c}$$

$$\frac{e_1 \Downarrow \lambda x : \tau \to e \quad e[e_2/x] \Downarrow c}{e_1\,e_2 \Downarrow c} \qquad \frac{e(\mathtt{fix}\,e) \Downarrow c}{\mathtt{fix}\,e \Downarrow c}$$

. . . .

$$\frac{a \in \mathbb{A}}{a \Downarrow a} \qquad \frac{e \Downarrow c \quad a\backslash c := c'}{\nu a.\,e \Downarrow c'} \qquad \frac{e_1 \Downarrow a_1 \quad e_2 \Downarrow a_2 \quad e_3 \Downarrow c}{(e_1 \rightleftharpoons e_2)\,e_3 \Downarrow (a_1\,a_2)\cdot c}$$

$$\frac{e_1 \Downarrow a \quad e_2 \Downarrow a}{e_1 = e_2 \Downarrow \mathtt{T}} \qquad \frac{e_1 \Downarrow a \quad e_2 \Downarrow a' \quad a \neq a'}{e_1 = e_2 \Downarrow \mathtt{F}}$$

$$\frac{e \Downarrow c}{\mathtt{V}\,e \Downarrow \mathtt{V}\,c} \qquad \frac{e_1 \Downarrow c_1 \quad e_2 \Downarrow c_2}{\mathtt{A}\,e_1\,e_2 \Downarrow \mathtt{A}\,c_1\,c_2} \qquad \frac{e \Downarrow c}{\mathtt{L}\,e \Downarrow \mathtt{L}\,c}$$

$$\frac{e \Downarrow \mathtt{V}\,c \quad e_1[c/x_1] \Downarrow c'}{\mathtt{case}\,e\,\mathtt{of}\,(\mathtt{V}\,x_1 \to e_1 \mid \cdots) \Downarrow c'}$$

$$\frac{e \Downarrow \mathtt{A}\,c\,c' \quad e_2[c/x_2, c'/x_2'] \Downarrow c''}{\mathtt{case}\,e\,\mathtt{of}\,(\cdots \mid \mathtt{A}\,x_2\,x_2' \to e_2 \mid \cdots) \Downarrow c''}$$

$$\frac{e \Downarrow \mathtt{L}\,c \quad e_3[c/x_3] \Downarrow c'}{\mathtt{case}\,e\,\mathtt{of}\,(\cdots \mid \mathtt{L}\,x_3 \to e_3) \Downarrow c'} \qquad \frac{e \Downarrow c}{\alpha a.\,e \Downarrow \alpha a.\,c}$$

$$\frac{e_1 \Downarrow \alpha a.\,c \quad e_2 \Downarrow a' \quad a \neq a' \quad \nu a.\,(a\,a')\cdot c \Downarrow c'}{e_1\,\mathtt{@}\,e_2 \Downarrow c'}$$

Figure 3. PNA evaluation rules

line that concern names. The evaluation rule for local names makes use of the auxiliary definition in Figure 4. Thus the relation $e \Downarrow c$ defines when a PNA expression e evaluates to *canonical form c*. See Figure 1 for the grammar of canonical forms. As for PCF, we only evaluate expressions that are *variable-closed* in the sense that $\mathrm{fv}(e) = \emptyset$, though they may contain free atomic names for evaluation. This is because, unlike variables, atomic names are canonical forms. We also choose to evaluate under name abstractions, so that $\alpha a.\,e$ is in canonical form if and only if e is. This permits a representation of α-equivalence classes of λ-terms in PNA that is as simple as PCF's representation of numbers: they are in bijection with variable-closed canonical forms of type term. (It is certainly possible to give a different operational semantics in which one does not evaluate under name abstractions. The corresponding denotational semantics would make more use of lifting than does the one in Section 8.) To deconstruct name abstractions, PNA features an operational version of the total concretion operation discussed in Section 6. Its behaviour is given by the last rule in Figure 3 and corresponds to property (10).

$$\frac{c = \mathtt{T} \mid \mathtt{F} \mid \mathtt{0} \mid \mathtt{S}\,c'}{a\backslash c := c} \qquad \overline{a\backslash(e_1 , e_2) := (\nu a.\,e_1 , \nu a.\,e_2)}$$

$$\overline{a\backslash\lambda x : \tau \to e := \lambda x : \tau \to \nu a.\,e} \qquad \frac{a \neq a'}{a\backslash a' := a'}$$

$$\frac{a\backslash c := c'}{a\backslash\mathtt{V}\,c := \mathtt{V}\,c'} \qquad \frac{a\backslash c_1 := c_1' \quad a\backslash c_2 := c_2'}{a\backslash\mathtt{A}\,c_1\,c_2 := \mathtt{A}\,c_1'\,c_2'}$$

$$\frac{a\backslash c := c'}{a\backslash\mathtt{L}\,c := \mathtt{L}\,c'} \qquad \frac{a\backslash c := c' \quad a \neq a'}{a\backslash\alpha a'.\,c := \alpha a'.\,c'}$$

Figure 4. Partial operation of name restriction, $a\backslash c := c'$

As for PCF, the PNA evaluation relation is easily seen to be deterministic ($e \Downarrow c \wedge e \Downarrow c' \Rightarrow c = c'$) and type-sound ($\emptyset \vdash e : \tau \wedge e \Downarrow c \Rightarrow \emptyset \vdash c : \tau$). It is also equivariant:

$$e \Downarrow c \Rightarrow \pi \cdot e \Downarrow \pi \cdot c. \tag{16}$$

Remark 21 (Odersky-style local names). Evaluation of locally scoped names $\nu a.\,e$ makes use of Odersky's functional theory of local names [24], because that way of evaluating them fits the intended model of PNA using nominal Scott domains. Scopes intrude in a type-directed fashion, as can be seen in the partial operation of name restriction on canonical forms $a\backslash c := c'$ defined in Figure 4. This operation is partial because $a\backslash a := c$ holds for no c. Thus, unlike in Pitts [27], we choose to follow Odersky [24] and make $\nu a.\,a$ a stuck expression that does not evaluate to any canonical form and whose denotation is $\bot$. This has the advantage that there are no exotic values (value-closed canonical forms) of type term—the only values of that type correspond to α-equivalence classes of λ-terms. The use of Odersky-style local names means that the operational semantics of PNA is stateless, unlike the operational semantics of the more usual generative version of $\nu a.\,\_$ used in the ν-calculus [29]. At the same time, it is known to be as expressive as that version in as much as there is an adequate translation from generative into Odersky-style local names [19].

