Nominal Type Theory by Nullary Internal Parametricity

Antoine Van Muylder

DistriNet, KU Leuven, Belgium

Andreas Nuyts

DistriNet, KU Leuven, Belgium

Dominique Devriese

DistriNet, KU Leuven, Belgium

Abstract

There are many ways to represent the syntax of a language with binders. In particular, nominal frameworks are metalanguages that feature (among others) name abstraction types, which can be used to specify the type of binders. The resulting syntax representation (nominal data types) makes alpha-equivalent terms equal, and features a name-invariant induction principle. It is known that name abstraction types can be presented either as existential or universal quantification on names. On the one hand, nominal frameworks use the existential presentation for practical reasoning since the user is allowed to match on a name-term pattern where the name is bound in the term. However inference rules for existential name abstraction are cumbersome to specify/implement because they must keep track of information about free and bound names at the type level. By contrast universal name abstractions are easier to specify since they are treated not as pairs, but as functions consuming fresh names. Yet the ability to pattern match on such functions is seemingly lost. In this work we show that this ability and others are recovered in a type theory consisting of (1) nullary (i.e. 0-ary) internally parametric type theory (nullary PTT) (2) a type of names Nm and a novel name induction principle (3) nominal data types. This extension of nullary PTT can act as a legitimate nominal framework. Indeed it has universal name abstractions, nominal pattern matching, a freshness type former, name swapping and local-scope operations and (non primitive) existential name abstractions. We illustrate how term-relevant nullary parametricity is used to recover nominal pattern matching. Our main example involves synthetic Kripke parametricity.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Nominal logic, Parametricity

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Nominal syntax There are many ways to formally define the syntax of a language with binders. In particular, nominal frameworks [24, 31, 27, 30, 12, 33] are metalanguages featuring (among others) a primitive type of names Nm as well as a "name abstraction" type former which we write $@\mathsf{N} \multimap -$. Name abstraction types can be used to specify the type of binders of a given object language. For example, we can define the syntax of untyped lambda calculus (ULC) with the following data type.

```
data Ltm : 𝒰 where
  var : Nm → Ltm
  app : Ltm → Ltm → Ltm
  lam : (@N ⊸ Ltm) → Ltm
```

The resulting representation is called a "nominal data type" or "nominal syntax" and offers several advantages: α-equivalent terms are equal, names are more readable/writable than e.g. De Bruijn indices, indexing on contexts can often be dropped, and importantly nominal

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

data types feature a name-invariant induction principle. A drawback of this representation is that it is not definable in plain type theory.

Inference rules for the name abstraction type former can be defined in two ways: either in an existential/positive, or in a universal/negative fashion. Interestingly, these presentations are semantically equivalent [30, 29] but syntactically have different pros and cons.

On the one hand, the existential presentation makes the name abstraction type former behave as an existential quantification on names. So intuitively an element of that type, i.e. a name abstraction, is a *pair* $\langle a, t\rangle$ where the name a is bound in the term t. Additionally the system ensures that the pair is handled up to α-renaming. This presentation is convenient since the user is allowed to *pattern match* on such "binding" name-term pairs, an ability that we call nominal pattern matching. The following example is Example 2.1 from [31] and illustrates this ability. It is a (pseudo-) FreshML program computing the equality modulo α-renaming of two existential name abstractions. In the example A has decidable equality $\mathsf{eq}_A : A \to A \to \mathsf{Bool}$ and the existential name abstraction type is written $@\mathsf{N} \cdot -$.

```
eqabs : (@N · A) → (@N · A) → Bool
eqabs ⟨x₀,a₀⟩ ⟨x₁,a₁⟩ = eq_A (swap x₀,x₁ in a₀) a₁
```

Note (1) the occurrences of x_0, x_1 outside of $\langle x_0, a_0\rangle, \langle x_1, a_1\rangle$ and (2) the appearance of the swap operation exchanging free names.

Nominal pattern matching is convenient and for that reason nominal frameworks use the existential presentation for practical reasoning on nominal syntax. However, inference rules for existential abstraction types are cumbersome to specify. The issue is that a name x is considered fresh in a binding pair $\langle x, y\rangle$ and that information must be encoded at the type level and propagated when pattern matching. The consequence is that the rules for existential abstraction types are polluted with typal freshness information. This makes the implementation of such rules in a proof assistant environment harder.

On the other hand, the universal presentation treats name abstractions not as pairs, but rather as *functions* consuming fresh names (a.k.a. affine or fresh functions), as in [27, 12]. This has several consequences: names can only be used when they are in scope, no explicit swapping primitive is needed and the inference rules are straightforward to specify. However one seems to lose the important ability to pattern match.

In this paper we will propose a new foundation for nominal frameworks relying on the notion of *parametricity*, which we describe now. Parametricity is a language property expressing that all values automatically satisfy properties that can be derived structurally from their types [28]. More precisely, parametricity asserts that types behave as reflexive graphs [6]. For example the parametricity of a function $f : A \to B$ is a proof that f maps related inputs in A to related outputs in B, i.e. f is a morphism of reflexive graphs. Although the notion of parametricity is typically used in its binary form, the property can actually be considered generally for n-ary relations and n-ary graphs.

Parametricity can be regarded as a property that is defined and proven to hold externally about a dependent type theory (DTT). Indeed in [10] the parametricity property, or *translation*, is a map $[-]$ defined by induction on the types and terms of DTT. The translation $[A]$ of a type A asserts the parametricity property at A and the translation of a term a is a proof $[a] : [A]$. By contrast, n-ary internally parametric type theory (n-ary PTT for short) [11, 17, 19, 21, 2, 1] extends DTT with new type and term formers. These primitives make it possible to prove parametricity results *within* n-ary PTT.

To that end, n-ary PTT typically provides two kinds of primitives, whose syntax internalize aspects of reflexive graphs. Firstly, Bridge types are provided, which intuitively are to a type what an edge is to a reflexive graph. Syntactically, a bridge $q : \mathsf{Bridge}\, A\, a_0\, \ldots\, a_{n-1}$ at type

$K : \mathcal{U}$	$\mathsf{Bridge}\, K\, k_0\, k_1 \simeq \ldots$	$k_0 \equiv_K k_1 \simeq \ldots$
$A \to B$	$\forall a_0\, a_1.\ \mathsf{Bridge}\, A\, a_0\, a_1 \to \mathsf{Bridge}\, B\, (k_0\, a_0)\, (k_1\, a_1)$	$\forall\, a_0\, a_1.\, a_0 \equiv_A a_1 \to k_0\, a_0 \equiv_B k_1\, a_1$
$A \times B$	$\mathsf{Bridge}\, A\, (k_0\, .\mathsf{fst})(k_1\, .\mathsf{fst}) \times$ $\mathsf{Bridge}\, B\, (k_0\, .\mathsf{snd})(k_1\, .\mathsf{snd})$	$(k_0\, .\mathsf{fst}) \equiv_A (k_1\, .\mathsf{fst}) \times$ $(k_0\, .\mathsf{snd}) \equiv_B (k_1\, .\mathsf{snd})$
$\mathcal{U}$	$k_0 \to k_1 \to \mathcal{U}$	$k_0 \simeq k_1$

Figure 1 The Bridge and Path type former commute with some example type formers.

A between $a_0, \ldots, a_{n-1}$ is treated as a function $\mathsf{N} \to A$ out of a posited bridge interval N. The N interval contains n endpoints $(e_i)_{i<n}$ and q must respect these definitionally $q\, e_i = a_i$. When $n = 2$ this is similar to the "paths" of cubical type theory [35] which play the role of equality proofs in the latter. However, bridges do not satisfy various properties of paths, e.g. they can not be composed and one can not transport values of a type $P\ x$ over a bridge $\mathsf{Bridge}\ A\ x\ y$ to type $P\ y$. Moreover, contrary to paths, bridges may only be applied to variables $x : \mathsf{N}$ that do not appear freely in them, i.e. fresh variables. Functions with such a freshness side condition are also known as affine or fresh functions and we will use the symbol $\multimap$ instead of $\to$ for their type. Secondly, the other primitives of n-ary PTT make it possible to prove that the Bridge type former has a commutation law with respect to every other type former. Figure 1 lists some of these laws (equivalences) for arity $n = 2$ and compares the situation for bridges and paths. Path types are written $\equiv$.

The last column in Figure 1 states a collection of equivalences known as the Structure Identity Principle (SIP) in HoTT/UF. Concisely, it states that (path-)equality at a type K is equivalent to observational equality [3] at K. The Bridge column of Figure 1 shows an analogue Structure Relatedness Principle (SRP) [34] which expresses equivalence of K's Bridge type to the parametricity translation of K, i.e. the Bridge type former internalizes the parametricity translation for types *up to SRP equivalences*. Finally parametricity of a term k is the reflexivity bridge $\lambda(\_ : \mathsf{N}).\, k : \mathsf{Bridge}\, K\, k\, \ldots\, k$ mapped through the SRP equivalence at K. This is summarized in the slogan "parametricity = every term is related with itself".

Nullary Parametricity as Foundation for Nominal Type Theory

In this work, we propose nullary PTT (i.e. 0-ary PTT) as the foundation for nominal dependent type theory. First, we make the simple observation that universal name abstraction types have the same rules as the nullary (i.e. 0-ary) Bridge types from nullary PTT: nullary bridges and universal abstractions are simply affine functions. This has been noticed in various forms [20, 14] but never exploited, to our knowledge. Second, we show that, on its own, nullary PTT (i.e. 0-ary PTT) already supports important features of nominal frameworks: universal name abstractions (bridge types), typal freshness, (non-primitive) existential name abstractions, a name swapping operation and the local-scoping primitive ν of [27] (roughly, the latter primitive is used to witness freshness). Third we can recover nominal pattern matching in this parametric setting by extending nullary PTT with a full-fledged type $\vdash \mathsf{Nm}$. It comes equipped with a novel dependent eliminator $\mathsf{ind}_{\mathsf{Nm}}$ called *name induction*, which expresses for a term $\Gamma \vdash n : \mathsf{Nm}$ and a bridge variable x in Γ, that n is either just x, or x is *fresh* in n.

Nominal pattern matching can be recovered via an extended version of the nullary SRP: the Structure Abstraction Principle (SAP), as we call it. The SAP expresses that the nullary bridge former $@\mathsf{N} \multimap -$ commutes with every type former in a specific way, including (1) standard type formers, (2) the Nm type (3) nominal data types. For standard type formers, the SAP asserts that name abstraction commutes as one might expect, e.g. $(@\mathsf{N} \multimap A \to B) \simeq ((@\mathsf{N} \multimap A) \to (@\mathsf{N} \multimap B))$. However, for Nm, the SAP asserts that $(@\mathsf{N} \multimap \mathsf{Nm}) \simeq 1 + \mathsf{Nm}$, which can be proved by name induction. The SAP also holds for nominal data types. For example, the SAP at the Ltm nominal data type asserts that that there is an equivalence e between $@\mathsf{N} \multimap \mathsf{Ltm}$ and the following data type. Intuitively, it corresponds to Ltm terms with Nm-shaped holes and we say it ought to be the nullary parametricity translation of Ltm.

```
data Ltm₁ : 𝓤 where
  hole : Ltm₁
  var : Nm → Ltm₁
  app : Ltm₁ → Ltm₁ → Ltm₁
  lam : (@N ⊸ Ltm₁) → Ltm₁
```

Nominal pattern matching can now be recovered: for a term $\mathsf{lam}\, g : \mathsf{Ltm}$: we have $g : @\mathsf{N} \multimap \mathsf{Ltm}$ and equivalently $e\, g : \mathsf{Ltm}_1$ for which the induction principle of Ltm_1 applies.

Contributions and Outline In this paper, we propose nullary PTT (i.e. 0-ary PTT) as the foundation for nominal dependent type theory and evidence for its suitability. Specifically:

- In Section 2, we define a variant of the univalent parametric type theory of [11] where (1) we replace the arity $n = 2$ by $n = 0$, (2) we construct a type Nm from the bridge interval $@\mathsf{N}$ and provide a name induction principle. We explain how the primitives validate our Structure Abstraction Principle (SAP) and discuss semantics and soundness.
- In Section 3, we systematically discuss the inference rules of existing nominal frameworks for typal freshness, name swapping, the local-scoping primitive ν, existential and universal name abstractions. We explain in detail how they can all be (adapted and) implemented in terms of nullary PTT primitives.
- Section 4 demonstrates examples of nominal recursion in a nullary parametric setting. Section 4.1 shows that the nullary translation of data types lets us emulate nominal pattern matching: functions that are defined by matching on patterns which bind variables. In Section 4.2 we provide a novel example of a function f defined by nominal recursion, whose correctness proof requires computing its nullary translation $[f_0]$, i.e. the parametricity of f is term-relevant. Specifically, we connect the Ltm nominal data type to a nominal HOAS representation and our proofs externalize to Atkey's Kripke parametricity model [5].

Relational parametric cubical type theory has already been implemented, particularly as an extension of the mature proof assistant Agda [34]. As such, our work offers a clear path to a first practical implementation of nominal type theory. To our knowledge, we are also the first to make explicit and active use of nullary parametricity. We discuss related work in more detail in Section 5.

2 Nullary PTT

In this section we present the nullary internally parametric type theory (nullary PTT) which we use as a nominal framework in the next sections. It is an extension of the binary PTT of Cavallo and Harper (CH type theory, [11]), where we have replaced the arity 2 by 0. More precisely, our rules are obtained by (1) considering the rules of the parametricity primitives of the CH binary PTT and replacing the arity 2 by 0 instead and (2) adding novel rules to turn the bridge interval $@\mathsf{N}$ into a fully-fledged type Nm, including a name induction principle.

