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, has a closed induction principle and is well-behaved w.r.t. weakening. It is known that name abstraction types can be presented either as an existential or as a universal quantification on names. Existential name abstractions support matching on name-binding patterns but have cumbersome typing rules; universal ones have clean rules but apparently no such nominal matching. In this work we show that this matching ability and other nominal features are recovered in a type theory consisting of (1) Nullary (0-ary) Internally Parametric Type Theory (nullary PTT), (2) a type of names and a novel name induction principle, (3) nominal data types. This type theory is a legitimate nominal framework: it has universal and (non-primitive) existential name abstractions, a freshness type former, restricted name swapping and local-scope operations. Nominal pattern matching is recovered via term-relevant nullary parametricity. We provide an example involving synthetic Kripke parametricity.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Nominal techniques, Parametricity

Digital Object Identifier 10.4230/LIPIcs.TYPES.2025.12

Funding Andreas Nuyts holds a Postdoctoral Fellowship from the Research Foundation – Flanders (FWO; 12AB225N). This research is partially funded by the Research Fund KU Leuven and the Internal Funds KU Leuven.

Acknowledgements We thank the reviewers for their time and helpful comments.

1 Introduction

There are many ways to formally define the syntax of a language with binders. In particular, nominal frameworks [25, 33, 28, 32, 11, 35] 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 the 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. First, α-equivalent terms are definitionally equal, a consequence of how $@\mathsf{N} \multimap -$ is axiomatized. Second, nominal data types have a (closed) induction principle. Third, by contrast with the De Bruijn representation, weakening by a name simply means extending the context with that name, and all programs in the metalanguage are automatically stable under this operation. Fourth, formal reasoning with nominal syntax can sometimes match on-paper reasoning [35]. 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 [32, 31] 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 an element of that type, i.e. a name abstraction, can be understood as a *name-term pair up to α-renaming* $\langle x, a\rangle$ where a may bind the freshly chosen name x. 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 [33] 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 about 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 $\langle x, a\rangle$, and such 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 harder in a proof assistant environment.

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 [28, 11]. 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 alleviating the above issues and relying on the notion of *parametricity*. Concisely put, n-ary parametricity asserts that types behave as n-ary reflexive graphs. More precisely, the n-ary parametricity of a type T computes by induction on T a certain n-ary reflexive graph structure for T. Even more precisely, (1) the n-ary parametricity of a (open) type T computes to a certain (open) n-ary relation written $[T]_n : T \to \ldots \to T \to \mathcal{U}$, (2) the n-ary parametricity of a (open) term $t : T$ computes to a certain reflexivity edge at t, i.e. $[t]_n : [T]_n\, t\, \ldots\, t$. Parametricity is most often used in its binary form, at types of polymorphic functions, where it specializes to a uniformity property [30, 38]. For example if $t : T = \forall(X : \mathcal{U}).\, X \to X$ then $[t]_2 : [T]_2\, t\, t$ is a proof that t maps related types to related functions. Indeed $[-]_2$ is defined by induction (not given here) and $[T]_2\, t\, t$ reduces to $\forall(X_0 X_1 : \mathcal{U})(R : X_0 \to X_1 \to \mathcal{U}).\forall x_0\, x_1.\, R\, x_0\, x_1 \to R\,(t\, X_0\, x_0)\,(t\, X_1\, x_1)$. Intuitively this property disallows t from inspecting its type argument $X : \mathcal{U}$. It can be shown that the property entails that t equals the identity $\lambda\, X\, x.\, x$.

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

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

In the above account, parametricity is regarded as a property that is defined and proven to hold *externally* about a dependent type theory (DTT), as in [9]. In other words, the parametricity property, or translation, is a map $[-]_n$ defined by induction on the (open) types and terms of DTT. By contrast, n-ary *internally* parametric type theory (n-ary PTT for short) [10, 17, 20, 22, 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 internalizing 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 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 [37] which play the role of equality proofs in the latter. However, bridges do not satisfy various properties of paths, e.g. they cannot be composed and one cannot 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. that are fresh for the bridge. 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 excerpts a collection of equivalences known as the Structure Identity Principle (SIP) in HoTT/UF [34]. Concisely, it states that at every type K, equality is equivalent to observational equality [3]. The Bridge column of Figure 1 shows an analogous Structure Relatedness Principle (SRP) [36] 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*. Accordingly, in this setting the parametricity of a term k is k's reflexivity bridge $\lambda(\_ : \mathsf{N}).\, k : \mathsf{Bridge}\, K\, k\ \ldots\ k$ mapped through the SRP equivalence at K.