Definition 22 (contextual equivalence). As usual, a PNA *context* $C[\_]$ is an expression with a single sub-expression replaced by the place-holder '$\_$', and $C[e]$ is the expression that results from replacing $\_$ by an expression e (possibly capturing free variables and atomic names in e). Given well-typed expressions $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$, we write $\Gamma \vdash e \cong_{\mathsf{PNA}} e' : \tau$ and say that e and e' are *contextually equivalent* if for all contexts $C[\_]$ for which $\emptyset \vdash C[e] : \mathtt{bool}$ and $\emptyset \vdash C[e'] : \mathtt{bool}$ hold, it is the case that $C[e] \Downarrow \mathtt{T}$ if and only if $C[e'] \Downarrow \mathtt{T}$.

Example 23. Although PNA contains the expressions of the ν-calculus [29] as a subset, the two languages have different semantics for local names—Odersky-style for PNA versus generative for the ν-calculus. This affects properties of contextual equivalence in the two languages. For example, if $\Gamma, x : \tau \vdash e : \tau'$, then

$$\Gamma \vdash \nu a.\,\lambda x : \tau \to e \cong_{\mathsf{PNA}} \lambda x : \tau \to \nu a.\,e : \tau \to \tau' \tag{17}$$

is valid. However, this is not always the case in the ν-calculus [29, Example 2]. One can prove (17) by checking that this identity holds in the denotational model developed in the next section and then appealing to the computational adequacy result proved there (Theorem 27). On the other hand, analogues of some ν-calculus equivalences are also true for PNA, once one takes into account the fact that, like PCF, PNA is call-by-name but the ν-calculus

is call-by-value. For example, here are call-by-name analogues of equivalences in Pitts and Stark [29, Example 4]

$$\emptyset \vdash \nu a.\, \lambda x : \texttt{name} \to (x = a) \cong_{\mathsf{PNA}}$$
$$\lambda x : \texttt{name} \to \texttt{if } x = x \texttt{ then F else F} : \texttt{name} \to \texttt{bool} \qquad (18)$$

$$\emptyset \vdash \nu a.\, \nu a'.\, \lambda(f : \texttt{name} \to \texttt{bool}) \to eq\,(f\,a)\,(f\,a') \cong_{\mathsf{PNA}}$$
$$\lambda(f : \texttt{name} \to \texttt{bool}) \to \nu a.\, \texttt{if } f\,a \texttt{ then T else T} \qquad (19)$$
$$: (\texttt{name} \to \texttt{bool}) \to \texttt{bool}.$$

Here $eq : \texttt{bool} \to \texttt{bool} \to \texttt{bool}$ is an abbreviation for a boolean-equality test defined using conditionals. In contrast to the ν-calculus, where it takes some effort to prove equivalences like (18) and (19), for PNA these properties are easily seen to hold in the straightforward and computationally adequate denotational semantics that we describe next.

8. Denotational semantics of PNA

For each PNA type τ, we define a nominal Scott domain $[\![\tau]\!]$ by recursion on the structure of τ as follows.

- $[\![\texttt{bool}]\!] = 2_\perp$, the flat domain (*cf.* Remark 9) on a discrete nominal set with two elements, $2 = \{\text{true}, \text{false}\}$.
- $[\![\texttt{nat}]\!] = \mathbb{N}_\perp$, the flat domain on the discrete nominal set of natural numbers, $\mathbb{N} = \{0, 1, 2, \ldots\}$.
- $[\![\tau \times \tau']\!] = [\![\tau]\!] \times [\![\tau']\!]$, the product of nominal Scott domains.
- $[\![\tau \to \tau']\!] = [\![\tau]\!] \to [\![\tau']\!]$, the nominal Scott domain of finitely supported, uniform-continuous functions (Theorem 12).
- $[\![\texttt{name}]\!] = \mathbb{A}_\perp$, the flat domain on the nominal set of atomic names, $\mathbb{A} = \{a, b, c, \ldots\}$.
- $[\![\texttt{term}]\!] = (\Lambda_\alpha)_\perp$, the flat domain on the nominal set of α-equivalence classes of λ-terms [14, Theorem 6.2],

$$\Lambda_\alpha \triangleq \{t ::= a \mid \lambda a.t \mid t\,t\}/=_\alpha \quad (\text{where } a \in \mathbb{A}). \qquad (20)$$

- $[\![\delta\,\tau]\!] = [\mathbb{A}][\![\tau]\!]$, the domain of name abstractions of the nominal Scott domain $[\![\tau]\!]$ (Theorem 17).

Typing environments are interpreted as finite cartesian products: $[\![\{x_1 : \tau_1, \ldots, x_n : \tau_n\}]\!] = [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]$. Finite tuples $\rho \in [\![\Gamma]\!]$ can be interpreted as partial functions from variables to domains such that $\text{dom}(\rho) = \text{dom}(\Gamma)$ and $\rho(x) \in [\![\Gamma(x)]\!]$ for all $x \in \text{dom}(\Gamma)$. We call such partial functions Γ-*environments*. If $\rho \in [\![\Gamma]\!]$, $x \notin \text{dom}(\Gamma)$ and $d \in [\![\tau]\!]$, then we write $\rho[x \mapsto d]$ for the $(\Gamma, x : \tau)$-environment that maps x to d and otherwise acts like ρ.

For each well-typed expression $\Gamma \vdash e : \tau$ and Γ-environment $\rho \in [\![\Gamma]\!]$ we define an element $[\![e]\!]\rho \in [\![\tau]\!]$ satisfying the clauses in Figure 5, by recursion over the structure of e.[3] The clauses for syntax constructs from PCF are analogous to the standard denotational semantics of PCF in Scott domains. The functions π_1 and π_2 in the clauses for $\texttt{fst}\,e$ and $\texttt{snd}\,e$ are the first and second projection functions for pairs. fix in the clause for $\texttt{fix}\,e$ is the least fixed point function (9). The clauses below the dotted line in Figure 5 are for the new syntactic constructs of PNA. In the clauses involving expressions of type $\texttt{term}$, we use $[t]_\alpha$ to denote the α-equivalence class of the syntax tree t of a λ-term. We use the concretion function from (10) in the clause for concretion expressions $e_1 @ e_2$. The clause for $\nu a.\, e$ makes use of the uniform-continuous name restriction operation that each $[\![\tau]\!]$ has by virtue of Theorem 19. Note that the side conditions in the clauses for $\nu a.\, e$ and $\alpha a.\, e$ are always satisfiable as we identify expression up to α-equivalence. One can reformulate these clauses without side