Apart from its parametricity primitives, the CH theory is a cubical type theory, and so is our system. We briefly explain what that means before reviewing the nullary parametricity primitives, and our rules for Nm. For the impatient reader, the relevant rules of our system appear in Figure 2 and $\mathsf{Gel}\, A\, x$ can be understood as "the a's in A for which x is fresh".

Cubical type theory Cubical type theory (cubical TT) is a form of homotopy type theory (HoTT, [32]) that adds new types, terms and equations on top of plain dependent type theory. These extra primitives make it possible, among other things, to prove univalence and more generally the SIP. Recall from Section 1 that the SIP gives, for each type former, a characterization of the equality type at that type former (see Figure 1).

Conveniently we will only need to know that the SIP holds in our system in order to showcase our examples. That is to say, we will not need to know about the exact primitives that cubical TT introduces. Nonetheless we list them for completeness: (1) the path interval I, dependent path types and their rules; paths play the role of equality proofs thus non-dependent path types are written $\equiv$ (2) the transport and cube-composition operations, a.k.a the Kan operations; these are used e.g. to prove transitivity of $\equiv$ (3) a type former to turn equivalences into paths in the universe, validating one direction of univalence (4) higher inductive types (HITs) if desired.

Additionally, in HoTT an equivalence $A \simeq B$ is by definition a function $A \to B$ with contractible fibers. We rather build equivalences using the following fact (Theorem 4.2.3 of [32]): Let $f : A \to B$ have a quasi-inverse, i.e. a map $g : B \to A$ satisfying the two roundtrip equalities $\forall a.\, g(f\, a) \equiv a$ and $\forall b.\, f(g\, b) \equiv b$. Then f can be turned into an equivalence $A \simeq B$.

2.1 Nullary CH

Let us explain the rules of Figure 2. We begin with the nullary primitives of the CH type theory since our rules for the bridge interval Nm depend on them.

Contexts There are two ways to extend a context. The first way is the usual "cartesian" comprehension operation where Γ gets extended with a type $\Gamma \vdash A\, \mathsf{type}$ resulting in a context $\Gamma, (a : A)$. The second, distinct way to extend a context Γ is an "affine" comprehension operation where Γ gets extended with a bridge variable x resulting in a context written $\Gamma, (x : @\mathsf{N})$. Note that $@\mathsf{N}$ is *not* a type and morally always appears on the left of $\vdash$ (the Bridge type former will be written $@\mathsf{N} \multimap -$ but this is just a suggestive notation).

Intuitively the presence of $@\mathsf{N}$ is required because the theory treats affine variables in a special way that is ultimately used to prove the SAP (briefly mentioned in Section 1). Specifically, terms, types and substitutions depending on an affine variable $x : @\mathsf{N}$ are not allowed to duplicate x in affine positions. So typechecking an expression that depends on $x : @\mathsf{N}$ may involve checking that x does not appear freely in some subexpressions. Since variables declared after x in the context may eventually be substituted by terms mentioning x, a similar verification must be performed for them.

More formally, such "freshness" statements about free variables are specified in the inference rules using a context restriction operation $-\backslash-$. If Γ is a context containing $(x : @\mathsf{N})$ then $\Gamma\backslash x$ is the context obtained from Γ by removing x itself, as well as all the cartesian (i.e. not $@\mathsf{N}$) variables to the right of x[1]. If $(x : @\mathsf{N}) \in \Gamma$ and $\Gamma\backslash x \vdash a : A$ we say that x is fresh in a.

[1] Path variables $i : \mathrm{I}$ and cubical constraints are not removed.

$$\frac{\Gamma, (x : @\mathsf{N}) \vdash A\ \mathsf{type}}{\Gamma \vdash (x : @\mathsf{N}) \multimap A\ \mathsf{type}}\ {\multimap}\mathrm{F} \qquad \frac{\Gamma, (x : @\mathsf{N}) \vdash a : A}{\Gamma \vdash \lambda\, x.\, a : (x : @\mathsf{N}) \multimap A}\ {\multimap}\mathrm{I}$$

$$\frac{(x : @\mathsf{N}) \in \Gamma \qquad \Gamma \backslash x \vdash a' : (y : @\mathsf{N}) \multimap A}{\Gamma \vdash a'\, x : A[x/y]}\ {\multimap}\mathrm{E} \qquad \frac{(x : @\mathsf{N}) \in \Gamma \qquad \Gamma \backslash x, (y : @\mathsf{N}) \vdash a : A}{\Gamma \vdash (\lambda\, y.\, a)\, x = a[x/y] : A[x/y]}\ {\multimap}\beta$$

$$\frac{\Gamma, (x : @\mathsf{N}) \vdash a'_0\, x = a'_1\, x : A}{\Gamma \vdash a'_0 = a'_1 : (x : @\mathsf{N}) \multimap A}\ {\multimap}\eta \qquad \frac{\textsc{ext prem.} \qquad \Gamma \backslash x, (y : @\mathsf{N}) \vdash a : A}{\Gamma \vdash \mathsf{ext}\, (\lambda\, a'\, y.\, b)\, x\, (a[x/y]) = b[(\lambda\, y.\, a)/a', x/y] : B[x/y, a[x/y]/a]}\ \textsc{ext}\beta$$

$$\frac{\begin{array}{c}(x : @\mathsf{N}) \in \Gamma \qquad \Gamma \backslash x, (y : @\mathsf{N}) \vdash A\ \mathsf{type} \qquad \Gamma \backslash x, (y : @\mathsf{N}), (a : A\, y) \vdash B\ \mathsf{type} \\ \Gamma \backslash x, (a' : (z : @\mathsf{N}) \multimap A[z/y]), (y : @\mathsf{N}) \vdash b : B[a'\, y/a] \qquad \Gamma \vdash a_x : A[x/y]\end{array}}{\Gamma \vdash \mathsf{ext}\, (\lambda\, a'\, y.\, b)\, x\, a_x : B[x/y, a_x/a]}\ \textsc{ext}$$

$$\frac{(x : @\mathsf{N}) \in \Gamma \qquad \Gamma \backslash x \vdash A\ \mathsf{type}}{\Gamma \vdash \mathsf{Gel}\, A\, x\ \mathsf{type}}\ \textsc{GelF} \qquad \frac{\Gamma, (x : @\mathsf{N}) \vdash g : \mathsf{Gel}\, A\, x}{\Gamma \vdash \mathsf{ung}\, (\lambda\, x.\, g) : A}\ \textsc{GelE} \qquad \frac{\Gamma \vdash a : A}{\mathsf{ung}\, (\lambda\, x.\, \mathsf{gel}\, a\, x) = a}\ \textsc{Gel}\beta$$

$$\frac{(x : @\mathsf{N}) \in \Gamma \qquad \Gamma \backslash x \vdash a : A}{\mathsf{gel}\, a\, x : \mathsf{Gel}\, a\, x}\ \textsc{GelI} \qquad \frac{\begin{array}{c}\textsc{GelF prem.} \qquad \Gamma \backslash x, (y : @\mathsf{N}) \vdash g_0, g_1 : \mathsf{Gel}\, A\, y \\ \Gamma \vdash \mathsf{ung}(\lambda y.\, g_0)\ =\ \mathsf{ung}(\lambda y.\, g_1) : A\end{array}}{\Gamma \vdash g_0[x/y] = g_1[x/y] : \mathsf{Gel}\, A\, x}\ \textsc{Gel}\eta$$

$$\frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathsf{Nm}\ \mathsf{type}}\ \mathsf{NmF} \qquad \frac{(x : @\mathsf{N}) \in \Gamma}{\Gamma \vdash \mathsf{c}\, x : \mathsf{Nm}}\ \mathsf{NmI}$$

$$\frac{\begin{array}{c}(x : @\mathsf{N}) \in \Gamma \qquad \Gamma \vdash n : \mathsf{Nm} \qquad \Gamma, z : \mathsf{Nm} \vdash B\ \mathsf{type} \qquad \Gamma \vdash b_0 : B[\mathsf{c}\, x/z] \\ \Gamma, (g : \mathsf{Gel}\, \mathsf{Nm}\, x) \vdash b_1 : B[\mathsf{forg}\, x\, g/z] \qquad \Gamma, (g : \mathsf{Gel}\, \mathsf{Nm}\, x) \vdash \mathsf{forg}\, x\, g := \mathsf{ext}\, (\lambda\, g'\, y.\, \mathsf{ung}\, g')\, x\, g : \mathsf{Nm}\end{array}}{\Gamma \vdash \mathsf{ind}_{\mathsf{Nm}}\, x\, n\, b_0\, (\lambda\, g.\, b_1) : B[n/z]}\ \mathsf{NmE}$$

$$\frac{\mathsf{NmE}\ \text{prem. without}\ n}{\Gamma \vdash \mathsf{ind}_{\mathsf{Nm}}\, x\, (\mathsf{c}\, x)\, b_0\, (\lambda\, g.\, b_1) = b_0 : B[\mathsf{c}\, x/y]}\ \mathsf{Nm}\beta_0 \qquad \frac{\mathsf{NmE}\ \text{prem. without}\ n \qquad \Gamma \backslash x \vdash n : \mathsf{Nm}}{\Gamma \vdash \mathsf{ind}_{\mathsf{Nm}}\, x\, n\, b_0\, (\lambda\, g.\, b_1) = b_1[\mathsf{gel}\, n\, x/g] : B[n/z]}\ \mathsf{Nm}\beta_1$$

Figure 2 Parametricity fragment of the CH theory [11] where we replace the arity by 0, and novel rules for the interval Nm. For binding rules such as EXT, we rely on the invertibility of both variable and name abstraction to bind using λ.

Nullary bridges The type former of dependent bridges with dependent codomain $\Gamma, (x : @\mathsf{N}) \vdash A$ is denoted by $(x : @\mathsf{N}) \multimap A\, x$. The type of non-dependent bridges with codomain $\Gamma \vdash A$ is written $@\mathsf{N} \multimap A$ and defined as $(- : @\mathsf{N}) \multimap A$. Introducing a bridge requires providing a term in a context extended with a bridge variable $(x : @\mathsf{N})$. A bridge $a' : (y : @\mathsf{N}) \multimap A\, y$ can be eliminated at a variable $(x : @\mathsf{N})$ only if x is fresh in a'. In summary, nullary bridges are treated and written like (dependent) functions out of the $@\mathsf{N}$ pretype but are restricted to consume fresh variables only. From a nominal point of view, a bridge $a' : @\mathsf{N} \multimap A$ is a name-abstracted value in A.

Since we will use the theory on non-trivial examples in the next sections, we prefer to explain the other primitives of Figure 2 from a user perspective: we derive programs in the empty context corresponding to the primitives of Figure 2 and express types as elements of the universe type $\mathcal{U}$, whose rules are not listed but standard. Furthermore we explain what the equations of Figure 2 entail for these closed programs. The closed programs derived from the CH nullary primitives have a binary counterpart in [34], an implementation of the CH binary PTT. The nullary and binary variants operate in a similar way. The pseudo-code we write uses syntax similar to Agda. We ignore writing universe levels and $\multimap$ is parsed like $\to$, e.g., it is right associative. We use ; to group several declarations in one line.

Lastly we indicate that the SAP holds at equality types $(a'_0\, a'_1 : @\mathsf{N} \multimap A) \to ((x : @\mathsf{N}) \multimap a'_0 x \equiv_A a'_1 x) \simeq a'_0 \equiv a'_1$. Proofs of this fact in the binary case can be found in [11, 34]. The nullary proof is obtained by erasing all mentions of (bridge) endpoints.

The extent primitive The rule for the extent primitive (EXT) together with the rules of $\multimap$ and $\mathcal{U}$ provide a term ext in the empty context with the following type.

```
ext : {A : @N ⊸ 𝒰} {B : (x:@N) ⊸ A x → 𝒰}
  (f' : (a' : (x:@N) ⊸ A x) → (x:@N) ⊸ B x (a' x)) →
  (x:@N) ⊸ (a : A x) → B x a
```

From a function f' mapping a bridge in A to a bridge in B, the extent primitive lets us build a bridge in the dependent function type formed from A and B, or $\Pi\, A\, B$ for short. Concisely put, the extent primitive validates one direction of the SAP at Π. It can be upgraded into the following equivalence, using the β-rule of extent, described below.

$$((a' : (x : @\mathsf{N}) \multimap A\, x) \to (x : @\mathsf{N}) \multimap B\, x\, (a'\, x)) \quad \simeq \quad (x : @\mathsf{N}) \multimap (a : A\, x) \to B\, x\, a$$

Proofs in the binary case can be found in [11, 34]. The nullary proof is obtained by erasing all mentions of endpoints. This equivalence entails $((@\mathsf{N} \multimap A) \to (@\mathsf{N} \multimap B)) \simeq @\mathsf{N} \multimap (A \to B)$ in the non-dependent case.

The β-rule of extent (EXTβ) is peculiar because a substituted premise $a[x/y]$ appears in the redex on the left-hand side. To compute the right-hand side out of the left-hand side only, the premise a is rebuilt (*) and a term where a variable is bound in a is returned. The step (*) is possible thanks to the restriction in the context $\Gamma \backslash x, (y : @\mathsf{N})$ of a. This is quite technical and explained in [11, 34]. We instead explain what that entails for our ext program above. Non-trivial examples of β-reductions for extent will appear in Section 2.2.