Parametric Nominal Type Theory. In this work, we put forward and elaborate an idea that existed in the community [18, 21], which is to use nullary PTT (i.e. 0-ary PTT) as the foundation for nominal dependent type theory. Specifically, we contribute Parametric Nominal Type Theory (PNTT). Concisely put, PNTT = nullary PTT + a type of names with an induction principle + rules for nominal data types.

In more detail, PNTT is obtained from nullary PTT in several steps. First, we make the simple observation that universal name abstraction types have the same rules as the nullary Bridge types from nullary PTT: nullary bridges and universal abstractions are simply affine functions out of the interval. Second, we show that, on its own, nullary 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 [28] (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 of names $\vdash \mathsf{Nm}$. It comes equipped with a novel eliminator called *name induction*, which expresses for a term $\Gamma \vdash n : \mathsf{Nm}$ and a bridge variable x in Γ, that either n is just x, or x is *fresh* in n. Fourth, PNTT also has rules for the nominal data types we wish to study, like the Ltm type.

Nominal pattern matching can be indirectly 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 type former $@\mathsf{N} \multimap -$ commutes with every type former in a specific way, including (1) standard type formers, (2) the Nm type and (3) nominal data types. For standard type formers, the SAP asserts that name abstraction commutes as one might expect, e.g. $(@\mathsf{N} \multimap A \to B) \simeq ((@\mathsf{N} \multimap A) \to (@\mathsf{N} \multimap B))$. 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 for Ltm asserts that there is an equivalence $e : (@\mathsf{N} \multimap \mathsf{Ltm}) \simeq \mathsf{Ltm}_1$ between $@\mathsf{N} \multimap \mathsf{Ltm}$ and the following data type (this was proved semantically by Hofmann [14]). 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
```

When matching on $\mathsf{lam}\, g : \mathsf{Ltm}$ 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 as the foundation for nominal dependent type theory, and provide evidence for the suitability of this foundation.

- In Section 2, we present Parametric Nominal Type Theory (PNTT). PNTT is a variant of the univalent parametric type theory of Cavallo and Harper [10] where (1) we replace the arity $n = 2$ by $n = 0$ (see Section 2.1), (2) we construct a type Nm from the bridge interval $@\mathsf{N}$ and provide a name induction principle (see Section 2.2), (3) we add rules for the specific nominal data types we wish to study. We explain how the above primitives validate our Structure Abstraction Principle (SAP). We discuss semantics and soundness in Section 2.4.
- 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 shows nominal techniques in PNTT in action. Section 4.1 shows that the nullary translation of data types lets us indirectly emulate nominal pattern matching: defining functions by matching on patterns which bind variables. Section 4.2 connects the Ltm nominal data type to a nominal HOAS representation, and serves two purposes. First, it illustrates that more often than not, proving the correctness of a function $f : D \to E$ defined by recursion out of a nominal data type D requires computing the parametricity translation of f, i.e. the parametricity of f is term-relevant. Second, the example sheds some light on the notion of *Kripke* binary parametricity [5]. A proof in Atkey's Kripke model is ported to PNTT, and the Kripke aspect of it is emulated by nullary parametricity.

Binary parametric cubical type theory has already been implemented [36]. Thus our work offers a clear path to a 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 Parametric Nominal Type Theory (PNTT)

PNTT is largely based on the binary PTT of Cavallo and Harper (CH type theory, [10]). Concretely, 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, (2) adding novel rules to turn the bridge interval @N into a full-fledged type Nm, including a name induction principle and (3) adding rules for the desired nominal data types. For the impatient reader, the relevant rules of PNTT appear in Figure 2 and $\mathsf{Gel}\, A\, x$ can be understood as the elements of A for which x is fresh.

Cubical type theory. Apart from its parametricity primitives, the CH theory is a cubical type theory in the way it handles equality proofs (*paths*), and so is our system. Cubical type theory (cubical TT, [12, 4]) is a form of homotopy type theory (HoTT, [34]) 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 (see Figure 1) at any concrete, fully known type.

The SIP matters to us because, e.g., univalence is used to characterize the bridge type at the universe $(@\mathsf{N} \multimap \mathcal{U}) \simeq \mathcal{U}$. Otherwise, the fact that our system is a cubical type theory is not of primary significance, although we wish to remain as close as possible to the CH theory to be able to reuse their proofs and semantics.