---

[3] Strictly speaking, it is by α-structural recursion [26] since we identify expressions up to α-equivalence of bound identifiers.

$$[\![x]\!]\rho = \rho\,x$$

$$[\![\texttt{T}]\!]\rho = \text{true} \qquad [\![\texttt{F}]\!]\rho = \text{false}$$

$$[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]\rho = \begin{cases} [\![e_2]\!]\rho & \text{if } [\![e_1]\!]\rho = \text{true} \\ [\![e_3]\!]\rho & \text{if } [\![e_1]\!]\rho = \text{false} \\ \perp & \text{otherwise} \end{cases}$$

$$[\![\texttt{O}]\!]\rho = 0$$

$$[\![\texttt{S}\,e]\!]\rho = \begin{cases} n+1 & \text{if } [\![e]\!]\rho = n \in \mathbb{N} \\ \perp & \text{otherwise} \end{cases}$$

$$[\![\texttt{pred}\,e]\!]\rho = \begin{cases} n & \text{if } [\![e]\!]\rho = n+1 \in \mathbb{N} \\ \perp & \text{otherwise} \end{cases}$$

$$[\![\texttt{zero}\,e]\!]\rho = \begin{cases} \text{true} & \text{if } [\![e]\!]\rho = 0 \in \mathbb{N} \\ \text{false} & \text{if } [\![e]\!]\rho = n+1 \in \mathbb{N} \\ \perp & \text{otherwise} \end{cases}$$

$$[\![(e_1, e_2)]\!]\rho = ([\![e_1]\!]\rho, [\![e_2]\!]\rho)$$

$$[\![\texttt{fst}\,e]\!]\rho = \pi_1([\![e]\!]\rho) \qquad [\![\texttt{snd}\,e]\!]\rho = \pi_2([\![e]\!]\rho)$$

$$[\![\lambda x : \tau \to e]\!]\rho = \lambda d \in [\![\tau]\!].\, [\![e]\!]\rho[x \mapsto d]$$

$$[\![e_1\,e_2]\!]\rho = [\![e_1]\!]\rho([\![e_2]\!]\rho)$$

$$[\![\texttt{fix}\,e]\!]\rho = \text{fix}([\![e]\!]\rho)$$

$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$

$$[\![a]\!]\rho = a$$

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

$$[\![(e_1 \rightleftharpoons e_2)\,e_3]\!]\rho =$$
$$\begin{cases} (a_1\,a_2) \cdot ([\![e_3]\!]\rho) & \text{if } [\![e_i]\!]\rho = a_i \in \mathbb{A}\ (i=1,2) \\ \perp & \text{otherwise} \end{cases}$$

$$[\![e_1 = e_2]\!]\rho = \begin{cases} eq_a([\![e_2]\!]\rho) & \text{if } [\![e_1]\!]\rho = a \in \mathbb{A} \\ \perp & \text{otherwise} \end{cases}$$

$$[\![\texttt{V}\,e]\!]\rho = \begin{cases} [a]_\alpha & \text{if } [\![e]\!]\rho = a \in \mathbb{A} \\ \perp & \text{otherwise} \end{cases}$$

$$[\![\texttt{A}\,e_1\,e_2]\!]\rho = \begin{cases} [t_1\,t_2]_\alpha & \text{if } [\![e_i]\!]\rho = [t_i]_\alpha \in \Lambda_\alpha\ (i=1,2) \\ \perp & \text{otherwise} \end{cases}$$

$$[\![\texttt{L}\,e]\!]\rho = \begin{cases} [\lambda a.t]_\alpha & \text{if } [\![e]\!]\rho = \langle a\rangle[t]_\alpha \in [\mathbb{A}]\Lambda_\alpha \\ \perp & \text{otherwise} \end{cases}$$

$$[\![\texttt{case } e \texttt{ of } (\texttt{V}\,x_1 \to e_1 \mid \texttt{A}\,x_2\,x_2' \to e_2 \mid \texttt{L}\,x_3 \to e_3)]\!]\rho =$$
$$\begin{cases} [\![e_1]\!]\rho[x_1 \mapsto a] & \text{if } [\![e]\!]\rho = [a]_\alpha \\ [\![e_2]\!]\rho[x_2 \mapsto [t]_\alpha, x_2' \mapsto [t']_\alpha] & \text{if } [\![e]\!]\rho = [t\,t']_\alpha \\ [\![e_3]\!]\rho[x_3 \mapsto \langle a\rangle[t]_\alpha] & \text{if } [\![e]\!]\rho = [\lambda a.t]_\alpha \\ \perp & \text{otherwise} \end{cases}$$

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

$$[\![e_1 @ e_2]\!]\rho = ([\![e_1]\!]\rho) @ ([\![e_2]\!]\rho)$$

Figure 5. PNA denotational semantics

conditions [cf. 27, Figure 4]

$$[\![\nu a.\, e]\!] = a\backslash(\lambda\rho \in [\![\Gamma]\!].\, [\![e]\!]\rho) \tag{21}$$

$$[\![\alpha a.\, e]\!] = a\backslash(\lambda\rho \in [\![\Gamma]\!].\, \langle a\rangle([\![e]\!]\rho)) \tag{22}$$

by appealing to the slightly subtle properties of the name restriction operation for exponential domains (Remark 20).

Notation 24. For the empty typing environment $\emptyset$, there is a unique $\emptyset$-environment, ρ_0. Given a variable-closed expression $\emptyset \vdash e : \tau$, we simply write $[\![e]\!]$ for $[\![e]\!]\rho_0$.

Using the developments from Sections 5 and 6 one can prove the following results.

Lemma 25. *The denotation of any well-typed* PNA *expression* $\Gamma \vdash e : \tau$ *is a well-defined, finitely supported and uniform-continuous function* $[\![e]\!] : [\![\Gamma]\!] \to [\![\tau]\!]$. $\quad\square$

Lemma 26 (PNA soundness). *If* $e \Downarrow c$, *then* $[\![e]\!] = [\![c]\!]$.

Proof. The proof is by rule induction for $\Downarrow$, using the following properties of the denotational semantics whose proofs we omit.