In a context Γ containing $(x : @\mathsf{N})$, The term $\Gamma \vdash \mathsf{ext}\, f'\, x\, a$ reduces if $\Gamma \vdash a$ is a term that does not mention cartesian variables strictly to the right of x in Γ, i.e. declared after x in Γ. In particular a can mention x. If such a freshness condition holds, x can in fact soundly be captured in a and the reduction can trigger. By default the reduction does not trigger because $f', x, a \vdash \mathsf{ext}\, f'\, x\, a$ does not satisfy the freshness condition as in this case a is a variable appearing to the right of x in the context.

Gel types Internally parametric type theories that feature interval-based Bridge types [17, 11] need a primitive to convert a n-ary relation of types into an n-ary bridge. In the CH theory the primitive is called Gel. The rules for Gel types together with the rules of $\multimap$ and $\mathcal{U}$ imply the existence of the following programs in the empty context. Note that ung is called ungel in the CH theory.

```
Gel : 𝒰 → @N ⊸ 𝒰
gel : {A : 𝒰} → A → (x : @N) ⊸ Gel A x
ung : {A : 𝒰} → ((x : @N) ⊸ Gel A x) → A
```

Similar to extent, Gel validates one direction of the SAP, this time at the universe $\mathcal{U}$. The Gel function can be upgraded into an equivalence $\mathcal{U} \simeq (@\mathsf{N} \multimap \mathcal{U})$ by using the rules of Gel and ext. In particular this requires proving that gel and ung are inverses, i.e. the SAP at Gel types $A \simeq (x : @\mathsf{N}) \multimap \mathsf{Gel}\, A\, x$. Again these theorems are proved in [11, 34] in the binary case, and the nullary proofs are obtained by erasing endpoints.

From a nominal perspective, we observe that Gel is a type former that can be used to express freshness information. This can be seen by looking at the GELI rule: canonical inhabitants of $\mathsf{Gel}\, A\, x$ are equivalently terms $a : A$ such that x is fresh in a. Conversely the ung primitive is a binder that makes available a fresh variable x when a value of type A is being defined, as long as the result value is typally fresh w.r.t. x. The ung primitive can also be used to define a map forgetting freshness information.

```
forg : {A : 𝒰} → (x:@N) ⊸ Gel A x → A
forg {A} = ext [λ(g':(x:@N) ⊸ Gel A x).λ(_:@N). ung g']
```

Lastly, GELη can be used if a certain freshness side condition holds, similar to EXTβ. We will not need to make explicit use of GELη.

2.2 The Nm type, name induction, nominal data types

So far the rules we presented involved occurrences of the bridge interval morally to the left of $\vdash$, as affine comprehensions. Since we wish to use nullary PTT as a nominal framework, we need a first-class type of names $\vdash \mathsf{Nm}\,\mathsf{type}$. Indeed this is needed to express nominal data types, whose constructors may take names as arguments. For example the var constructor of the Ltm nominal data type declared in Section 1 has type $\mathsf{var} : \mathsf{Nm} \to \mathsf{Ltm}$, a regular "cartesian" function type.

Constructors must be cartesian functions because that is what the initial algebra semantics of (even nominal) data types dictates. In other words it is unclear whether there exists a sound notion of data types D with "constructors" of the form $@\mathsf{N} \multimap D$ for instance[2]. Moreover the fact that nominal constructors use standard functions out of Nm is one of the main ingredients required to recover nominal pattern matching as we shall see.

Another important ingredient is the SAP at the type of names Nm which reads $1 + \mathsf{Nm} \simeq (@\mathsf{N} \multimap \mathsf{Nm})$. The principle expresses that a bridge at the type of names Nm is either the "identity" bridge or a constant bridge. This principle is proved based on our rules for Nm which we explain now. The rules of Nm together with the other rules of Figure 2 entail the existence of the following programs in the empty context.

```
Nm : 𝒰
c : @N ⊸ Nm
ind_Nm : {B : Nm → 𝒰} →
  (x:@N) ⊸ (n:Nm) →
  (b0 : B (c x)) → (b1 : (g : Gel Nm x) → B (forg x g)) → B n
```

The NmI rule expresses that the canonical inhabitants of Nm in a context Γ are simply the affine variables $(x : @\mathsf{N})$ appearing in Γ. The "identity" bridge program c above is derived from that rule. In simple terms, c coerces affine bridge variables x into names $\mathsf{c}\,x : \mathsf{Nm}$.

Name induction The $\mathsf{ind}_{\mathsf{Nm}}$ program/rule is an induction principle, or dependent eliminator for the type Nm. It expresses that in a context Γ containing an affine variable $(x : @\mathsf{N})$ we can do a case analysis on a term $\Gamma \vdash n : \mathsf{Nm}$. Either x is bound in n and n is in fact exactly $c\,x$, or x is *fresh* in n. The call $\mathsf{ind}_{\mathsf{Nm}}\,x\,n\,b_0\,b_1$ returns b_0 if $n = \mathsf{c}\,x$ (see rule $\mathsf{Nm}\beta_0$) and returns $b_1\,(\mathsf{gel}\,n\,x)$ if x is fresh in n (see rule $\mathsf{Nm}\beta_1$). The freshness assumption in b_1 is expressed typally using a Gel type.

We show that the rules $\mathsf{Nm}\beta_0$ and $\mathsf{Nm}\beta_1$ are well typed, i.e. that the $\mathsf{ind}_{\mathsf{Nm}}$ program above reduces to something of type $B\,n$ in both scenarios. If $n = \mathsf{c}\,x$ then the reduction result is $\mathsf{ind}_{\mathsf{Nm}}\,x\,(\mathsf{c}\,x)\,b_0\,b_1 = b_0 : B\,(\mathsf{c}\,x)$ and $B(\mathsf{c}\,x) = B\,n$. Else if x is fresh in n then $\mathsf{gel}\,n\,x$ typechecks and the reduction result is $\mathsf{ind}_{\mathsf{Nm}}\,x\,n\,b_0\,b_1 = b_1\,(\mathsf{gel}\,n\,x) : B(\mathsf{forg}\,x\,(\mathsf{gel}\,n\,x))$ where forg is the function defined in Section 2.1 So we need to prove that $\mathsf{forg}\,x\,(\mathsf{gel}\,n\,x) = n$. Note

[2] If the arity is 2, the path analogue of such a notion does exist: higher inductive types (HITs) [32].

that forg is defined as ext applied to a function of type $((x : @\mathsf{N}) \multimap \mathsf{Gel}\,\mathsf{Nm}\,x) \to @\mathsf{N} \multimap \mathsf{Nm}$.

$$\begin{aligned}
\mathsf{forg}\,x\,(\mathsf{gel}\,n\,x) &= \mathsf{ext}\,[\lambda(g' : (x : @\mathsf{N}) \multimap \mathsf{Gel}\,\mathsf{Nm}\,x).\lambda(_ : @\mathsf{N}).\mathsf{ung}\,g']\,x\,(\mathsf{gel}\,n\,x) && \text{(def.)}\\
&= (\lambda(_ : @\mathsf{N}).\,\mathsf{ung}\,(\lambda y.\,\mathsf{gel}\,n\,y))\,x && (\textsc{ext}\beta)\\
&= \mathsf{ung}\,(\lambda y.\,\mathsf{gel}\,n\,y) && (\multimap \beta)\\
&= n && (\textsc{Gel}\beta)
\end{aligned}$$

The $\textsc{ext}\beta$ rule triggers because (1) by assumption, x is fresh in n, i.e. n does not mention x, nor cartesian variables strictly to the right of x (2) thus the term $\mathsf{gel}\,n\,x$ mentions x but no cartesian variables declared later. The latter condition is exactly the condition under which this ext call can reduce. To that end, x is captured in the term $\mathsf{gel}\,n\,x$ leading to a term $g' := \lambda y.\,\mathsf{gel}\,n\,y$ appearing on the second line.

SAP at Nm We now prove that $1 + \mathsf{Nm} \simeq @\mathsf{N} \multimap \mathsf{Nm}$. We take inspiration from [11] who developed a relational encode-decode technique to prove the SAP at data types. The type Nm is defined existentially, i.e. by an induction principle, and it turns out that a similar technique can be applied.

```
loosen : 1 + Nm → @N ⊸ Nm
loosen (inl _) = λx. c x    ;    loosen (inr n) = λ_. c n

t1 : (@N ⊸ Nm) → (x:@N) ⊸ 1 + Gel Nm x
t1 n' x = ind_Nm x (n' x) (inl _) (λ (g:Gel Nm x). inr g)

t2pre : (x:@N) ⊸ (1 + Gel Nm x) → Gel (1 + Nm) x
t2pre x (inl _) = gel (inl _) x
t2pre x (inr g) = ext [λg'. λy. gel (inr (ung g')) y] x g

t2 : ((x:@N) ⊸ 1 + Gel Nm x) → (x:@N) ⊸ Gel (1 + Nm) x
t2 = λ s' x. t2pre x (s' x)

tighten : (@N ⊸ Nm) → 1 + Nm
tighten n' = ung (t2 (t1 n'))
```

It remains to prove the roundtrip equalities. The roundtrip for $s : 1 + \mathsf{Nm}$ is obtained by induction on s. If $s = \mathsf{inl}-$ then $(\mathsf{ung} \circ t_2 \circ t_1 \circ \mathsf{loosen})\,s = (\mathsf{ung} \circ t_2 \circ t_1)(\lambda x.\,\mathsf{c}\,x) \overset{\mathsf{Nm}\beta_0}{=} (\mathsf{ung} \circ t_2)(\lambda x.\,\mathsf{inl}-) = \mathsf{ung}\,(\lambda x.\,\mathsf{gel}\,(\mathsf{inl}-)\,x) = s$. If $s = \mathsf{inr}\,n$ then $(\mathsf{ung} \circ t_2 \circ t_1 \circ \mathsf{loosen})\,s = (\mathsf{ung} \circ t_2 \circ t_1)(\lambda x.\,n) \overset{\mathsf{Nm}\beta_1}{=} (\mathsf{ung} \circ t_2)(\lambda x.\,\mathsf{inr}(\mathsf{gel}\,n\,x)) = \mathsf{ung}\,(\lambda x.\,\mathsf{ext}\,[\lambda g' y.\,\mathsf{gel}\,(\mathsf{inr}(\mathsf{ung}\,g'))\,y]\,x\,(\mathsf{gel}\,n\,x)) \overset{\textsc{ext}\beta}{=} \mathsf{ung}\,(\lambda x.\,\mathsf{gel}\,(\mathsf{inr}\,n) x) = \mathsf{inr}\,n$.

For the other roundtrip we first give a sufficient condition. It is the first type of this chain of equivalences and can be understood as a propositional η-rule for Nm.

$$\begin{aligned}
(x : @\mathsf{N}) \multimap (n : \mathsf{Nm}) \to (n \equiv_{\mathsf{Nm}} \mathsf{ext}\,[\mathsf{loosen} \circ \mathsf{tighten}]\,x\,n) &\simeq \\
(n' : @\mathsf{N} \multimap \mathsf{Nm}) \to (x : @\mathsf{N}) \multimap (n' x \equiv_{\mathsf{Nm}} (\mathsf{loosen} \circ \mathsf{tighten})\,n'\,x) &\simeq && (\mathrm{SAP}_{\to})\\
(n' : @\mathsf{N} \multimap \mathsf{Nm}) \to n' \equiv (\mathsf{loosen} \circ \mathsf{tighten}) n' & && (\mathrm{SAP}_{\equiv})
\end{aligned}$$

The ext in the first line vanishes in the second because $\textsc{ext}\beta$ triggers. We prove the sufficient condition. Let $(x : @\mathsf{N}), (n : \mathsf{Nm})$ in context. We reason by name induction on n. If $n = \mathsf{c}\,x$ then the right-hand side is $\mathsf{ext}\,[\mathsf{loosen} \circ \mathsf{tighten}]\,x\,(\mathsf{c}\,x) \overset{\textsc{ext}\beta}{=} ((\mathsf{loosen} \circ \mathsf{tighten})\,\mathsf{c})\,x$. From the proof above we know that $\mathsf{tighten}\,\mathsf{c} = \mathsf{inl}-$, and by definition $\mathsf{loosen}\,(\mathsf{inl}-) = \mathsf{c}$. So both sides of the equality are equal to $\mathsf{c}\,x$. If x is fresh in n then the right-hand side is

Parameters	$(x : \mathsf{@N}) \multimap$	K	$\simeq$	K'
$A : \mathsf{@N} \multimap \mathcal{U}$, $B : (x : \mathsf{@N}) \multimap A\ x \ \to \mathcal{U}$.	$(x : \mathsf{@N}) \multimap$	$(a : A\ x) \to B\ x\ a$	$\simeq$	$(a' : (x : \mathsf{@N}) \multimap A\,x) \to (x : \mathsf{@N}) \multimap B\,x\,(a'\,x)$
	$(x : \mathsf{@N}) \multimap$	$(a : A\ x) \times (B\ x\ a)$	$\simeq$	$(a' : (x : \mathsf{@N}) \multimap A\ x) \times ((x : \mathsf{@N}) \multimap B\ x\ (a'\ x))$
	$(x : \mathsf{@N}) \multimap$	$\mathcal{U}$	$\simeq$	$\mathcal{U}$
$A : \mathcal{U}, a_0, a_1 : \mathsf{@N} \multimap A$.	$(x : \mathsf{@N}) \multimap$	$a_0\ x \equiv a_1\ x$	$\simeq$	$a_0 \equiv_{\mathsf{@N} \multimap A} a_1$
	$(x : \mathsf{@N}) \multimap$	Nm	$\simeq$	$1 + \mathsf{Nm}$
$A : \mathcal{U}$	$(x : \mathsf{@N}) \multimap$	$\mathsf{Gel}\ A\,x$	$\simeq$	A
$A : (x : \mathsf{@N}) \multimap (y : \mathsf{@N}) \multimap \mathcal{U}$	$(x : \mathsf{@N}) \multimap$	$(y : \mathsf{@N}) \multimap A\,x\,y$	$\simeq$	$(y : \mathsf{@N}) \multimap (x : \mathsf{@N}) \multimap A\,x\,y$

Table 1 The Structure Abstraction Principle

$\mathsf{ext}\,[\mathsf{loosen} \circ \mathsf{tighten}]\,x\,n \overset{\text{EXT}\beta}{=} (\mathsf{loosen} \circ \mathsf{tighten})(\lambda -.\,n)x$. By the proof above $\mathsf{tighten}\,(\lambda -.\,n) = \mathsf{inr}\,n$ and by definition $\mathsf{loosen}\,(\mathsf{inr}\,n) = \lambda -.\,n$. Thus both sides are equal to $n : \mathsf{Nm}$.