Accordingly, we only need to know that the primitives introduced by cubical type theory validate the SIP in order to showcase our theorems and examples. Yet we list these primitives 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 the path relation, (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 Theorem 4.2.3 of [34]: 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 @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).

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

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

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

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

$$\frac{(x : @\mathsf{N}) \in \Gamma \quad \Gamma\backslash x \vdash A\ \mathsf{type}}{\Gamma \vdash \mathsf{Gel}\, A\, x\ \mathsf{type}}\ \textsc{Gel}\mathrm{F} \qquad \frac{\Gamma, (x : @\mathsf{N}) \vdash g : \mathsf{Gel}\, A\, x}{\Gamma \vdash \mathsf{ung}\, (\lambda\, x.\, g) : A}\ \textsc{Gel}\mathrm{E}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{ung}\, (\lambda\, x.\, \mathsf{gel}\, a\, x) = a}\ \textsc{Gel}\beta$$

$$\frac{(x : @\mathsf{N}) \in \Gamma \quad \Gamma\backslash x \vdash a : A}{\Gamma \vdash \mathsf{gel}\, a\, x : \mathsf{Gel}\, a\, x}\ \textsc{Gel}\mathrm{I} \qquad \frac{\text{prem. of } \textsc{Gel}\mathrm{F} \quad \Gamma\backslash x, (y : @\mathsf{N}) \vdash g_0, g_1 : \mathsf{Gel}\, A\, y \quad \Gamma \vdash \mathsf{ung}(\lambda y.\, g_0) = \mathsf{ung}(\lambda y.\, g_1) : A}{\Gamma \vdash g_0[x/y] = g_1[x/y] : \mathsf{Gel}\, A\, x}\ \textsc{Gel}\eta$$

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

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

$$\frac{\text{prem. of } \mathsf{Nm}\mathrm{E} - n}{\Gamma \vdash \mathsf{ind}_{\mathsf{Nm}}\, x\, (\mathsf{c}\, x)\, b_0\, (\lambda\, g.\, b_1) = b_0 : B[\mathsf{c}\, x/y]}\ \mathsf{Nm}\beta_0 \qquad \frac{\text{prem. of } \mathsf{Nm}\mathrm{E} - n \quad \Gamma\backslash x \vdash n : \mathsf{Nm}}{\Gamma \vdash \mathsf{ind}_{\mathsf{Nm}}\, x\, n\, b_0\, (\lambda\, g.\, b_1) = b_1[\mathsf{gel}\, n\, x/g] : B[n/z]}\ \mathsf{Nm}\beta_1$$

Figure 2 Parametricity primitives of PNTT. Prem. of X - y means premises of rule X except y. For binding rules such as EXT, we rely on the invertibility of both variable and name abstraction to bind using λ.

The presence of @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 $-\backslash-$ (as in [10], section 2.1). 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 @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.

Nullary bridges. The type former of dependent bridges with dependent codomain $\Gamma, (x : @\mathsf{N}) \vdash A$ is written $(x : @\mathsf{N}) \multimap A$. 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 $a' : (y : @\mathsf{N}) \multimap A$ 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 @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.

[1] A detail: variables $i : \mathrm{I}$ and cubical constraints are not removed [10, 36] as they cannot depend on x.

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 corresponding to these primitives in the empty context 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 [36], 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 omit 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 [10, 36]. 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 (as a side note, the proof also uses a lemma akin to a propositional η-rule for extent).

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

By way of comparison, this equivalence appears in [28] (Example 2.2) though the left-to-right direction is implemented via a composite capturing operation instead of a primitive like extent. Proofs of the above equivalence in the binary case can be found in [10, 36]. 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 [10, 36]. 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 be soundly 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, 10] need a primitive to convert an 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 $\mathcal{U} \simeq (@\mathsf{N} \multimap \mathcal{U})$ where $\mathsf{Gel}^{-1}\, A' := (x : @\mathsf{N}) \multimap A'\, x$, and by using the rules of Gel and ext, and notably univalence. The proof also requires showing 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 [10, 36] 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']
```

To our knowledge the equivalence $\mathcal{U} \simeq (@\mathsf{N} \multimap \mathcal{U})$ had not been considered in the literature on nominal techniques. 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 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. Besides, it is unclear whether there exists a sound notion of data types D with "constructors" of the form $@\mathsf{N} \multimap D$.