- **Substitution lemma** If $\Gamma \vdash e : \tau$ and $\Gamma, x : \tau \vdash e' : \tau'$, then $\Gamma \vdash e'[e/x] : \tau'$ holds and for all $\rho \in [\![\Gamma]\!]$ we have $[\![e'[e/x]]\!]\rho = [\![e']\!]\rho[x \mapsto [\![e]\!]\rho]$.
- **Equivariance lemma** $\pi \cdot [\![e]\!] = [\![\pi \cdot e]\!]$.
- **Restriction lemma** If $a\backslash c := c'$ with c and c' variable-closed canonical forms of type τ, then the uniform-continuous name restriction operation defined on $[\![\tau]\!]$ as in Theorem 19 satisfies $a\backslash[\![c]\!] = [\![c']\!]$. $\quad\square$

The following result allows one to establish PNA contextual equivalences by proving equality of denotations.

Theorem 27 (PNA **computational adequacy**). *Given* $\Gamma \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau$, *if* $[\![e]\!] = [\![e']\!] \in [\![\Gamma]\!] \to [\![\tau]\!]$, *then* $\Gamma \vdash e \cong_{\mathsf{PNA}} e' : \tau$.

Proof. It is not hard to see that the denotational semantic is compositional, in the sense that $[\![e]\!] = [\![e']\!] \Rightarrow [\![C[e]]\!] = [\![C[e']]\!]$. So in view of Lemma 26 it suffices to show that if e is a variable-closed expression of type bool, then

$$[\![e]\!] = \mathrm{true} \Rightarrow e \Downarrow \mathtt{T}. \tag{23}$$

We prove this by devising a suitable logical relation

$$d \lhd_\tau e \quad (d \in [\![\tau]\!],\ \emptyset \vdash e : \tau) \tag{24}$$

between the semantics and the syntax of PNA. (See Streicher [38, Chapter 4] for a good exposition of this method of proving computational adequacy for the Scott domain model of PCF.) The definition of $\lhd_\tau$ is by recursion on the structure of the type τ:

$$d \lhd_\gamma e \triangleq d = \bot \lor (\exists c)\, e \Downarrow c \land [\![c]\!] = d$$
$$\text{for } \gamma = \mathsf{bool}, \mathsf{nat}, \mathsf{name}, \mathsf{term}$$
$$(d_1, d_2) \lhd_{\tau_1 \times \tau_2} e \triangleq d_1 \lhd_{\tau_1} \mathtt{fst}\, e \land d_2 \lhd_{\tau_2} \mathtt{snd}\, e$$
$$d \lhd_{\tau_1 \to \tau_2} e \triangleq (\forall d_1, e_1)\, d_1 \lhd_{\tau_1} e_1 \Rightarrow d\, d_1 \lhd_{\tau_2} e\, e_1$$
$$d \lhd_{\delta\tau} e \triangleq (\mathsf{N}a)\, d\, @\, a \lhd_\tau e\, @\, a.$$

The definition is standard except for the last clause, which is for name abstraction types. There we use the *freshness quantifier* ($\mathsf{N}a$) of nominal logic [25]. Thus $d \lhd_{\delta\tau} e$ holds if and only if $d\, @\, a \lhd_\tau e\, @\, a$ holds for some $a \# (d, e)$, or equivalently, for any $a \# (d, e)$.

The proof that this logical relation is closed under restriction, abstraction and concretion

$$d \lhd_\tau e \Rightarrow (\forall a)\, a\backslash d \lhd_\tau \nu a.\, e \tag{25}$$
$$d \lhd_\tau e \Rightarrow (\forall a)\, \langle a\rangle d \lhd_{\delta\tau} \alpha a.\, e \tag{26}$$
$$d \lhd_{\delta\tau} e \Rightarrow (\forall a)\, d\, @\, a \lhd_\tau e\, @\, a \tag{27}$$

is not straightforward, and we omit the details here. Armed with those properties, the fundamental property of the logical relation follows by induction on the structure of expressions; in particular we get $\emptyset \vdash e : \tau \Rightarrow [\![e]\!] \lhd_\tau e$. This, combined with the definition of $\lhd_{\mathtt{bool}}$ yields (23). $\quad\square$

Using Theorem 27 we can prove many contextual equivalences in PNA, such as those in Example 23, in a straightforward manner via the denotational semantics. For example (18) is proved by the following argument. Since we identify expressions up to α-equivalence, for any given $a' \in \mathbb{A}$ we can pick a representative expression $\nu a.\, \lambda x : \mathtt{name} \to (x = a)$ such that $a \neq a'$, then

$$[\![\nu a.\, \lambda x : \mathtt{name} \to (x = a)]\!]\, a'$$
$$= (a\backslash[\![\lambda x : \mathtt{name} \to (x = a)]\!])\, a' \quad \text{by definition in Figure 5}$$
$$= a\backslash([\![\lambda x : \mathtt{name} \to (x = a)]\!]\, a') \quad \text{by (13), since } a \neq a'$$
$$= a\backslash\mathrm{false} \quad \text{as } a \neq a'$$
$$= \mathrm{false}$$
$$= [\![\lambda x : \mathtt{name} \to \mathtt{if}\ x = x\ \mathtt{then}\ \mathtt{F}\ \mathtt{else}\ \mathtt{F}]\!]\, a'.$$

Similarly $[\![\nu a.\, \lambda x : \mathtt{name} \to (x = a)]\!]\, \bot = \bot = [\![\lambda x : \mathtt{name} \to \mathtt{if}\ x = x\ \mathtt{then}\ \mathtt{F}\ \mathtt{else}\ \mathtt{F}]\!]\, \bot$. Hence $[\![\nu a.\, \lambda x : \mathtt{name} \to (x = a)]\!] = [\![\lambda x : \mathtt{name} \to \mathtt{if}\ x = x\ \mathtt{then}\ \mathtt{F}\ \mathtt{else}\ \mathtt{F}]\!]$ and so (18) holds by Theorem 27. To prove example (19) one can combine the definition in Figure 5 with the fact that if $a, a' \# f \in (\mathbb{A}_\bot \to 2_\bot) \to 2_\bot$, then $f\, a = f((a\, a') \cdot a') = (a\, a') \cdot (f\, a') = f\, a'$ (since $f\, a' \in 2_\bot$). In contrast to the situation here, properties like (18) and (19) for the ν-calculus, which uses generative rather than Odersky-style local names, can be hard to establish whether one uses operational or denotational techniques. See Tzevelekos [43], which focusses on an equivalence like (19).

9. Full abstraction for PNA$^+$

Plotkin [30] famously proves that the Scott domain model of PCF is computationally adequate, but not fully abstract: equality of denotations implies, but is not implied by, PCF contextual equivalence. Furthermore, he shows that the Scott model becomes fully abstract once one extends PCF with a parallel-or construct.