Nominal data types When looking at a specific example of a nominal data type D we temporarily extend the type system with the rules of D. This includes the dependent eliminator of D. For example in the case of the Ltm type of Section 1 we add a rule that entails the existence of this closed program:

```
indLtm : (P : Ltm → 𝒰) ((n : Nm) → P(var n)) (∀a b. P(app a b)) →
  (g : @N ⊸ Ltm)((x:@N) ⊸ P(g x)) → ∀t.P t
```

2.3 The Structure Abstraction Principle (SAP)

As hinted above, the primitives of our nullary PTT allow us to prove the Structure Abstraction Principle (SAP). Similar to the SIP of HOTT/UF, or the SRP of binary PTT [34], the SAP defines how (nullary) Bridge types commute with other type formers. Table 1 lists several SAP instances, including the ones we have encountered so far. Each instance is of the form $\forall \cdots.((x : \mathsf{@N}) \multimap K) \equiv K'$, where K may depend on some terms and $(x : \mathsf{@N})$. Note that the instances in Table 1 involve primitive type formers K. In order to obtain the SAP instance of a composite type K, we can combine the SAP instances of the primitives used to define K. This was done e.g. in Section 2.2 when proving the propositional η-rule of Nm. A larger example will appear e.g. in Theorem 1.

For a type K (potentially dependent, composite), the SAP provides a type[3] $K' \simeq \mathsf{@N} \multimap K$ which we will refer to as (1) the observational parametricity of K or (2) the nullary parametricity translation of K. Types of DTT can contain terms and thus there exists a SAP for terms as well. The basic idea is simple. Any term $k : K$ induces a reflexivity bridge $\lambda(\_ : \mathsf{@N}).k : \mathsf{@N} \multimap K$, and the SAP for K says $\mathsf{SAP}_K : K' \simeq \mathsf{@N} \multimap K$. K' is (typically) the recursive translation $[K]_0$ of the type K and the translation is defined in such a way that $[k]_0 : [K]_0$. Now, The SAP for $k : K$ asserts that the reflexivity bridge of k and its translation agree up to $\equiv$, i.e. $\mathsf{SAP}_K^{-1}(\lambda\_.\,k) \equiv [k]_0$. For that reason, we define and write the

[3] With this direction $\mathsf{SAP}_{\mathcal{U}} = \mathsf{Gel}$ and $\mathsf{SAP}_{\Pi} = \mathsf{ext}$.

observational parametricity of a term to be $[k]_0 := \mathsf{SAP}_K^{-1}(\lambda\_.\, k)$. Note that this definition depends on the exact choice of K' and equivalence SAP_K.

For example suppose A and B are types with SAP instances $A' \simeq (@\mathsf{N} \multimap A)$ and $B' \simeq (@\mathsf{N} \multimap B)$. The SAP at $A \to B$ is the composition $\mathsf{SAP}_{A\to B}^{-1} : (@\mathsf{N} \multimap (A \to B)) \simeq ((@\mathsf{N} \multimap A) \to @\mathsf{N} \multimap B) \simeq A' \to B'$. The map $\mathsf{SAP}_{A\to B}^{-1}$ is given by $\lambda f'.\, \mathsf{SAP}_B^{-1} \circ (\mathsf{ext}^{-1}\, f') \circ \mathsf{SAP}_A$ where $\mathsf{ext}^{-1} f' = \lambda(a' : @\mathsf{N} \multimap A).\, \lambda(x : @\mathsf{N}).\, f'\, x\, (a'\, x)$. With this choice of SAP instances, the observational parametricity of a function $f : A \to B$ has type $[f]_0 : A' \to B'$ and unfolds to $[f]_0 = \mathsf{SAP}_B^{-1} \circ (@\mathsf{N} \multimap f) \circ \mathsf{SAP}_A$ where $(@\mathsf{N} \multimap f) = \lambda a'.\, f\,(a' x)$ is the action of f on bridges. So the SAP for a function $f : A \to B$ asserts that its action on bridges agrees with its translation, up to the SAP at A, B.

The SAP of a composite type K is obtained by manually applying the SAP rules from Table 1. This process will produce an observational parametricity/parametricity translation K' that is equivalent to $@\mathsf{N} \multimap K$. Although we suggestively use the notation $[-]_0$, the actual function $[-]_0$ which maps *every* type/term to its recursively defined translation, does not exist in our system and we manually define K' on a case-by-case basis. In the binary case, Van Muylder et al. [34] have shown that this process can be systematized, by organizing types and terms together with their SRP instances into a (shallowly embedded) type theory called ROTT. For types K and terms $k : K$ that fall in the ROTT syntax, the parametricity translations $[K]_0$,$[k]_0$ and the SAP equivalences $[K]_0 \simeq @\mathsf{N} \multimap K$ and $\mathsf{SAP}_K^{-1}(\lambda\_.\, k) \equiv [k]_0$ can be derived automatically. Although we believe the same process applies here, we have not constructed the corresponding DSL, instead applying SAP instances manually in examples.

2.4 Semantics, Soundness and Computation

We follow Cavallo and Harper [11] in modeling internal parametricity in cubical type theory in presheaves over the product of two base categories: the binary cartesian cube category $\boxslash_2$ for path dimensions [4] and the n-ary affine cube category $\square_n$ for bridge dimensions. For nullary PTT, $\square_0$ is a category whose objects are finite ordinals (sets of names) and whose morphisms $\varphi : V \to W$ are injections $-[\varphi] : W \hookrightarrow V$.

We note that $\square_0$ is the base category of the Schanuel sheaf topos [26, §6.3] which is equivalent to the category of nominal sets [26] used to model FreshMLTT [27]. The sheaf-condition requires that presheaves $\Gamma : \square_0^{\mathsf{op}} \to \mathrm{Set}$ preserve pullbacks, i.e. compatible triples in $\Gamma_{U \uplus V} \to \Gamma_{U \uplus V \uplus W} \leftarrow \Gamma_{U \uplus W}$ have a unique preimage in Γ_U. Conceptually, if a cell $\gamma \in \Gamma_{U \uplus V \uplus W}$ is both fresh for V and for W, then it is fresh for $V \uplus W$, with unique evidence. This property is not available internally in nullary PTT, nor do we have the impression that it is important to add it, so we content ourselves by modeling types as *pre*sheaves over $\square_0$. It is worth noting that the property holds for any presheaf over $\square_{n>0}$, so that both $\mathrm{Psh}(\square_0)$ and the Schanuel topos are legitimate nullary analogues of $\mathrm{Psh}(\square_{n>0})$.

The semantics of the primitives and inference rules of nullary CH are then straightforwardly adapted (and often simplified) to our model.

The type Nm is interpreted as the Yoneda-embedding of the base object with 1 name and no path dimensions. The elimination and computation rules for this type are based on a semantic isomorphism $x : @\mathsf{N} \vdash \mathsf{Nm} \cong \top \uplus \mathsf{Gel}\,\mathsf{Nm}\, x$ which is straightforwardly checked. Kan fibrancy of this type is semantically trivial, since we can tell from the base category that any path in Nm will be constant. As for computation, we propose to wait until all arguments to the Kan operation reduce to $\mathsf{c}\, x$ for the same affine name x, in which case we return $\mathsf{c}\, x$. This is in line with how Kan operations for positive types with multiple constructors are usually reduced [13, 4]. Regarding nominal data types, which we only consider in an example-based fashion in this paper, we take the viewpoint that these arise from an interplay between the usual type formers, bridge types, Nm, and an initial algebra operation for 'nominal strictly positive functors'. We do not attempt to give a categorical description of such functors or prove that they have initial algebras. The Kan operation will reduce recursively and according to the Kan operations of all other type formers involved.

3 Nominal Primitives for Free

In this section, we argue that the central features in a number of earlier nominal type systems, can essentially be recovered in nullary PTT. The case of nominal pattern matching is treated separately, in the next section. Concretely, we consider Shinwell, Pitts and Gabbay's FreshML [31], Schöpp and Stark's bunched nominal type theory which we shall refer to here as BNTT [30, 29], Cheney's $\lambda^{\Pi И}$ [12] and Pitts, Matthiesen and Derikx's FreshMLTT [27]. The central features we identify there, are: existential and universal name quantification, a type former expressing freshness for a given name, name swapping, and locally scoped names. Existential and universal name quantification are known to be equivalent in the usual (pre)sheaf or nominal set semantics of nominal type theory, but generally have quite different typing rules: the former is an existential type former with pair-like constructor and matching eliminator (opening the door to matching more deeply), whereas the latter is a universal type operated through name abstraction and application (getting in the way of matching more deeply). We note that some systems (FreshMLTT, $\lambda^{\Pi И}$) support multiple name types, something we could also easily accommodate but leave out so as not to distract from the main contributions (but see Section 5). BNTT even allows substructural quantification and typal freshness for arbitrary closed types (rather than just names), which is not something we intend to support and which inherently seems to require a bunched context structure. In what follows, we will speak of 'our rules', not to claim ownership (as they are inherited from Bernardy, Coquand and Moulin [9] and Cavallo and Harper [11]), but to distinguish with the other systems.

Universal name quantification Universal name quantification is available in BNTT, $\lambda^{\Pi И}$ and FreshMLTT. In our system, it is done using the nullary bridge type $(x : @\mathsf{N}) \multimap A$, whose rules are given in Figure 2. The rules $\multimap$F and $\multimap$I correspond almost perfectly with the other three systems; for BNTT we need to keep in mind that our context extension with a fresh name, is semantically a monoidal product. The application rules in $\lambda^{\Pi И}$ and BNTT also correspond almost precisely to $\multimap$E, but the one in BNTT is less algorithmic than ours. Specifically, the BNTT bunched application rule takes a function $\Theta \vdash f : (x : T) \multimap A\,x$ and an argument $\Delta \vdash t : T$ and produces $f\,t : A\,t$ in a non-general context $\Theta * \Delta$. Our rule $\multimap$E (inherited from [9, 11]) improves upon this by taking in an arbitrary context Γ and *computing* a context $\Gamma\backslash x$ such that there is a morphism $\Gamma \to (\Gamma\backslash x, (x : @\mathsf{N}))$. The application rule in FreshMLTT is similar, but uses definitional freshness – based on a variable swapping test – to ensure that the argument is fresh for the function.

Typal freshness A type former expressing freshness is available in BNTT (called the free-from type). FreshMLTT uses definitional freshness instead. In nullary PTT, elements of A that are fresh for x are classified by the type $\mathsf{Gel}\,A\,x$. BNTT's free-from types only apply to closed types A, so GelF is evidently more general. BNTT's introduction rule corresponds to GelI, which is however again more algorithmic. BNTT's elimination rule is explained in terms of single-hole bunched contexts [29, §4.1.1] which specialize in our setting (where the only monoidal product is context extension with a name) to contexts with a hole up front. Essentially then, the rule can be phrased in nullary PTT as

$$\frac{\Gamma, x : @\mathsf{N}, z : \mathsf{Gel}\, B\, x, \Theta \vdash T\ \mathsf{type} \qquad \Gamma, y : B, x : @\mathsf{N}, \Theta[\mathsf{gel}\, y\, x/z] \vdash t : T[\mathsf{gel}\, y\, x/z]}{\Gamma, x : @\mathsf{N}, z : \mathsf{Gel}\, B\, x, \Theta \vdash t' : T}$$

where Γ must be empty, together with β- and η-rules establishing that the above operation is inverse to applying the substitution $[\mathsf{gel}\, y\, x/z]$[4]. We can in fact accommodate the rule for non-empty Γ. Without loss of generality, we can assume Θ is empty: by abstraction/application, we can subsume Θ in T[5]. We then have an equivalence

$$\begin{aligned} ((y : B) \to ((x : @\mathsf{N}) \multimap T[\mathsf{gel}\, y\, x/z])) \quad &\simeq \quad (z' : (x : @\mathsf{N}) \multimap \mathsf{Gel}\, B\, x) \to ((x : @\mathsf{N}) \multimap T[z'\, x/z]) \\ &\simeq \quad (x : @\mathsf{N}) \multimap (z : \mathsf{Gel}\, B\, x) \to T \end{aligned}$$

where in the first step, we precompose with the $\mathsf{gel}/\mathsf{ung}$ isomorphism (the SAP for Gel), and in the second step, we apply the ext equivalence (the SAP for functions).