Perhaps the most important result in our system 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 $\mathsf{c} : @\mathsf{N} \multimap \mathsf{Nm}$ or a constant bridge. This principle is proved based on our rules for Nm, explained 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 $\mathsf{c}\,x$, 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. And we have:

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

The EXTβ 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 [10] who developed a relational encode-decode technique to prove the SRP at data types. The type Nm is defined via an induction principle, and a similar technique can be applied.

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

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

t2pre : (x:@N) ⊸ (1 + Gel Nm x) → Gel (1 + Nm) x
t2pre x (inl tt) = gel (inl tt) 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)

encode : (@N ⊸ Nm) → 1 + Nm
encode 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 $s = \mathsf{inl}\,\mathsf{tt}$ then $(\mathsf{ung} \circ t_2 \circ t_1 \circ \mathsf{decode})\,s = (\mathsf{ung} \circ t_2 \circ t_1)(\lambda x.\,\mathsf{c}\,x) \overset{\mathsf{Nm}\beta_0}{=} (\mathsf{ung} \circ t_2)(\lambda x.\,\mathsf{inl}\,\mathsf{tt}) = \mathsf{ung}\,(\lambda x.\,\mathsf{gel}\,(\mathsf{inl}\,\mathsf{tt})\,x) = \mathsf{inl}\,\mathsf{tt} = s$. If $s = \mathsf{inr}\,n$ then $(\mathsf{ung} \circ t_2 \circ t_1 \circ \mathsf{decode})\,s = (\mathsf{ung} \circ t_2 \circ t_1)(\lambda x.\,n) \overset{\mathsf{Nm}\beta_1}{=} (\mathsf{ung} \circ t_2)(\lambda x.\,\mathsf{inr}(\mathsf{gel}\,n\,x)) = \mathsf{ung}\,(\lambda x.\,\mathsf{ext}\,[\lambda g' y.\,\mathsf{gel}\,(\mathsf{inr}(\mathsf{ung}\,g'))\,y]\,x\,(\mathsf{gel}\,n\,x)) \overset{\text{EXT}\beta}{=} \mathsf{ung}\,(\lambda x.\,\mathsf{gel}\,(\mathsf{inr}\,n)x) = \mathsf{inr}\,n = s$.

The other roundtrip is the last type in the following chain of equivalences. The first type is a sufficient condition that can be understood as a propositional η-rule for Nm.

$$\begin{aligned}
(x : @\mathsf{N}) \multimap (n : \mathsf{Nm}) \to (n \equiv_{\mathsf{Nm}} \mathsf{ext}\,[\mathsf{decode} \circ \mathsf{encode}]\,x\,n) \quad &\simeq_{(\text{SAP}_\to)}\\
(n' : @\mathsf{N} \multimap \mathsf{Nm}) \to (x : @\mathsf{N}) \multimap (n'x \equiv_{\mathsf{Nm}} (\mathsf{decode} \circ \mathsf{encode})\,n'\,x) \quad &\simeq_{(\text{SAP}_\equiv)}\\
(n' : @\mathsf{N} \multimap \mathsf{Nm}) \to n' \equiv (\mathsf{decode} \circ \mathsf{encode})\,n'
\end{aligned}$$

Table 1 The Structure Abstraction Principle.

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

The ext in the first line vanishes in the second because $\text{EXT}\beta$ triggers. We prove the sufficient condition. Let $(x : @\mathsf{N}), (n : \mathsf{Nm})$ in context. We reason by name induction. If $n = \mathsf{c}\,x$ then the right-hand side is $\mathsf{ext}\,[\mathsf{decode} \circ \mathsf{encode}]\,x\,(\mathsf{c}\,x) \overset{\text{EXT}\beta}{=} ((\mathsf{decode} \circ \mathsf{encode})\,\mathsf{c})\,x$. From the proof above we know that $\mathsf{encode}\,\mathsf{c} = \mathsf{inl}\,\mathsf{tt}$, and by definition $\mathsf{decode}\,(\mathsf{inl}\,\mathsf{tt}) = \mathsf{c}$. So both sides of the equality are equal to $\mathsf{c}\,x$. Else, formally we must provide a b_1 argument to $\mathsf{ind}_{\mathsf{Nm}}$. We define $b_1 := b_1'\,x$ where the type of b_1' is the first type in this SAP derivation (performed in the empty context and where $[\ldots] := \mathsf{decode} \circ \mathsf{encode}$):