Moving to nominal Scott domains and PNA, Plotkin's negative result can certainly be extended to show that the converse of Theorem 27 fails to hold. We do not yet know what happens if one adds just parallel-or to PNA (see Section 10). However, adding not only parallel-or, but also operational versions of the uniform-compact functionals in Examples 14 and 15, we will show that the nominal Scott domain model is fully abstract for contextual equivalence in the extended language.

Note that this section does *not* give a full abstraction result for the more common generative local names used for example in FreshML or the ν-calculus. Generative names can be modelled adequately in **Nsd** through a continuation monad as described at the end of Section 6, but full abstraction fails in this model, because of the results of Stark [36, Page 66].

Definition 28. The language PNA$^+$ is obtained by extending PNA with expressions for parallel-or, for existentially quantifying over $\mathtt{name}$ ('there exists some $x : \mathtt{name}$ such that...') and for forming definite descriptions over $\mathtt{name}$ ('the unique $x : \mathtt{name}$ such that...'). The syntax, typing and evaluation rules for this extension are given in Figure 6. Contextual equivalence for the extended language $\Gamma \vdash e \cong_{\mathsf{PNA}^+} e' : \tau$ is defined in the same way as it is for PNA in Definition 22.

Remark 29. The addition of existential quantification and definite description is mainly motivated by the need for them in our proof of

$$e ::= \qquad\qquad\qquad\qquad\quad expressions$$
$$\cdots \qquad\qquad\qquad\qquad\text{(as for PNA)}$$
$$e \,\texttt{por}\, e \qquad\qquad\qquad\qquad parallel\text{-}or$$
$$\texttt{ex}\, x.\, e \qquad\qquad \text{existential name quantification}$$
$$\texttt{the}\, x.\, e \qquad\qquad \text{definite name description}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1\,\texttt{por}\,e_2 : \texttt{bool}}$$

$$\frac{\Gamma, x : \texttt{name} \vdash e : \texttt{bool}}{\Gamma \vdash \texttt{ex}\,x.\,e : \texttt{bool}} \qquad \frac{\Gamma, x : \texttt{name} \vdash e : \texttt{bool}}{\Gamma \vdash \texttt{the}\,x.\,e : \texttt{name}}$$

$$\frac{e_1 \Downarrow \texttt{T}}{\texttt{por}\,e_1\,e_2 \Downarrow \texttt{T}} \qquad \frac{e_2 \Downarrow \texttt{T}}{e_1\,\texttt{por}\,e_2 \Downarrow \texttt{T}} \qquad \frac{e_1 \Downarrow \texttt{F} \qquad e_2 \Downarrow \texttt{F}}{e_1\,\texttt{por}\,e_2 \Downarrow \texttt{F}}$$

$$\frac{e[a/x] \Downarrow \texttt{T}}{\texttt{ex}\,x.\,e \Downarrow \texttt{T}} \qquad \frac{a' \mathbin{\#} e \qquad (\forall b \in \mathrm{fn}(e) \cup \{a'\})\, e[b/x] \Downarrow \texttt{F}}{\texttt{ex}\,x.\,e \Downarrow \texttt{F}}$$

$$\frac{\begin{array}{c} e[a/x] \Downarrow \texttt{T} \qquad a' \mathbin{\#} (e, a) \\ (\forall b \in (\mathrm{fn}(e) - \{a\}) \cup \{a'\})\, e[b/x] \Downarrow \texttt{F} \end{array}}{\texttt{the}\,x.\,e \Downarrow a}$$

$$\llbracket e_1\,\texttt{por}\,e_2 \rrbracket \rho = por\,(\llbracket e_1 \rrbracket \rho)\,(\llbracket e_2 \rrbracket \rho)$$
$$\llbracket \texttt{ex}\,x.\,e \rrbracket \rho = exists_{\mathbb{A}}(\llbracket \lambda x : \texttt{name} \to e \rrbracket \rho)$$
$$\llbracket \texttt{the}\,x.\,e \rrbracket \rho = the_{\mathbb{A}}(\llbracket \lambda x : \texttt{name} \to e \rrbracket \rho)$$

Figure 6. PNA$^+$

full abstraction (Theorem 30). Existential quantification for numbers (rather than, as here, for names) occurs in Plotkin's original PCF paper [30], and definite description has a long history in logic, but it is harder to motivate from a programming language perspective. In fact, a definite description functional for numbers rather than for names is not computable. The computability of $\texttt{ex}\,x.\,e$ and $\texttt{the}\,x.\,e$ provide an example of the phenomenon of 'finite modulo symmetry' mentioned in the Introduction. For example, to prove $\texttt{ex}\,x.\,e \Downarrow \texttt{F}$, we just have to pick one of the infinitely many atomic names a' that do not occur free in e and then show $e[a'/x] \Downarrow \texttt{F}$ and $e[b/x] \Downarrow \texttt{F}$ for each of the finitely many atomic names b that do occur free in e. This works because the equivariance property (16) of evaluation implies that if $e[a'/x] \Downarrow \texttt{F}$, then $e[a''/x] \Downarrow \texttt{F}$ holds for any a'' not occurring free in e.

The denotational semantics of PNA$^+$ expressions is given by extending the definition in Figure 5 with the clauses at the bottom of Figure 6. Here $por \in \mathrm{K}(2_\perp \to 2_\perp \to 2_\perp)$ is the usual parallel-or function satisfying

$$por\,d\,d' = \begin{cases} \text{true} & \text{if } d = \text{true or } d' = \text{true} \\ \text{false} & \text{if } d = \text{false and } d' = \text{false} \\ \perp & \text{otherwise;} \end{cases}$$

$exists_{\mathbb{A}} \in \mathrm{K}((\mathbb{A}_\perp \to 2_\perp) \to 2_\perp)$ is as in Example 14, and $the_{\mathbb{A}} \in \mathrm{K}((\mathbb{A}_\perp \to 2_\perp) \to \mathbb{A}_\perp)$ is as in Example 15.

Theorem 30 (full abstraction for PNA$^+$). *For all well-typed expressions $\Gamma \vdash e : \tau$ and $\Gamma \vdash e' : \tau$ in PNA$^+$, we have*

$$\llbracket e \rrbracket = \llbracket e' \rrbracket \in \llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket \Leftrightarrow \Gamma \vdash e \cong_{\mathsf{PNA+}} e' : \tau. \qquad (\mathrm{FA}_\tau)$$

The sketch of the proof of this result occupies the rest of this section.