Existential name quantification Existential name quantification is available in FreshML and BNTT. We first discuss how we can accommodate the BNTT rules, and then get back to FreshML. The BNTT existential quantifier is just translated to the nullary bridge type $(x : @\mathsf{N}) \multimap A$ again, i.e. both quantifiers become definitionally the same type in nullary PTT. BNTT's introduction rule follows by applying a function

$$\mathsf{bind} : (x : @\mathsf{N}) \multimap B\, x \to \mathsf{Gel}\, ((w : @\mathsf{N}) \multimap B\, w)\, x,$$

which is obtained from the identity function on $(w : @\mathsf{N}) \multimap B\, w$ by the SAP for functions and Gel. BNTT's elimination rule essentially provides a function

$$\mathsf{matchbind} : ((x : @\mathsf{N}) \multimap B\, x \to \mathsf{Gel}\, C\, x) \to (((x : @\mathsf{N}) \multimap B\, x) \to C)$$

such that if we apply $\mathsf{matchbind}\, f$ under Gel to $\mathsf{bind}\, x\, b$, then we obtain $f\, x\, b$. Again, the SAP for Gel and functions reveals that the source and target of $\mathsf{matchbind}$ are equivalent.

The non-dependently typed system FreshML has a similar type former, but lacks any typal or definitional notion of freshness. Their introduction rule then follows by postcomposing the above bind with the function forg that forgets freshness (Section 2.1). If we translate FreshML's declarations $\Gamma \vdash d : \Delta$ to operations that convert terms $\Gamma, \Delta \vdash t : T$ to terms $\Gamma \vdash t\{d\}$ (not necessarily by substitution), then we can translate their declaration $\Gamma \vdash \mathtt{val}\, \langle x \rangle y = e : (x : @\mathsf{N}, y : B)$, where e is an existential pair, as $\mathsf{matchdecl}\, e$ where

$$\mathsf{matchdecl} : (@\mathsf{N} \multimap B) \to (@\mathsf{N} \multimap B \to T) \to (@\mathsf{N} \multimap T).$$

This function is obtained by observing that the second argument type, by the SAP for functions, is equivalent to $(@\mathsf{N} \multimap B) \to (@\mathsf{N} \multimap T)$. It may be surprising that the result has type $@\mathsf{N} \multimap T$ rather than just T; this reflects the fact that FreshML cannot enforce freshness, and is justified by the fact that it allows arbitrary declaration of fresh names via the declaration $\Gamma \vdash \mathtt{fresh}\, x : (x : @\mathsf{N})$, which we do not support.

Swapping names The name swapping operation is available in FreshML and FreshMLTT. We can accommodate the full rule of FreshML (which is not dependently typed) and a restricted version of the rule in FreshMLTT, where we allow the type of the affected term

[4] The original rule immediately subsumes a substitution $\Delta \to (x : @\mathsf{N}, z : \mathsf{Gel}\, A\, x)$.
[5] This may raise questions about preservation of substitution w.r.t. Θ. However, we are unaware of any non-bunched dependent type systems that assert preservation of substitution w.r.t. a part of the context that is dependent on one of the premises of an inference rule.

to depend on the names being swapped, but other than that, only on variables fresh for those names. We simply use the function $\mathsf{swap} : (x\,y : @\mathsf{N}) \multimap T\,x\,y \to T\,y\,x$ whose type is equivalent to $((x\,y : @\mathsf{N}) \multimap T\,x\,y) \to ((x\,y : @\mathsf{N}) \multimap T\,y\,x)$ by the SAP, and the latter is clearly inhabited by $\lambda\,t\,x\,y.\,t\,y\,x$. Finally, we note that the aforementioned restriction on the type can be mitigated in an ad hoc manner, because types $T : (x\,y : @\mathsf{N}) \multimap \Delta \to \mathcal{U}$ (where Δ denotes any telescope consisting of both affine names and non-affine variables) are by the SAP in correspondence with types $\Delta'' \to (x\,y : @\mathsf{N}) \multimap \mathcal{U}$ for a different telescope Δ''. This however does not imply that we can accommodate the full FreshMLTT name swapping rule in a manner that commutes with substitution. We expect that this situation can be improved by integrating ideas related to the transpension type [20, 18] into the current system.

Using swap, we can follow Pitts et al. [27] in defining non-binding abstraction $\langle x\rangle - = \lambda\,a\,y.\,\mathsf{swap}\,x\,y\,a : A\,x \to (y : @\mathsf{N}) \multimap A\,y$, where A can depend only on variables fresh for x.

Locally scoped names Locally scoped names [22, 25] are available in FreshMLTT, by the following rule on the left, and allow us to spawn a name from nowhere, provided that we use it to form a term that is fresh for it:

$$\frac{\Gamma, x : @\mathsf{N} \vdash T\;\mathsf{type} \quad \Gamma, x : @\mathsf{N} \vdash t : T \quad x \text{ is fresh for } t : T}{\Gamma \vdash \nu\,x.\,t : \nu\,x.\,T} \qquad \begin{array}{l}\Gamma, x : @\mathsf{N} \vdash \nu\,x.\,t = t : T \\ \text{Added: } \nu\,x.\,t = t \text{ if } x \text{ is not free in } t.\end{array} \qquad \frac{\Gamma \vdash S\;\mathsf{type} \quad \Gamma, x : @\mathsf{N} \vdash t : S \quad x \text{ is fresh for } t : T}{\Gamma \vdash \nu\,x.\,t : S}$$

It is used ([23]) in FreshMLTT to define e.g.

$$\lambda\,c'.\,\nu\,x.\,\mathsf{case}\,c'\,x \begin{cases} \mathsf{inl}\,a \mapsto \mathsf{inl}\,\langle x\rangle a \\ \mathsf{inr}\,b \mapsto \mathsf{inr}\,\langle x\rangle b \end{cases} : (@\mathsf{N} \multimap A + B) \to (@\mathsf{N} \multimap A) + (@\mathsf{N} \multimap B)$$

It has a computation rule which says that as soon as x comes into scope again, we can drop the ν-binder. Combined with α-renaming, this means $\nu\,x.$ says 'let x be any name we have in scope, or a fresh one, it doesn't matter'. In particular, $\nu\,x.-$ is idempotent. Before translating to nullary PTT, we add another equation rule, which says that we can drop $\nu\,x.-$ if x is not used freely at all (in the example above, x is used freely but freshly). This way, the FreshMLTT ν-rule above becomes equivalent to the one to its right. Indeed, the functions $T \mapsto \nu\,x.\,T$ and $S \mapsto S$ now constitute an isomorphism between types that are and are not dependent on $x : @\mathsf{N}$. The advantage of the rule on the right is that it is not self-dependent. In nullary PTT, we express freshness using Gel, suggesting the following adapted rules:

$$\frac{\Gamma, x : @\mathsf{N} \vdash t : \mathsf{Gel}\,S\,x}{\Gamma \vdash \nu\,x.\,t : S} \qquad \begin{array}{l}\Gamma, x : @\mathsf{N} \vdash \mathsf{gel}\,(\nu\,x.\,t)\,x = t : \mathsf{Gel}\,S\,x \\ \nu\,x.\,\mathsf{gel}\,t\,x = t \text{ (where } x \text{ cannot be free in } t \text{ by } \textsc{GelI}).\end{array}$$

In this formulation, it is now clear that $\nu x.\,t$ can be implemented as $\mathsf{ung}\ t$, while the two computation rules follow from $\textsc{Gel}\eta$ and $\textsc{Gel}\beta$.

4 Nominal Pattern Matching

In this section we provide concrete examples of functions $D \to E$ defined by recursion on a nominal data type D, within nullary PTT as introduced in Section 2.

4.1 Patterns that bind

Some nominal frameworks with existential name-abstraction types [31] provide a convenient user interface to define functions $f : D \to E$ out of a nominal data type. The user can define

f by matching on patterns that bind names (see eqabs in Section 1). We explain how nullary parametricity lets us informally recover this feature in our system.

The following nominal data type is the nominal syntax of the π-calculus [16]. This data type appears in [8]. The constructors stand for: terminate, silent computation step, parallelism, non-determinism, channel allocation, receiving[6], and sending.

```
data Proc : 𝓤 where
  nil : Proc
  τpre : Proc → Proc
  par, sum : Proc → Proc → Proc
```

```
  nu : (@N ⊸ Proc) → Proc
  inp : Nm → (@N ⊸ Proc) → Proc
  out : Nm → Nm → Proc → Proc
```

The π-calculus [16] is a formal language whose expressions represent concurrently communicating processes. An example of a process (in an appropriate context) is $\mathsf{par}\,(\mathsf{out}\, a\, b\, q)\,(\mathsf{inp}\, a\,(\lambda(x : @\mathsf{N}).p'\,x))$. The first argument of par emits a name b on channel a and continues with q. Simultaneously, the second argument waits for a name x on channel a and continues with $p'\,x$. The expectation is that such an expression should reduce to $\mathsf{par}\,q\,(p'\{b/x\})$ where $p'\{b/x\}$ replaces occurrences of $(x : @\mathsf{N})$ in the body of p' by the name $b : \mathsf{Nm}$. Note that $p\,b$ does not typecheck and that this substitution operation called nsub must be defined recursively, as done in [8] and here. The following is an informal definition of nsub where some patterns bind (bridge) variables.

```
nsub : Nm → (@N ⊸ Proc) → Proc
nsub b (λx. par (u' x) (v' x)) = par (nsub b u') (nsub b v')
... --nil, τpre, sum similar
nsub b (λx. (nu (q' x))) = nu (λy. nsub b (λx. q' x y))
nsub b (λx. inp a  (q' x)) = inp a (λy. nsub b (λx. q' x y)) --(0)
nsub b (λx. inp (c x)  (q' x)) = inp b (λy. nsub b (λx. q' x y)) --(1)
...
```

Note how patterns (0) and (1) match on the same term constructor $\mathsf{inp}\; m\; q'$, but cover the case where m is different resp. equal to the variable being substituted. The informal nsub function reduces accordingly.

More formally in our system we define nsub by using the SAP at Proc, so $\mathsf{nsub}\,b$ is the following composition $(@\mathsf{N} \multimap \mathsf{Proc}) \xrightarrow{\simeq} \mathsf{AProc} \longrightarrow \mathsf{Proc}$. The SAP at Proc asserts that $(@\mathsf{N} \multimap \mathsf{Proc})$ is equivalent to AProc, the nullary translation of Proc:

```
data AProc : 𝓤 where
  nil, τpre, par, sum, nu : ...  --similar
  inp0 : Nm → (@N ⊸ AProc) → AProc
  inp1 : (@N ⊸ AProc) → AProc
  out00 : Nm → Nm → AProc → AProc
  out01 , out10 : Nm → AProc → AProc
  out11 : AProc → AProc
```

We can then define $\mathsf{nsub}' : \mathsf{Nm} \to \mathsf{AProc} \to \mathsf{Proc}$ by induction on the second argument. Then the informal clauses (0), (1) can be translated into formal ones using inp0, inp1, respectively.

[6] The reader might wonder if we could instead bind a cartesian name in the second argument of inp. However, this would lead to exotic process terms which can check the bound name for equality to other names.

4.2 A HOAS Example by term-relevant parametricity

Let D be a nominal data type. This example illustrates the fact that defining and proving correct a function $f : D \to E$ often requires (1) the SAP at E and (2) to compute the translation of the term $f : D \to E$.

We connect the nominal syntax of the untyped lambda calculus (ULC) to a higher-order abstract syntax (HOAS) representation. The nominal syntax of ULC is expressed as the following data type family Ltm, parametrized by a natural number j : `nat`.

```
data Ltm (j : nat) : 𝒰
  holes : Fin j → Ltm j
  var : Nm → Ltm j
  app : Ltm j → Ltm j → Ltm j
  lam : (@N ⊸ Ltm j) → Ltm j
```

The type `Fin j` is the finite type with j elements $\{0, \ldots, j-1\}$. Ltm_j is a shorthand for $\mathsf{Ltm}\, j$. The types Ltm and Ltm_1 of Section 1 are Ltm_0 and Ltm_1 respectively.

The corresponding HOAS representation, or encoding, is $\mathsf{HEnc}_j : \mathcal{U}$ defined below. Since it uses a Π-type we say it is a Π-encoding (there are other ways to define and use HOAS not discussed here).

```
HMod (j : nat) = (H : 𝒰) × (Fin j → H) × (Nm → H) ×
  (H → H → H) × ((H → H) → H)

--projections
|_| : ∀ {j}. HMod j → 𝒰
|_| M = M .fst
holesOf, varOf, appOf, hlamOf = ... --other projections of HMod

HEnc (j : nat) = (M : HMod j) → |M|

NMod (j : nat) = (H : 𝒰) × (Fin j → H) × (Nm → H) ×
  (H → H → H) × ((@N ⊸ H) → H)
```

The type of "nominal models" NMod_j is also defined as it will be useful later on. The carrier function $|{-}|$ and other projections are defined similarly for nominal models. Additionally we define explicit constructors mkHM, mkNM for $\mathsf{HMod}_j, \mathsf{NMod}_j$. For instance $\mathsf{mkNM}_j : (H : \mathcal{U}) \to (\mathsf{Fin}\, j \to H) \to (\mathsf{Nm} \to H) \to (H \to H \to H) \to ((@\mathsf{N} \multimap H) \to H) \to \mathsf{NMod}_j$.