$$
\begin{aligned}
&(x : @\mathsf{N}) \multimap (g : \mathsf{Gel}\,\mathsf{Nm}\,x) \to \mathsf{forg}\,x\,g \equiv \mathsf{ext}\,[\ldots]\,x\,(\mathsf{forg}\,x\,g)\\
\simeq\ &(g' : (x : @\mathsf{N}) \multimap \mathsf{Gel}\,\mathsf{Nm}\,x) \to (x : @\mathsf{N}) \multimap \mathsf{forg}\,x\,(g'\,x) \equiv \mathsf{ext}\,[\ldots]\,x\,(\mathsf{forg}\,x\,(g'\,x))\\
\simeq\ &(n : \mathsf{Nm}) \to (x : @\mathsf{N}) \multimap \mathsf{forg}\,x\,(\mathsf{gel}\,n\,x) \equiv \mathsf{ext}\,[\ldots]\,x\,(\mathsf{forg}\,x\,(\mathsf{gel}\,n\,x))
\end{aligned}
$$

By a computation above, $\mathsf{forg}\,x\,(\mathsf{gel}\,n\,x) = n$ since x is fresh in n. The right-hand side computes to n as well by $\text{EXT}\beta$ and definition of $[\ldots] = \mathsf{decode} \circ \mathsf{encode}$.