The extension of Theorem 27 to PNA$^+$ is straightforward and gives us the left-to-right implication in (FA_τ). Establishing the re-

verse implication inevitably leads to an investigation of the definability of elements of the nominal Scott domain model by PNA$^+$ expressions. However, our proof of the right-to-left implication in (FA_τ) does not exactly follow any of the proof patterns linking definability with full abstraction surveyed by Curien [10]. In particular, we only know that the uniform-compact elements of the nominal Scott domain $\llbracket \tau \rrbracket$ are definable in PNA$^+$ for certain types that avoid the use of function types $\tau_1 \to \tau_2$ in which the nominal Scott domain $\llbracket \tau_2 \rrbracket$ might contain elements with non-empty support. So $\tau_2 = \texttt{nat}$ is OK, but $\tau_2 = \texttt{name}$ is not, for example. This leads us to make the following definition.

Definition 31 (simple types). Let $\mathrm{Styp} \subseteq \mathrm{Typ}$ be the subset of the collection of types (Figure 1) given by the following grammar:

$$\sigma ::= \texttt{nat} \mid \texttt{name} \mid \sigma \times \sigma \mid \sigma \to \texttt{nat}.$$

The following lemma is the key to the usefulness of simple types. It is where the presence of '$\texttt{the}\,x.\,e$' expressions in PNA$^+$ gets used.

Lemma 32. *Any type τ is a PNA$^+$-definable retract of a simple type $\sigma \in \mathrm{Styp}$, meaning that there are closed PNA$^+$ expressions $\emptyset \vdash i : \tau \to \sigma$ and $\emptyset \vdash r : \sigma \to \tau$ with $\llbracket \lambda x : \tau \to r\,(i\,x) \rrbracket = id_{\llbracket \tau \rrbracket}$.*

Proof. First note that because name abstraction satisfies a form of η-expansion ($\llbracket \alpha a.\,(e\,@\,a) \rrbracket = \llbracket e \rrbracket$, if $a \notin \mathrm{fn}(e)$), each $\delta\tau$ is a PNA$^+$-definable retract of $\texttt{name} \to \tau$ (PNA-definable, in fact), via

$$i \triangleq \lambda x : \delta\tau \to \lambda y : \texttt{name} \to x\,@\,y \qquad (28)$$
$$r \triangleq \lambda(f : \texttt{name} \to \tau) \to \alpha a.\,f\,a \qquad (29)$$

(*cf.* [27, Theorem 2.13]). Secondly, using '$\texttt{the}\,x.\,e$' expressions, we also have that $\texttt{name}$ itself is a PNA$^+$-definable retract of $\texttt{name} \to \texttt{nat}$, via

$$i \triangleq \lambda x : \texttt{name} \to \lambda y : \texttt{name} \to \texttt{if } x = y \texttt{ then } \texttt{0} \texttt{ else } \texttt{S}\,\texttt{0}$$
$$r \triangleq \lambda(f : \texttt{name} \to \texttt{nat}) \to \texttt{the}\,x.\,\texttt{zero}\,(f\,x).$$

Thirdly, again using '$\texttt{the}\,x.\,e$' expressions, one can also show that $\texttt{term}$ is a PNA$^+$-definable retract of $\texttt{nat} \times (\texttt{name} \to \texttt{nat})$ (proof omitted). Finally, it is not hard to see that $\texttt{bool}$ is a PNA$^+$-definable (actually PCF-definable) retract of $\texttt{nat}$.

Using these four facts, one can proceed by induction on the structure of types to show that each τ is a PNA$^+$-definable retract of some simple type. $\qquad\blacksquare$

Lemma 33. *If τ is a PNA$^+$-definable retract of σ, then full abstraction at σ implies full abstraction at τ, i.e. $(\mathrm{FA}_\sigma) \Rightarrow (\mathrm{FA}_\tau)$.*

Proof. Given i and r as in Lemma 32, suppose (FA_σ) holds and that $\Gamma \vdash e \cong_{\mathsf{PNA+}} e' : \tau$, then we have $\Gamma \vdash i\,e \cong_{\mathsf{PNA+}} i\,e' : \sigma$ by compositionality of $\cong_{\mathsf{PNA+}}$. Thus by (FA_σ), for any $\rho \in \llbracket \Gamma \rrbracket$ we have $\llbracket i \rrbracket (\llbracket e \rrbracket \rho) = \llbracket i\,e \rrbracket \rho = \llbracket i\,e' \rrbracket \rho = \llbracket i \rrbracket (\llbracket e' \rrbracket \rho)$. We know that $\llbracket i \rrbracket$ is injective since it has a left inverse $\llbracket r \rrbracket$, and hence $(\forall \rho \in \llbracket \Gamma \rrbracket)\,\llbracket e \rrbracket \rho = \llbracket e' \rrbracket \rho$. Thus $\llbracket e \rrbracket = \llbracket e' \rrbracket \in \llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket$. $\qquad\blacksquare$

Combining these two lemmas, to prove Theorem 30 it suffices to show that (FA_σ) holds for all simple types $\sigma \in \mathrm{Styp}$. As surveyed in Curien [10], this follows from definability of all uniform-compact elements of the nominal Scott domains $\llbracket \sigma \rrbracket$. That is, for all $u \in \mathrm{K}\llbracket \sigma \rrbracket$ we wish to prove that $u = \llbracket e \rrbracket$ for some variable-closed PNA$^+$ expression $\emptyset \vdash e : \sigma$.

Remark 34 (definability of uniform-compact elements). Are all the uniform-compact elements of the nominal Scott domain $\llbracket \tau \rrbracket$ definable in PNA$^+$, for *any* type τ? We introduced simple types because we did not find a way to prove such a definability result at all types. We succeed in showing uniform-compact definability at

simple types essentially because the codomains of simple functions are restricted to `nat`, which makes life much easier in this setting. If all the definable retracts used in our proof of Theorem 30 were actually definable *embedding-projection pairs* (in the sense that $[\![\lambda x : \sigma \to i\,(r\,x)]\!] \sqsubseteq id_{[\![\sigma]\!]}$ holds), then uniform-compact definability at simple types would immediately imply uniform-compact definability at any type.[4] Unfortunately for name abstraction types, (28) and (29) do *not* form an embedding-projection pair. There remains the possibility that embedding-projection pairs as above can be used to show uniform-compact definability at all types for a simpler language without name abstraction, such as the $\lambda\nu$-calculus of Odersky [24] extended with fixed points.