We will show that we can define maps in and out of the encoding HEnc_j and prove the roundtrip at ULC if a certain binary parametricity axiom is assumed, as explained below.

```
ubd : ∀{j}. HEnc_j → Ltm_j
toh : ∀{j}. Ltm_j → HEnc_j
rdt-ulc : ∀{j}. (t : Ltm_j) → ubd(toh_j t) ≡ t
```

Unembedding We begin by defining the "unembedding" map denoted by `ubd`, which has a straightforward definition that does not involve nominal pattern matching. The name and the idea behind the definition come from [5, 7], where Atkey et al. were interested in comparing (non-nominal) syntax and HOAS Π-encodings using a strengthened form of binary parametricity called Kripke parametricity. We claim that the correspondence obtained here between Ltm_j and HEnc_j is a partial internalization of what these works achieve, and we do in fact rely on a (non-Kripke) binary parametricity axiom to prove `rdt-ulc`. This is discussed later, for now let us focus on the the example.

```
ubd {j} = λ(h : HEnc_j). h (Ltm-as-HMod j) where
    Ltm-as-HMod : ∀ j. HMod_j
    Ltm-as-HMod j = mkHM_j Ltm_j holes var app (hlamLtm j)
    hlamLtm : ∀ j.(Ltm_j → Ltm_j) → Ltm_j
    hlamLtm j f = lam(λ(x:@N). f(var (c x))))
```

So unembedding a HOAS term h consists of applying h at Ltm_j. This is possible thanks to the fact that Ltm_j can be equipped with a higher-order operation $\mathsf{hlamLtm}_j$.

Defining the map into HOAS The other map toh_j is defined by nominal recursion. In other words it is defined by recursion on its input $t : \mathsf{Ltm}_j$, i.e. using the eliminator of Ltm_j. We write the (uncurried) non-dependent eliminator as rec_j. Its type expresses that Ltm_j is the initial nominal model.

```
rec_j : (N : NMod_j) → Ltm_j → |N|
```

Hence in order to define toh_j we need to turn its codomain HEnc_j into a nominal model. For fields in NMod_j that are not binders this is straightforward.

```
holesH : Fin j → HEnc_j
varH : Nm → HEnc_j
appH : HEnc_j → HEnc_j → HEnc_j
```

```
holesH k = λ (M:HEnc_j). holesOf M k
varH n = λ (M:HEnc_j). varOf M n
appH u v = λ M. appOf M (u M) (v M)
```

Additionally we need to provide a function $\mathsf{lamH} : (@\mathsf{N} \multimap \mathsf{HEnc}_j) \to \mathsf{HEnc}_j$. This is done by using the SAP at HEnc_j, i.e. the following characterization of $(@\mathsf{N} \multimap \mathsf{HEnc}_j)$.

▶ **Theorem 1.** *We have* $\mathsf{mbump}_j : (@\mathsf{N} \multimap \mathsf{HMod}_j) \simeq \mathsf{HMod}_{j+1}$, $\mathsf{nbump}_j : (@\mathsf{N} \multimap \mathsf{NMod}_j) \simeq \mathsf{NMod}_{j+1}$, $\mathsf{ebump}_j : (@\mathsf{N} \multimap \mathsf{HEnc}_j) \simeq \mathsf{HEnc}_{j+1}$ *and* $\mathsf{lbump}_j : (@\mathsf{N} \multimap \mathsf{Ltm}_j) \simeq \mathsf{Ltm}_{j+1}$.

Proof. The lbump_j equivalence is the SAP at a (nominal) data type and its proof is performed using an encode-decode argument similar to the proof of $\mathsf{SAP}_{\mathsf{Nm}}$, or other data types as in [11, 34]. One salient feature of Ltm_j is its nominal constructor $\mathsf{var} : \mathsf{Nm} \to \mathsf{Ltm}_j$. Intuitively the type of var is the reason why the j index gets bumped to $j+1$. Indeed the SAP at $\mathsf{Nm} \to \mathsf{Ltm}_j$ contains an extra factor $\mathsf{Ltm}_{j+1} \times (\mathsf{Nm} \to \mathsf{Ltm}_{j+1}) \simeq (@\mathsf{N} \multimap \mathsf{Nm} \to \mathsf{Ltm}_j)$. We don't prove lbump_j and prove mbump_j instead, which uses a similar fact.

For space reasons we sometimes omit types for Σ and $\multimap$, e.g. we write $x \multimap T$ as shorthand for $(x : @\mathsf{N}) \multimap T$.

$$\begin{aligned}
x \multimap \mathsf{HMod}_j &\simeq (H' : x \multimap \mathcal{U}) \times (x \multimap [(\mathsf{Fin}\, j \to H'x) \times \ldots]) && \mathsf{SAP}_\Sigma \\
&\simeq (H' : x \multimap \mathcal{U}) \times (x \multimap (\mathsf{Fin}\, j \to H'x)) \times (x \multimap [\ldots]) && \mathsf{SAP}_\Sigma \\
&\simeq (H' : x \multimap \mathcal{U}) \times (\mathsf{holes}' : \mathsf{Fin}\, j \to (x \multimap H'x)) \times (x \multimap [\ldots]) && \mathsf{SAP}_{\mathsf{Fin}\, j, \to}
\end{aligned}$$

Since $\mathsf{Fin}\, j$ is a non-nominal data type its SAP instance asserts $\mathsf{Fin}\, j \simeq (x \multimap \mathsf{Fin}\, j)$, i.e. the only bridges in $\mathsf{Fin}\, j$ are reflexive bridges. This was proved in the binary case in [11, 34] for various data types. This is related to the fact that $\mathsf{Fin}\, j$ is "bridge-discrete" [11]. Moving on,

$$\begin{aligned}
&\simeq H' \times \mathsf{holes}' \times (x \multimap [(\mathsf{Nm} \to H'x) \times \ldots]) \\
&\simeq H' \times \mathsf{holes}' \times (x \multimap (\mathsf{Nm} \to H'x)) \times (x \multimap [\ldots]) && \mathsf{SAP}_\Sigma \\
&\simeq H' \times \mathsf{holes}' \times ((x \multimap \mathsf{Nm}) \to (x \multimap H'x)) \times (x \multimap [\ldots]) && \mathsf{SAP}_\to \\
&\simeq H' \times \mathsf{holes}' \times (\mathsf{foo} : (1 + \mathsf{Nm}) \to (x \multimap H'x)) \times (x \multimap [\ldots]) && \mathsf{SAP}_{\mathsf{Nm}}
\end{aligned}$$

$$\simeq H' \times \mathsf{holes}'_+ \times (\mathsf{var}' : \mathsf{Nm} \to x \multimap H'x) \times (x \multimap [\ldots])$$

where $\mathsf{holes}'_+ : \mathsf{Fin}\,(j+1) \to (x \multimap H'x)$ is defined as $\mathsf{holes}'_+\,0 = \mathsf{foo}\,(\mathsf{inl}\,(-))$ and $\mathsf{holes}'_+\,(k+1) = \mathsf{holes}'k$. The next two types in $x \multimap [\ldots]$ are computed using the SAP at $\to$. Thus so far we have shown that $@\mathsf{N} \multimap \mathsf{HMod}_j$ is equivalent to:

$$(H' : x \multimap \mathcal{U}) \times \mathsf{holes}'_+ \times \mathsf{var}' \times (\mathsf{app}' : (x \multimap H'x) \to (x \multimap H'x) \to (x \multimap H'x))$$
$$\times (\mathsf{hlam}' : ((x \multimap H'x) \to (x \multimap H'x)) \to (x \multimap H'x))$$

Now since the SAP at $\mathcal{U}$ is $\mathsf{Gel} : \mathcal{U} \overset{\simeq}{\to} (@\mathsf{N} \multimap \mathcal{U})$, we can do a change of variable in this last Σ type, i.e. use a variable $(K : \mathcal{U})$ instead of $(H' : x \multimap \mathcal{U})$ at the cost of replacing occurrences of H' by $\mathsf{Gel}\,K$. Pleasantly, occurrences of $(x \multimap H'x)$ become $(x \multimap \mathsf{Gel}\,K\,x)$ and are equivalent to K by the SAP for Gel types: the inverse of Gel is the nullary bridge type former. Hence $@\mathsf{N} \multimap \mathsf{HMod}_j \simeq \mathsf{HMod}_{j+1}$. Equivalences for ebump_j and nbump_j are obtained similarly. ◀

Next, we can define the desired operation $\mathsf{lamH} : (@\mathsf{N} \multimap \mathsf{HEnc}_j) \to \mathsf{HEnc}_j$ as $\mathsf{lamH}\,h' = \lambda(M : \mathsf{HMod}_j).\,\mathsf{lamOf}\,M\,\$\,\lambda(m : |M|).\,(\mathsf{ebump}_j\,h')\,(m :: M)$ where $(m :: M)$ is the HMod_{j+1} obtained out of $M : \mathsf{HMod}_{j+1}$ by pushing m onto the list $\mathsf{holesOf} M$. In other words, $\mathsf{holesOf}\,(m :: M)\,0 = m$ and $\mathsf{holesOf}\,(m :: M)\,(k+1) = \mathsf{holesOf}\,M\,k$. All in all we proved that HEnc_j is a nominal model, and since Ltm_j is the initial one we obtain the desired map $\mathsf{toh}_j : \mathsf{Ltm}_j \to \mathsf{HEnc}_j$.

```
tohⱼ = recⱼ (mkNMⱼ holesH varH appH
  (λ h' M. lamOf M (λ(m:|M|). ebumpⱼ h' (m::M))))
```

Observational parametricity of toh_j Computing the nullary observational parametricity of $\mathsf{toh}_j : \mathsf{Ltm}_j \to \mathsf{HEnc}_j$ will be needed to show the roundtrip at Ltm_j. Recall from Section 2.3 that, since SAP instances are fixed for Ltm_j and HEnc_j, we can define the observational parametricity of toh_j as $[\mathsf{toh}_j]_0 = \mathsf{ebump}_j^{-1} \circ (@\mathsf{N} \multimap \mathsf{toh}_j) \circ \mathsf{lbump}_j : \mathsf{Ltm}_{j+1} \to \mathsf{HEnc}_{j+1}$. It turns out that $[\mathsf{toh}_j]_0 \equiv \mathsf{toh}_{j+1}$. As explained in Section 2.3, the global, recursive nullary translation is not first-class. Using a notation $[-]_0$ that reminds the latter recursive translation has merit because the proof of the square $[\mathsf{toh}_j]_0 \equiv \mathsf{toh}_{j+1}$ is structurally the same than a "proof" by computation showing that $[\mathsf{toh}_j]_0$ reduces to toh_{j+1}. Indeed we can prove the following derivation (up to a small fixable lemma, see below) which looks like a series of reductions, ending in a term that looks like toh_{j+1} (some parts are proved/stated afterwards).

$$\begin{aligned}
[\mathsf{toh}_j]_0 &\equiv [\mathsf{rec}_j\,(\mathsf{mkNM}_j\,\mathsf{HEnc}_j\,\mathsf{holesH}_j\,\mathsf{varH}_j\,\mathsf{appH}_j\,\mathsf{lamH}_j)]_0\\
&\equiv [\mathsf{rec}_j]_0\,([\mathsf{mkNM}_j]_0\,[\mathsf{HEnc}_j]_0\,[\mathsf{holesH}_j]_0\,[\mathsf{varH}_j]_0\,[\mathsf{appH}_j]_0\,[\mathsf{lamH}_j]_0)\\
&\equiv \mathsf{rec}_{j+1}([\mathsf{mkNM}_j]_0\,[\mathsf{HEnc}_j]_0\,[\mathsf{holesH}_j]_0\,[\mathsf{varH}_j]_0\,[\mathsf{appH}_j]_0\,[\mathsf{lamH}_j]_0)\\
&\equiv \mathsf{rec}_{j+1}([\mathsf{mkNM}_j]_0\,\mathsf{HEnc}_{j+1}\,[\mathsf{holesH}_j]_0\,[\mathsf{varH}_j]_0\,[\mathsf{appH}_j]_0\,[\mathsf{lamH}_j]_0)\\
&\equiv \mathsf{rec}_{j+1}(\mathsf{mkNM}_{j+1}\,\mathsf{HEnc}_{j+1}\,([\mathsf{varH}_j]_0\,(\mathsf{inl}\,\mathsf{tt}) :: [\mathsf{holesH}_j]_0)\\
&\qquad\qquad ([\mathsf{varH}_j]_0 \circ \mathsf{inr})\,[\mathsf{appH}_j]_0\,[\mathsf{lamH}_j]_0)\\
&\equiv \mathsf{rec}_{j+1}(\mathsf{mkNM}_{j+1}\,\mathsf{HEnc}_{j+1}\,\mathsf{holesH}_{j+1}\,\mathsf{varH}_{j+1}\,\mathsf{appH}_{j+1}\,[\mathsf{lamH}_j]_0)
\end{aligned}$$

Before proving that $[\mathsf{lamH}]_0 \equiv \mathsf{lamH}_{j+1}$ we justify some of the steps above. First, since the function rec_j is dependent, its square, i.e. the equality $[\mathsf{rec}_j]_0 \equiv \mathsf{rec}_{j+1}$ is dependent as well. It asserts that $\forall(K' : @\mathsf{N} \multimap \mathsf{NMod}_j)(K_+ : \mathsf{NMod}_{j+1}).\,(\mathsf{nbump}_j\,K' \equiv K_+) \to (@\mathsf{N} \multimap \mathsf{rec}_j)\,K' \sim \mathsf{rec}_{j+1}\,N_+$ where $a \sim b$ means a, b are in correspondence through the equivalence $((x : @\mathsf{N}) \multimap (\mathsf{Ltm}_j \to |N'x|)) \simeq \mathsf{Ltm}_{j+1} \to |N_+|$. We were able to prove $[\mathsf{rec}_j] \equiv \mathsf{rec}_{j+1}$ by induction on K_+ and by extracting the exact definition of nbump_j from Theorem 1. We don't give the proof here. Second, $[\mathsf{HEnc}_j : \mathcal{U}]_0 \equiv \mathsf{HEnc}_{j+1}$. Indeed the reflexivity bridge $\lambda(\_ : @\mathsf{N}).\,\mathsf{HEnc}_j : @\mathsf{N} \multimap \mathcal{U}$ maps to the type $@\mathsf{N} \multimap \mathsf{HEnc}_j$ via the SAP equivalence $\mathsf{Gel}^{-1} : (@\mathsf{N} \multimap \mathcal{U}) \overset{\simeq}{\to} \mathcal{U}$. And by univalence $(@\mathsf{N} \multimap \mathsf{HEnc}_j) \equiv \mathsf{HEnc}_{j+1}$. Third, $\mathsf{varH} : \mathsf{Nm} \to \mathsf{HEnc}j$ thus $[\mathsf{varH}]_0 : 1 + \mathsf{Nm} \to \mathsf{HEnc}_{j+1}$. Furthermore we can prove that $[\mathsf{mkNM}_j]_0 A\,\mathsf{env}_j\,\mathsf{vr}_0\,\mathsf{ap}\,\mathsf{lm} \equiv \mathsf{mkNM}_{j+1}\,A\,(\mathsf{vr}_0(\mathsf{inl}(tt)) :: \mathsf{env}_j)\,(\mathsf{vr} \circ \mathsf{inr})\,\mathsf{ap}\,\mathsf{lm}$. Fourth we can show $([\mathsf{varH}]_0(\mathsf{inl}(tt)) :: [\mathsf{holesH}]_0) \equiv \mathsf{holesH}_{j+1}$, $([\mathsf{varH}]_0 \circ \mathsf{inr}) \equiv \mathsf{varH}_{j+1}$ and $[\mathsf{appH}]_0 \equiv \mathsf{app}_{j+1}$. The proofs pose no particular problem.