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 a → P b → P(app a b)) →
(∀g. ((x:@N) ⊸ P(g x)) → P(lam g)) → ∀t.P t
```

2.3 The Structure Abstraction Principle (SAP)

For types and terms. As hinted above, PNTT validates the Structure Abstraction Principle (SAP). Similar to the SIP of HoTT/UF, or the SRP of binary PTT [36], the SAP defines how each type former commutes with the (nullary) Bridge type former. Table 1 lists several SAP instances, including the ones we have encountered so far. Each instance is[2] of the form

[2] Orienting SAP_K this way enables $\mathsf{SAP}_{\mathcal{U}} = \mathsf{Gel}$ and $\mathsf{SAP}_{\Pi} = \mathsf{ext}$.

$\forall \cdots . K' \simeq ((x : @\mathsf{N}) \multimap K)$, where K, K' may depend on some parameters. 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, and is further discussed below). Types of DTT can contain terms and accordingly there exists a SAP for terms. Let $\mathsf{SAP}_K : K' \simeq (@\mathsf{N} \multimap K)$ be a SAP instance. The SAP of the term $k : K$ is a pair (k', p) where $k' : K'$ and $p : \mathsf{SAP}_K\, k' \equiv \lambda(\_ : @\mathsf{N}).k$, i.e. k' is the reflexivity bridge $\mathsf{rfl}\, k := \lambda(\_ : @\mathsf{N}).k$ of $k : K$ through the SAP_K equivalence, up to propositional equality.

Relationship to the translation. For types, it is basically the case that the type K' provided by the SAP at K is $[K]_0 : \mathcal{U}$, the recursively and externally defined nullary parametricity translation of K. The n-ary translation is briefly discussed in Section 1. Note that in practice it is sometimes useful to consider SAP instances where K' is a type different but equivalent to $[K]_0$, but this detail can be ignored for clarity. For terms, it is also the case that the SAP at $k : K$ provides a k' which is $[k]_0 : [K]_0$, the recursively defined nullary translation of $k : K$ (analogously it is sometimes useful to let k' be a term different but propositionally equal to $[k]_0$). From that perspective, $\mathsf{SAP}_K : [K]_0 \simeq @\mathsf{N} \multimap K$ says that the Bridge type of K is its translation up to an equivalence, and the SAP of a term $k : K$ says that $[k]_0 \equiv \mathsf{SAP}_K^{-1}\, (\lambda(\_ : @\mathsf{N}).k)$, i.e. the reflexivity bridge at $k : K$ is its translation up to a propositional equality. It is useful to give a name to the latter right-hand side. If a SAP instance is fixed for K we define the *observational parametricity* of a term $k : K$ to be $[k]_0^{\mathcal{O}} := \mathsf{SAP}_K^{-1}\, (\lambda(\_ : @\mathsf{N}).k)$. The various parametricity mappings for types and terms can be informally summarized as follows, where $\overset{m}{\to}$ denotes a metatheoretical function.

$$\begin{aligned} [-]_0 : \mathcal{U} \overset{m}{\to} \mathcal{U} \quad &; \quad [-]_0, [-]_0^{\mathcal{O}} : \{K : \mathcal{U}\} \overset{m}{\to} K \overset{m}{\to} [K]_0 \\ @\mathsf{N} \multimap - : \mathcal{U} \to \mathcal{U} \quad &; \quad \mathsf{rfl} : \{K : \mathcal{U}\} \to (k : K) \to @\mathsf{N} \multimap K \\ \mathsf{SAP} : (K : \mathcal{U}) \overset{m}{\to} [K]_0 \simeq @\mathsf{N} \multimap K \quad &; \quad \mathsf{SAP} : \{K : \mathcal{U}\} \overset{m}{\to} (k : K) \overset{m}{\to} [k]_0 \equiv [k]_0^{\mathcal{O}} \end{aligned}$$

As an example, we compute the observational parametricity of a function $f : A \to B$. 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, we have $[f]_0^{\mathcal{O}} : A' \to B'$ and $[f]_0^{\mathcal{O}} = \mathsf{SAP}_B^{-1} \circ (@\mathsf{N} \multimap f) \circ \mathsf{SAP}_A$ where $(@\mathsf{N} \multimap f) = \lambda a'\, x.\, f\, (a' x)$ 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 a translation K' that is equivalent to $@\mathsf{N} \multimap K$. In the binary case, we have shown in earlier work [36] 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 and SAP instances can be derived straightforwardly. Although we believe the same process applies here, we have not constructed the corresponding DSL, instead applying SAP instances manually in examples.

2.4 About Semantics

Similar to our theory PNTT, our model is a simple extension of Cavallo and Harper's presheaf model [10] with the arity of parametricity changed from 2 to 0. This model is $\mathsf{Psh}(\boxslash_2 \times \square_0)$, where $\boxslash_2$ is the binary cartesian cube category [4] for path dimensions, and $\square_0$ is the 0-ary

affine cube category for bridge dimensions. Concretely $\square_0$ is the opposite of the category of finite ordinals and injections. We keep this section overall succinct, not for lack of formal understanding, because these semantics are a well-understood combination of established models of type theory [13, 15, 4, 8, 10].

In this model the type Nm is interpreted as the Yoneda embedding of the base object with 1 bridge dimension and no path dimensions. The elimination and computation rules for this type are based on a semantic isomorphism $x : @\mathsf{N} \vdash \mathsf{Nm} \cong 1 + \mathsf{Gel}\,\mathsf{Nm}\,x$ which is straightforwardly checked. In fact, the above (taken as an equivalence) is equivalent (by several invocations of the SAP) to $(@\mathsf{N} \multimap \mathsf{Nm}) \simeq 1 + \mathsf{Nm}$, an equivalence semantically proven by Hofmann [14]. Kan fibrancy of Nm is semantically trivial, since we can tell from the base category that any path in Nm will be constant. As for computation of Kan operations at Nm, we propose to wait until all arguments 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 [12, 4]. Regarding nominal data types, which we only consider through examples 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.