The proof of uniform-compact definability at simple types in principle follows the structure of the traditional proof by Plotkin [30]. A modern account of this proof can be found in Streicher's book [38]. However, in our nominal setting many uses of finite subsets in the traditional proof are replaced by uses of orbit-finite subsets and their presentation as orbit-finite hulls (Theorem 8). The definition of $\mathrm{hull}_A F$ involves existential quantification over finite permutations of $\mathbb{A}$, and for the definability proof we need to reduce this to existential quantification over elements of $\mathbb{A}$. This is where the presence of `ex`$\,x.\,e$ expressions in PNA$^+$ gets used (along with a traditional use of `por`) to prove the following two crucial lemmas. Neither is trivial to prove. In particular Lemma 36 works by a subtle case distinction over all the different ways the atomic names in the supports of u and u' can overlap.

Lemma 35. $\bigsqcup \mathrm{hull}_A\{[u\,,\mathrm{true}]\}$ *is* PNA$^+$-*definable for every* $\sigma \in$ Styp, $u \in \mathrm{K}[\![\sigma]\!]$ *and* $A \subseteq_f \mathbb{A}$.

Lemma 36. *Suppose that* $\sigma \in$ Styp *satisfies*

- *for all* $v, v' \in \mathrm{K}[\![\sigma]\!]$ *that do not have an upper bound in* $[\![\sigma]\!]$, $[v\,,\mathrm{true}] \sqcup [v'\,,\mathrm{false}]$ *is* PNA$^+$-*definable.*

Then $\bigsqcup \mathrm{hull}_A\{[u\,,\mathrm{true}],[u'\,,\mathrm{false}]\}$ *is* PNA$^+$-*definable for any* $u, u' \in \mathrm{K}[\![\sigma]\!]$ *and* $A \subseteq_f \mathbb{A}$ *satisfying:*

- *for all finite permutations* $\pi : \mathbb{A} \cong \mathbb{A}$ *satisfying* $\pi \mathrel{\#} A$ *(see Definition 7), it holds that* u *and* $\pi \cdot u'$ *do not have an upper bound in* $[\![\sigma]\!]$.

Using these two lemmas one can show by simultaneous induction on the structure of $\sigma \in$ Styp that

- u and $[u\,,\mathrm{true}]$ are definable for all uniform-compact elements $u \in \mathrm{K}[\![\sigma]\!]$, and

- $[u\,,\mathrm{true}] \sqcup [u'\,,\mathrm{false}]$ is definable whenever $u, u' \in \mathrm{K}[\![\sigma]\!]$ are uniform-compact elements that do not have an upper bound in $[\![\sigma]\!]$.

Most of the work lies in the case for functions types, which for simple types are of the form, $\sigma \to$ `nat`. By Theorem 8 and 12 each uniform-compact element u of $[\![\sigma \to$ `nat`$]\!]$ can be represented by $u = \bigsqcup \mathrm{hull}_A F$ for some $A \subseteq_f \mathbb{A}$, $F = \{[u_1\,,n_1],..,[u_k\,,n_k]\}$, $u_1,..,u_k \in \mathrm{K}[\![\sigma]\!]$ and $n_1,..,n_k \in \mathbb{N}$. One has to perform another induction on the size of F and make a case distinction based on the existence of $[u'\,,n'], [u''\,,n''] \in F$ such that for all $\pi \mathrel{\#} A$ the compact elements u' and $\pi \cdot u''$ have no upper bound in $[\![\sigma]\!]$. Using Lemmas 35 and 36 the proof goes through following the structure of Streicher [38, Theorem 13.9], thereby showing full abstraction for PNA$^+$.

10. Open problems

1. *Failure of full abstraction in the nominal Scott domain model. Is* **Nsd** *fully abstract for just* PNA+por? *Is it necessary to add*

both `ex`$\,x.\,e$ *and* `the`$\,x.\,e$ *to* PNA+por *in order to obtain full abstraction?*

We do not yet have examples of contextually equivalent expressions in PNA+por, PNA+por+ex, or PNA+por+the that have different denotations in **Nsd**. Probably the method of logical relations can be adapted to establish such contextual equivalences, but we have yet to pursue this.

2. *Is there a fully abstract model of* PNA *based on games in nominal sets?*

Just as PCF is of more interest from a programming point of view than PCF+por, we regard PNA (suitably extended with recursive types) as a 'pure' version of FreshML that is potentially useful for functional programming with syntactical data involving binders. Game semantics provided an interesting solution for the original full abstraction problem for PCF [4, 16], and its nominal version has provided computationally useful, fully abstract models of generative local state [3, 18, 23, 41]. Can nominal game semantics provide a similar thing for PNA?

3. *Is there a nominal Scott domain semantics for the form of nominal computation embodied by the* Nλ *language* [6]?

With Nλ, Bojańczyk *et al.* extend the simply-typed λ-calculus with a collection type—representing orbit-finite subsets (Section 2) via a syntax for orbit-finite hulls (Definition 7).[5] It is natural to consider adding fixed point recursion to this language, with a denotational semantics using nominal domains rather than nominal sets. The denotational semantics of such an extension of Nλ will require the development of *orbit-finite* power domains $\mathrm{F_n} D$ in **Nsd**, whose uniform-compact elements are orbit-finite subsets of the uniform-compact elements of D.

4. *What recursive domain equations can be solved in* **Nsd**?

In his thesis, Shinwell [33, Section 4.5] shows that the traditional method for constructing minimally invariant solutions for locally continuous functors of mixed variance can be applied to the simple notion of nominal domain given by nominal posets with joins of finitely supported ω-chains. This can be extended to udcpos and we expect it can also be used for nominal Scott domains, but we have yet to check the details. An interesting alternative approach is to develop a nominal version of Scott's information systems [32] and construct solutions for recursive domain equations via inductively defined nominal sets of information tokens. We have begun to develop such a theory of *nominal Scott information systems* in which the role of finite sets is replaced by orbit-finite nominal sets [unpublished]. From a logical point of view [2], nominal information systems are presentations of non-trivial nominal posets *with all orbit-finite meets*, rather than just finite meets. We expect this machinery can be used to good effect for the orbit-finite power domain construct mentioned above, as well as for a version for nominal Scott domains of Moggi's monad for dynamic allocation [21, Section 4.14] featuring a *freely generated* uniform-continuous name restriction operation [27, Remark 2.8].

11. Conclusion

The results in this paper provide further evidence for how a semantic theory (domain theory in this case) is enhanced by using nominal sets: we gain the ability to model constructs involving names and their symmetries while preserving many aspects of the classical theory. The complications arising from the use of nominal sets

---

[4] Thanks to a referee for pointing this out.