To obtain $[\mathsf{toh}_j]_0 \equiv \mathsf{toh}_{j+1}$ it remains to prove that $[\mathsf{lamH}]_0 \equiv \mathsf{lamH}_{j+1}$. Again we inspect lamH to do so and can prove

$$\begin{aligned}
[\mathsf{lamH}]_0 &\equiv [\lambda(k : @\mathsf{N} \multimap \mathsf{HEnc}_j)(M : \mathsf{HMod}_j).\,\mathsf{lamOf}\,M\,(\lambda m.\,\mathsf{ebump}_j\,k(m :: M))]_0 \\
&\equiv \lambda(k_+ : @\mathsf{N} \multimap \mathsf{HEnc}_{j+1})(M_+ : \mathsf{HMod}_{j+1}).\,\mathsf{lamOf}\,M_+(\lambda m_+.[\mathsf{ebump}_j]_0\,k_+\,([::]_0\,M_+\,m_+))
\end{aligned}$$

where $[::]_0 : (M_+ : \mathsf{HMod}_{j+1}) \to |M_+| \to \mathsf{HMod}_{j+2}$ is the translation of the function $- :: -$.

Things become less straightforward now because (1) $[\mathsf{ebump}_j]_0 \not\equiv \mathsf{ebump}_{j+1}$ and (2) $[::]_0\,M_+\,m_+ \not\equiv (m_+ :: M_+)$ (recall that the latter right-hand sides appear in the definition of toh_{j+1}). Instead we can prove $[\mathsf{ebump}_j]_0\,k_+ \equiv \mathsf{ebump}_{j+1}(\mathsf{flip}\,k_+)$ and $[::]_0\,M_+\,m_+ \equiv \mathsf{insert}_1\,M_+\,m_+$ where (1) flip is defined as the following composition $(x \multimap \mathsf{HEnc}_{j+1}) \to (x \multimap y \multimap \mathsf{HEnc}_j) \to (y \multimap x \multimap \mathsf{HEnc}_j) \to (y \multimap \mathsf{HEnc}_{j+1})$ and (2) insert_1 inserts the value m_+ at index 1 in the holes list of M_+ (by contrast $- :: -$ inserts values at index 0 instead). Hence what remains to be seen is the following lemma. We have not proved the lemma yet, but are confident we can provide a proof.

▶ **Lemma 2.** $\mathsf{ebump}_{j+1}(\mathsf{flip}\,k_+)(\mathsf{insert}_1\,M_+\,m_+) \equiv \mathsf{ebump}_{j+1}\,k_+\,(m_+ :: M_+)$

Observational parametricity of ubd_j The proof that $[\mathsf{ubd}_j]_0 \equiv \mathsf{ubd}_{j+1}$ is somewhat similar. We have $[\mathsf{ubd}_j]_0 \equiv \ldots$

$$\begin{aligned}
&\equiv [\lambda(h : \mathsf{HEnc}_j).\,(\mathsf{mkHM}_j\,\mathsf{Ltm}_j\,\mathsf{holes}_j\,\mathsf{var}_j\,\mathsf{app}_j\,\mathsf{hlamLtm_j})]_0 \\
&\equiv \lambda(h_+ : \mathsf{HEnc}_{j+1}).\,\big([\mathsf{mkHM}_j]_0\,\mathsf{Ltm}_{j+1}\,[\mathsf{holes}_j]_0\,[\mathsf{var}_j]_0\,[\mathsf{app}_j]\,[\mathsf{hlamLtm_j}]_0\big) \\
&\equiv \lambda\,h_+.\mathsf{mkHM}_{j+1}\,\mathsf{Ltm}_{j+1}\,([\mathsf{var}_j]_0.\mathsf{fst} :: [\mathsf{holes}_j]_0)\,([\mathsf{var}_j]_0.\mathsf{snd})\,[\mathsf{app}_j]\,[\mathsf{hlamLtm_j}]_0 \\
&\equiv \lambda h_+.\mathsf{mkHM}_{j+1}\,\mathsf{Ltm}_{j+1}\,\mathsf{var}_{j+1}\,\mathsf{var}_{j+1}\,\mathsf{app}_{j+1}\,[\mathsf{hlamLtm_j}]_0
\end{aligned}$$

And $[\mathsf{hlamLtm_j}]_0 \equiv \ldots$

$$\begin{aligned}
&\equiv [\lambda\,(f : \mathsf{Ltm}_j \to \mathsf{Ltm}_j).\,\mathsf{lam}_j\,(\lambda(x : @\mathsf{N}).\,f(\mathsf{var}_j\,(\mathsf{c}\,x)))]_0 \\
&\equiv \lambda\,(f_+ : \mathsf{Ltm}_{j+1} \to \mathsf{Ltm}_{j+1}).\,\mathsf{lam}_{j+1}\,(\lambda(x : @\mathsf{N}).\,f_+([\lambda(y : @\mathsf{N}).\mathsf{var}_j(\mathsf{c}\,y)]_0\,x)) \\
&\equiv \lambda\,(f_+ : \mathsf{Ltm}_{j+1} \to \mathsf{Ltm}_{j+1}).\,\mathsf{lam}_{j+1}\,(\lambda(x : @\mathsf{N}).\,f_+(\mathsf{lbump}_j\,(\lambda_.\,\mathsf{var}_j\,(\mathsf{c}\,x)))
\end{aligned}$$

By definition of lbump_j, which is extracted from Theorem 1, it turns out that the term $\mathsf{lbump}_j\,(\lambda\_.\,\mathsf{var}_j\,(\mathsf{c}\,x))$ reduces to $\mathsf{var}_{j+1}\,(\mathsf{c}\,x)$. Thus $[\mathsf{hlamLtm_j}]_0 \equiv \mathsf{hlamLtm}_{j+1}$, and thus $[\mathsf{ubd}_j]_0 \equiv \mathsf{ubd}_{j+1}$.

Roundtrip at Ltm_j We now prove that $\forall j\,(t : \mathsf{Ltm}_j).\,t \equiv \mathsf{ubd}_j\,(\mathsf{toh}_j t)$. Note that within the proof we use a specific binary parametricity axiom. The proofs for constructors other than lam_j are easy. For $t \equiv \mathsf{lam}_j(g : @\mathsf{N} \multimap \mathsf{Ltm}_j)$ we must prove that if the induction hypothesis holds $(g^\bullet : (z : @N) \multimap (gz \equiv \mathsf{ubd}_j\,(\mathsf{toh}_j(gz))))$ then $\mathsf{lam}_j\,g \equiv \mathsf{ubd}_j\,(\mathsf{toh}_j\,(\mathsf{lam}_j\,g))$. We are indeed doing an induction on t, i.e. using the dependent eliminator of Ltm_j. Let $g^\bullet$

in the context and let us compute the right-hand side of the latter equation. We use the \$ application operator found e.g. in Haskell. This operator is defined as function application but associates to the right, improving clarity.

$$
\begin{aligned}
&\equiv \mathsf{ubd}_j \,\$\, \lambda M.\,\mathsf{lamOf}\; M \,\$\, \lambda m.\,(\mathsf{ebump}_j \circ (@\mathsf{N} \multimap \mathsf{toh}_j))\; g\,(m :: M) && \text{(def.)}\\
&\equiv \mathsf{hlamLtm}_j \,\$\, \lambda m.\,(\mathsf{ebump}_j \circ (@\mathsf{N} \multimap \mathsf{toh}_j))\; g\,(m :: M) && \text{(def.)}\\
&\equiv \mathsf{hlamLtm}_j \,\$\, \lambda m.\,(\mathsf{toh}_{j+1} \circ \mathsf{lbump}_j)\; g\,(m :: M) && ([\mathsf{toh}_j]_0 \equiv \mathsf{toh}_{j+1})\\
&\equiv \mathsf{lam}_j \,\$\, \lambda(x : @\mathsf{N}).\,(\mathsf{toh}_{j+1} \circ \mathsf{lbump}_j)\; g\,(\mathsf{var}(\mathsf{c}\,x) :: (\mathsf{Ltm}_j, \ldots)) && \text{(def.)}\\
&\equiv \mathsf{lam}_j \,\$\, \lambda(x : @\mathsf{N}).\,(\mathsf{toh}_{j+1} \circ \mathsf{lbump}_j)\; g \,\$\\
&\qquad\qquad \mathsf{mkHM}_j\; \mathsf{Ltm}_j\; (\mathsf{var}(\mathsf{c}\,x) :: \mathsf{holes}_j)\; \mathsf{var}_j\; \mathsf{app}_j\; \mathsf{hlamLtm}_j && \text{(def.)}
\end{aligned}
$$

The latter model is of type HMod_{j+1} and we observe that it looks similar to $\mathsf{mkHM}_{j+1}\,\mathsf{Ltm}_{j+1}\,\mathsf{holes}_{j+1}\,\mathsf{var}_{j+1}\,\mathsf{app}_{j+1}\,\mathsf{hlamLtm}_{j+1} : \mathsf{HMod}_{j+1}$. More formally, for $(x : @\mathsf{N})$ in context the (graph of the) map $\mathsf{foo} : \mathsf{Ltm}_{j+1} \to \mathsf{Ltm}_j : u \mapsto \mathsf{lbump}_j^{-1}\, u\, x$ turns out be a structure-preserving relation between $\mathsf{Ltm}_{j+1}, \mathsf{Ltm}_j : \mathsf{HMod}_{j+1}$. The proof is not difficult. This suggests to use binary parametricity, which for dependent functions k out of HMod_{j+1} asserts that k preserve such structure-preserving relations. So taking $k = (\mathsf{toh}_{j+1} \circ \mathsf{lbump}_j)\, g$, in a context containing $(x : @\mathsf{N})$, binary parametricity grants the first equality of this chain:

$$
\begin{aligned}
&\mathsf{lam}_j \,\$\, \lambda(x : @\mathsf{N}).\,(\mathsf{toh}_{j+1} \circ \mathsf{lbump}_j)\; g\,(\mathsf{var}(\mathsf{c}\,x) :: (\mathsf{Ltm}_j, \ldots))\\
&\equiv \mathsf{lam}_j \,\$\, \lambda(x : @\mathsf{N}).\,\mathsf{foo} \,\$\, (\mathsf{toh}_{j+1} \circ \mathsf{lbump}_j)\; g\,(\mathsf{Ltm}_{j+1} : \mathsf{HMod}_{j+1})\\
&\equiv \mathsf{lam}_j \,\$\, \lambda(x : @\mathsf{N}).\,\mathsf{foo} \,\$\, (\mathsf{toh}_{j+1}\,(\mathsf{lbump}_j g))\,(\mathsf{Ltm}_{j+1} : \mathsf{HMod}_{j+1})\\
&\equiv \mathsf{lam}_j \,\$\, \lambda(x : @\mathsf{N}).\,\mathsf{foo} \,\$\, (\mathsf{ubd}_{j+1} \circ \mathsf{toh}_{j+1})(\mathsf{lbump}_j\, g)\\
&\equiv \mathsf{lam}_j \,\$\, \lambda(x : @\mathsf{N}).\,(\mathsf{lbump}_j^{-1} \$\, (\mathsf{ubd}_{j+1} \circ \mathsf{toh}_{j+1})(\mathsf{lbump}_j\, g))\; x\\
&\equiv \mathsf{lam}_j \,\$\, \mathsf{lbump}_j^{-1} \$\, (\mathsf{ubd}_{j+1} \circ \mathsf{toh}_{j+1})(\mathsf{lbump}_j\, g)
\end{aligned}
$$

Now it remains to see that $g \equiv \mathsf{lbump}_j^{-1} \$\, (\mathsf{ubd}_{j+1} \circ \mathsf{toh}_{j+1})(\mathsf{lbump}_j\, g)$, i.e. $\mathsf{lbump}_j\, g \equiv (\mathsf{ubd}_{j+1} \circ \mathsf{toh}_{j+1})(\mathsf{lbump}_j\, g)$. Let us denote the roundtrip function by $r_j := \mathsf{ubd}_j \circ \mathsf{toh}_j$. (1) The induction hypothesis tells us $g^\bullet : (z : @\mathsf{N}) \multimap (gz \equiv r_j(gz))$ so by the SAP at $\equiv$ we get $g \equiv \lambda(z : @\mathsf{N}).\, r_j(gz)$ and by applying lbump_j we get $\mathsf{lbump}_j\, g \equiv \mathsf{lbump}_j(\lambda(z : @\mathsf{N}).\, r_j(gz))$. (2) Observe that $[r_j]_0 \equiv [\mathsf{ubd}_j \circ \mathsf{toh}_j]_0 \equiv [\mathsf{ubd}_j]_0 \circ [\mathsf{toh}_j]_0 \equiv \mathsf{ubd}_{j+1} \circ \mathsf{toh}_{j+1} = r_{j+1}$. The function r_j has type $\mathsf{Ltm}_j \to \mathsf{Ltm}_j$ and $[r_j]_0 \equiv r_{j+1}$ desugars to $r_{j+1} \circ \mathsf{lbump}_j \equiv \mathsf{lbump}_j \circ (@\mathsf{N} \multimap r_j) = \lambda q.\, \mathsf{lbump}_j(\lambda(y : @\mathsf{N}).\, r_j(q\, y))$. Applying both sides to $g : @\mathsf{N} \multimap \mathsf{Ltm}_j$ we get $r_{j+1}(\mathsf{lbump}_j\, g) \equiv \mathsf{lbump}_j(\lambda(y : @\mathsf{N}).\, r_j(g\, y))$. (3) Now, step (1) told us $\mathsf{lbump}_j g \equiv \mathsf{lbump}_j(\lambda(y : @\mathsf{N}).\, r_j(q\, y))$ and step (2) told us $\mathsf{lbump}_j(\lambda(y : @\mathsf{N}).\, r_j(g\, y)) \equiv r_{j+1}(\mathsf{lbump}_j\, g)$. Hence $\mathsf{lbump}_j g \equiv r_{j+1}(\mathsf{lbump}_j\, g) = \mathsf{ubd}_{j+1}\,(\mathsf{toh}_{j+1}\,(\mathsf{lbump}_j\, g))$. This concludes the proof of the roundtrip equality at Ltm_j.

Synthetic Kripke parametricity The above definitions of toh_j, ubd_j and the proof of the roundtrip at Ltm_j draw inspiration from [5, 7]. In [5], R. Atkey shows that, in the presence of *Kripke* binary parametricity, the non-nominal syntax of ULC is equivalent to the standard HOAS Π-encoding $\forall A.(A \to A \to A) \to ((A \to A) \to A) \to A$. The Kripke binary parametricity of $f : A \to B$ is written $[f]_{K2}$ and is very roughly a proof of the following fact:

$$[f]_{\mathsf{K2}} : \forall(W : \mathsf{PreOrder})(a_0\, a_1 : A).\, \mathsf{Mon}(W, [A]_2\, a_0\, a_1) \to \mathsf{Mon}(W, [B]_2\,(f\, a_0)\,(f\, a_1)).$$

Here $\mathsf{Mon}(W, X)$ is the set of monotonic functions from W to X and $[A]_2$ denotes the binary (non-Kripke) parametricity translation. So the Kripke parametricity of f asserts that f preserves monotonic families of relations.

We claim without proof that the correspondence we obtain between Ltm_j and HEnc_j (done in nullary PTT with a binary axiom) maps to the analogue correspondence in Atkey's model (which uses $\mathbb{N}$-restricted Kripke 2-ary parametricity). I.e. externalizing $(0, 2)$-ary parametricity leads to $\mathbb{N}$-restricted Kripke 2-ary parametricity.

With regards to unrestricted Kripke parametricity, we observe that the quantification on preorders is *hard-coded* into $[f]_{\mathsf{K2}}$, i.e. $[f]_{\mathsf{K2}}$ is not the mere meta-theoretical conjunction of its W-restricted Kripke parametricities. In particular $[f]_{\mathsf{K2}}$ lives one universe level higher than f which is peculiar. Interestingly this hard-coding plays a crucial role in the proof of the roundtrip at HOAS. Indeed, within Atkey's model, the proof introduces a variable $B : \mathcal{U}$ and uses (roughly) $[f]_{\mathsf{K2}}(\mathsf{List}\, B)\,(1 : \mathcal{U})\,(B : \mathcal{U})$ to conclude. This is a form of diagonalization that can not be internalized in our nullary PTT. Hence we claim that in our system the roundtrip at HOAS can not be proved by a similar argument.

5 Related and Future Work

We remark here that $1 + \mathsf{Nm} \simeq @\mathsf{N} \multimap \mathsf{Nm}$ and $(@\mathsf{N} \multimap \mathsf{Ltm}_0) \simeq \mathsf{Ltm}_1$ were proved semantically in [15].

We have already discussed the most closely related work in nominal type theory [31, 29, 30, 12, 27] and internally parametric type theory [9, 11, 34] in detail in Sections 3 and 2 respectively.

Semantics of other nominal frameworks In Section 2.4 we modelled nullary PTT in presheaves over the nullary affine cube category $\square_0$, which is also the base category of the Schanuel sheaf topos [26, §6.3], which is in turn equivalent to the category of nominal sets [26] that forms the model of FreshMLTT [27]. FreshML [31] is modelled in the Fraenkel-Mostowski model of set theory, which nominal sets seem to have been inspired by. Schöpp and Stark's bunched system [30] is modelled in a class of categories, including the Schanuel topos, that is locally cartesian closed and also equipped with a semicartesian closed structure. Finally, $\lambda^{\Pi И}$ [12] has a syntactic soundness proof.

FreshMLTT and $\lambda^{\Pi И}$ support having multiple name types. As long as a set of name types $\mathfrak{N}$ is fixed in the metatheory, we can support the same and justify this semantically by considering presheaves over $\boxslash_2 \times \square_0^{\mathfrak{N}}$.

Transpension The category $\square_0$ is an example of a cube category without diagonals and as such, its internal language is among the first candidates to get a transpension type [20] with workable typing rules [18]. Dual to the fact that universal and existential name quantification semantically coincide, so do freshness and transpension. As such, the transpension type is already present as Gel in the current system, but Gel's typing rules are presently weaker: the rules GelF and GelI remove a part of the context, rather than quantifying it (which can be done manually using ext in the current system). Nuyts and Devriese [20] explain the relationship between Gel and transpension in more generality, and apply the transpension type and related operations to nominal type theory; however all in a setting where all name- and dimension-related matters are handled using modal techniques.

References

1 C. B. Aberlé. Parametricity via cohesion. *CoRR*, abs/2404.03825, 2024. URL: `https://doi.org/10.48550/arXiv.2404.03825`, `arXiv:2404.03825`, `doi:10.48550/ARXIV.2404.03825`.

2 Thorsten Altenkirch, Yorgo Chamoun, Ambrus Kaposi, and Michael Shulman. Internal parametricity, without an interval. *Proc. ACM Program. Lang.*, 8(POPL):2340–2369, 2024. doi:10.1145/3632920.

3 Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In Aaron Stump and Hongwei Xi, editors, *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*, pages 57–68. ACM, 2007. doi:10.1145/1292597.1292608.

4 Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Robert Harper, Kuen-Bang Hou (Favonia), and Daniel R. Licata. Syntax and models of cartesian cubical type theory. *Math. Struct. Comput. Sci.*, 31(4):424–468, 2021. doi:10.1017/S0960129521000347.

5 Robert Atkey. Syntax for free: Representing syntax with binding using parametricity. In Pierre-Louis Curien, editor, *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings*, volume 5608 of *Lecture Notes in Computer Science*, pages 35–49. Springer, 2009. doi:10.1007/978-3-642-02273-9_5.

6 Robert Atkey, Neil Ghani, and Patricia Johann. A relationally parametric model of dependent type theory. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 503–516. ACM, 2014. doi:10.1145/2535838.2535852.

7 Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In Stephanie Weirich, editor, *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, pages 37–48. ACM, 2009. doi:10.1145/1596638.1596644.

8 Jesper Bengtson and Joachim Parrow. Formalising the pi-calculus using nominal logic. *Log. Methods Comput. Sci.*, 5(2), 2009. URL: http://arxiv.org/abs/0809.3960.

9 Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. A presheaf model of parametric type theory. In Dan R. Ghica, editor, *The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22-25, 2015*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 67–82. Elsevier, 2015. URL: https://doi.org/10.1016/j.entcs.2015.12.006, doi:10.1016/J.ENTCS.2015.12.006.

10 Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free - parametricity for dependent types. *J. Funct. Program.*, 22(2):107–152, 2012. doi:10.1017/S0956796812000056.

11 Evan Cavallo and Robert Harper. Internal parametricity for cubical type theory. *Log. Methods Comput. Sci.*, 17(4), 2021. URL: https://doi.org/10.46298/lmcs-17(4:5)2021, doi:10.46298/LMCS-17(4:5)2021.

12 James Cheney. A dependent nominal type theory. *Log. Methods Comput. Sci.*, 8(1), 2012. doi:10.2168/LMCS-8(1:8)2012.

13 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. *FLAP*, 4(10):3127–3170, 2017. URL: http://collegepublications.co.uk/ifcolog/?00019.

14 Narya development team. Nominal syntax — narya documentation, nov 2025. Accessed: 2025-11-19. URL: https://narya.readthedocs.io/en/latest/nominal.html.

15 Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 204–213. IEEE Computer Society, 1999. doi:10.1109/LICS.1999.782616.

16 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992. doi:10.1016/0890-5401(92)90008-4.

17 Guilhem Moulin. *Internalizing Parametricity.* PhD thesis, Chalmers University of Technology, Gothenburg, Sweden, 2016. URL: http://publications.lib.chalmers.se/publication/235758-internalizing-parametricity.

18 Andreas Nuyts. Transpension for cubes without diagonals. In *Workshop on Homotopy Type Theory / Univalent Foundations*, 2025. URL: `https://lirias.kuleuven.be/retrieve/803969`.

19 Andreas Nuyts and Dominique Devriese. Degrees of relatedness: A unified framework for parametricity, irrelevance, ad hoc polymorphism, intersections, unions and algebra in dependent type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 779–788. ACM, 2018. `doi:10.1145/3209108.3209119`.

20 Andreas Nuyts and Dominique Devriese. Transpension: The right adjoint to the Pi-type. *Log. Methods Comput. Sci.*, 20(2), 2024. URL: `https://doi.org/10.46298/lmcs-20(2:16)2024`, `doi:10.46298/LMCS-20(2:16)2024`.

21 Andreas Nuyts, Andrea Vezzosi, and Dominique Devriese. Parametric quantifiers for dependent type theory. *Proc. ACM Program. Lang.*, 1(ICFP):32:1–32:29, 2017. `doi:10.1145/3110276`.

22 Martin Odersky. A functional theory of local names. In Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin, editors, *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pages 48–59. ACM Press, 1994. `doi:10.1145/174675.175187`.

23 Andrew Pitts. Nominal sets and dependent type theory. In *TYPES*, 2014. URL: `https://www.irif.fr/~letouzey/types2014/slides-inv3.pdf`.

24 Andrew M. Pitts. Nominal logic, a first order theory of names and binding. *Inf. Comput.*, 186(2):165–193, 2003. `doi:10.1016/S0890-5401(03)00138-X`.

25 Andrew M. Pitts. Nominal system T. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 159–170. ACM, 2010. `doi:10.1145/1706299.1706321`.

26 Andrew M Pitts. *Nominal sets: Names and symmetry in computer science*. Cambridge University Press, 2013.

27 Andrew M. Pitts, Justus Matthiesen, and Jasper Derikx. A dependent type theory with abstractable names. In Mauricio Ayala-Rincón and Ian Mackie, editors, *Ninth Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2014, Brasília, Brazil, September 8-9, 2014*, volume 312 of *Electronic Notes in Theoretical Computer Science*, pages 19–50. Elsevier, 2014. URL: `https://doi.org/10.1016/j.entcs.2015.04.003`, `doi:10.1016/J.ENTCS.2015.04.003`.

28 John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 513–523. North-Holland/IFIP, 1983.

29 Ulrich Schöpp. *Names and binding in type theory*. PhD thesis, University of Edinburgh, UK, 2006. URL: `https://hdl.handle.net/1842/1203`.

30 Ulrich Schöpp and Ian Stark. A dependent type theory with names and binding. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic*, pages 235–249, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. `doi:10.1007/978-3-540-30124-0_20`.

31 Mark R. Shinwell, Andrew M. Pitts, and Murdoch Gabbay. FreshML: programming with binders made simple. In Colin Runciman and Olin Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 263–274. ACM, 2003. `doi:10.1145/944705.944729`.

32 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. URL: `https://homotopytypetheory.org/book/`.

33 Christian Urban. Nominal techniques in isabelle/hol. *J. Autom. Reason.*, 40(4):327–356, 2008. URL: `https://doi.org/10.1007/s10817-008-9097-2`, `doi:10.1007/S10817-008-9097-2`.

34 Antoine Van Muylder, Andreas Nuyts, and Dominique Devriese. Internal and observational parametricity for cubical agda. *Proc. ACM Program. Lang.*, 8(POPL):209–240, 2024. `doi:10.1145/3632850`.

35 Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical agda: A dependently typed programming language with univalence and higher inductive types. *J. Funct. Program.*, 31:e8, 2021. doi:10.1017/S0956796821000034.