[5] Their paper [6] is concerned with general 'Fraïssé nominal sets'. Here we restrict our attention to the 'equality symmetry' and nominal sets in the original sense.

are feasible and somehow orthogonal to the other developments. At the same time, their use gives access to new constructs that are far from trivial. This is the case for the notion of *orbit-finite subset*, which formalizes the important idea of finiteness modulo symmetry within nominal sets. We agree with Bojańczyk *et al.* [6] that this is an important notion with many potential applications. Here we have used it to develop a nominal domain theory that, via our full abstraction result, has a good fit with higher-type computation involving local names and name abstractions.

Acknowledgments

References

[1] URL http://www.citeulike.org/group/11951/.

[2] S. Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51:1–77, 1991.

[3] S. Abramsky, D. R. Ghica, A. S. Murawski, C.-H. L. Ong, and I. D. B. Stark. Nominal games and full abstraction for the nu-calculus. In *Proc. LICS 2004*, pages 150–159. IEEE Computer Society Press, 2004.

[4] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Information and Computation*, 163(2):409–470, 2000.

[5] S. Abramsky and A. Jung. Domain theory. In *Handbook of Logic in Computer Science, Volume 3. Semantic Structures*, chapter 1. Oxford University Press, 1994.

[6] M. Bojańczyk, L. Braud, B. Klin, and S. Lasota. Towards nominal computation. In *Proc. POPL 2012*, pages 401–412. ACM Press, 2012.

[7] M. Bojańczyk, B. Klin, and S. Lasota. Automata with group actions. In *Proc. LICS 2011*, pages 355–364. IEEE Computer Society Press, 2011.

[8] M. Bojańczyk and S. Lasota. A machine-independent characterization of timed languages. In *Proc. ICALP 2012, Part II*, volume 7392 of *LNCS*, pages 92–103. Springer-Verlag, 2012.

[9] V. Ciancia and U. Montanari. Symmetries, local names and dynamic (de)-allocation of names. *Information and Computation*, 208(12): 1349–1367, 2010.

[10] P.-L. Curien. Definability and full abstraction. In *Computation, Meaning and Logic, Articles dedicated to Gordon Plotkin*, volume 172 of *ENTCS*, pages 301–310. Elsevier, 2007.

[11] M. J. Gabbay. A study of substitution, using nominal techniques and Fraenkel-Mostowski sets. *Theoretical Computer Science*, 410(12–13): 1159–1189, 2009.

[12] M. J. Gabbay. Foundations of nominal techniques: Logic and semantics of variables in abstract syntax. *Bulletin of Symbolic Logic*, 17(2): 161–229, 2011.

[13] M. J. Gabbay and V. Ciancia. Freshness and name-restriction in sets of traces with names. In *Proc. FOSSACS 2011*, volume 6604 of *LNCS*, pages 365–380. Springer-Verlag, 2011.

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

[15] F. Gadducci, M. Miculan, and U. Montanari. About permutation algebras, (pre)sheaves and named sets. *Higher-Order Symb. Computation*, 19:283–304, 2006.

[16] J. M. E. Hyland and C.-H. L. Ong. On full abstraction for PCF: I, II and III. *Information and Computation*, 163(2):285–408, 2000.

[17] P. T. Johnstone. *Sketches of an Elephant, A Topos Theory Compendium, Volumes 1 and 2*. Number 43–44 in Oxford Logic Guides. Oxford University Press, 2002.

[18] J. Laird. A game semantics of names and pointers. *Annals of Pure and Applied Logic*, 151(2):151–169, 2008.

[19] S. Lösch and A. M. Pitts. Relating Two Semantics of Locally Scoped Names. In *Proc. CSL 2011*, volume 12 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 396–411, 2011.

[20] G. Markowsky. Chain-complete p.o. sets and directed sets with applications. *Algebra Universalis*, 6:53–68, 1976.

[21] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Department of Computer Science, University of Edinburgh, 1989.

[22] U. Montanari and M. Pistore. π-Calculus, structured coalgebras and minimal HD-automata. In *Proc. MFCS 2000*, volume 1893 of *LNCS*, pages 569–578. Springer-Verlag, 2000.

[23] A. S. Murawski and N. Tzevelekos. Algorithmic games for full ground references. In *Proc. ICALP 2012, Part II*, volume 7392 of *LNCS*, pages 312–324. Springer-Verlag, 2012.

[24] M. Odersky. A functional theory of local names. In *Proc. POPL 1994*, pages 48–59. ACM Press, 1994.

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

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

[27] A. M. Pitts. Structural recursion with locally scoped names. *Journal of Functional Programming*, 21(3):235–286, 2011.

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

[29] A. M. Pitts and I. D. B. Stark. Observable properties of higher order functions that dynamically create local names, or: What's new? In *Proc. MFCS 1993*, volume 711 of *LNCS*, pages 122–141. Springer-Verlag, 1993.

[30] G. D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[31] F. Pottier. Static name control for FreshML. In *Proc. LICS 2007*, pages 356–365. IEEE Computer Society Press, 2007.

[32] D. S. Scott. Domains for denotational semantics. In *Proc. ICALP 1982*, volume 140 of *LNCS*, pages 577–613. Springer-Verlag, 1982.

[33] M. R. Shinwell. *The Fresh Approach: Functional Programming with Names and Binders*. PhD thesis, University of Cambridge, 2005.

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

[35] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Proc. ICFP 2003*, pages 263–274. ACM Press, 2003.

[36] I. D. B. Stark. *Names and Higher-Order Functions*. PhD thesis, University of Cambridge, Dec. 1994.

[37] S. Staton. *Name-Passing Process Calculi: Operational Models and Structural Operational Semantics*. PhD thesis, University of Cambridge, 2007.

[38] T. Streicher. *Domain-Theoretic Foundations of Functional Programming*. World Scientific, Singapore, 2006.

[39] D. C. Turner. *Nominal Domain Theory for Concurrency*. PhD thesis, University of Cambridge, 2009.

[40] D. C. Turner and G. Winskel. Nominal domain theory for concurrency. In *Proc. CSL 2009*, volume 5771 of *LNCS*, pages 546–560. Springer-Verlag, 2009.

[41] N. Tzevelekos. *Nominal Game Semantics*. PhD thesis, University of Oxford, 2008.

[42] N. Tzevelekos. Fresh-register automata. In *Proc. POPL 2011*, pages 295–306. ACM Press, 2011.

[43] N. Tzevelekos. Program equivalence in a simple language with state. *Computer Languages, Systems and Structures*, 38(2):181–198, 2012.