We note that $\square_0$ is the base category (carrier of the site) of the Schanuel sheaf topos [27, §6.3] which is equivalent to the category of nominal sets [27], used to model FreshMLTT [28]. 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 $\square_0$.

3 Nominal Primitives for Free

In this section, we argue that the central features in a number of earlier nominal dependent type systems can essentially be recovered in nullary PTT. One notable exception is nominal pattern matching (discussed in the next section), which is recovered in our extension PNTT. Concretely, we consider here Shinwell, Pitts and Gabbay's FreshML [33], Schöpp and Stark's bunched nominal type theory BNTT [32, 31], Cheney's $\lambda^{\Pi\mathcal{N}}$ [11] and Pitts, Matthiesen and Derikx's FreshMLTT [28]. 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, $\lambda^{\Pi\mathcal{N}}$) support multiple name types, something we could also easily accommodate, but leave out so as not to distract from the main contributions. 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 [8] and Cavallo and Harper [10]), but to distinguish them from the other systems.

Universal name quantification. Universal name quantification is available in BNTT, $\lambda^{\Pi\mathcal{U}}$ 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 $\lambda^{\Pi\mathcal{U}}$ 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 [8, 10]) 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 for which x is fresh are classified by the type $\mathsf{Gel}\,A\,x$. BNTT's free-from types only apply to closed types A, so GelF is 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 [31, §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 BNTT 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 $[\mathsf{gel}\,y\,x/z]$.[3] 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.[4] We then have an equivalence

$$\begin{aligned}(y : B) \to ((x : @\mathsf{N}) \multimap T[\mathsf{gel}\,y\,x/z]) &\simeq (y : (x : @\mathsf{N}) \multimap \mathsf{Gel}\,B\,x) \to ((x : @\mathsf{N}) \multimap T[y\,x/z]) \\ &\simeq (x : @\mathsf{N}) \multimap (z : \mathsf{Gel}\,B\,x) \to T\end{aligned}$$

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.

[3] The original rule immediately subsumes a substitution $\Delta \to (x : @\mathsf{N}, z : \mathsf{Gel}\,A\,x)$.

[4] This may raise questions about preservation of substitution w.r.t. Θ. However, it is not at all standard for dependently typed elimination rules to guarantee preservation of substitution w.r.t. a telescope depending on the eliminee.

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 $\mathsf{matchdecl} : (e : @\mathsf{N} \multimap B) \to (t : @\mathsf{N} \multimap B \to T) \to (@\mathsf{N} \multimap T)$. 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 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 $\lambda\,t\,x\,y.\,t\,y\,x$. Note that name swapping needs to happen w.r.t. affine names: swapping the cartesian names $x, y : \mathsf{Nm}$ in the term $\mathsf{app}\,(\mathsf{app}\,(\mathsf{var}\,x)\,(\mathsf{var}\,y))\,(\mathsf{var}\,z) : \mathsf{Ltm}$ does not commute with contraction $-[x/x, x/z]$. 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 [21, 19] into the current system. Lastly, using swap, we can follow Pitts et al. [28] 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 [23, 26] 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:

$$\frac{\Gamma, x : @\mathsf{N} \vdash T\,\mathsf{type} \quad \Gamma, x : @\mathsf{N} \vdash t : T \quad x \text{ is fresh for } t : T}{\Gamma \vdash \nu\,x.\,t : \nu\,x.\,T}$$

$\Gamma, x : @\mathsf{N} \vdash \nu\,x.\,t = t : T$
Added: $\nu\,x.\,t = t$ if x is not free in t.

$$\frac{\Gamma \vdash S\,\mathsf{type} \quad \Gamma, x : @\mathsf{N} \vdash t : S \quad x \text{ is fresh for } t : S}{\Gamma \vdash \nu\,x.\,t : S}$$

It is used [24] 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:

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

In this formulation, it is now clear that $\nu x.\, t$ can be implemented as $\mathsf{ung}\ (\lambda x.\, t)$, while the two computation rules follow from Gelη and Gelβ.

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 PNTT, explained in Section 2.

4.1 Patterns That Bind

Some nominal frameworks with existential name-abstraction types [33] 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 (e.g. eqabs in Section 1). We explain how this feature is indirectly recovered in PNTT.

The following nominal data type [7] is the nominal syntax of the π-calculus [16], a formal language whose expressions represent concurrently communicating processes. The constructors stand for: terminate, silent computation step, parallelism, non-determinism, channel allocation, receiving[5], and sending. The arguments of (output) type Proc can be thought of as continuations, e.g. nu prepends a channel allocation instruction.

```
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
```

An example of a process 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 the application $p'\, b$ does not typecheck and that this substitution operation called nsub below must be defined recursively, as done in [7]. 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)
...
```

[5] Binders have arguments $@\mathsf{N} \multimap -$ and not $\mathsf{Nm} \to -$ since the latter could lead to exotic process terms checking the bound name for equality to other names.

Note how patterns (0) and (1) match on a binding pattern of the form $\lambda x.\,\mathsf{inp}\ m\ (q'\ x)$, and 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 (its constructors use a suggestive $[-]_0$ notation):

```
data AProc : 𝒰 where
  [nil]₀ : AProc
  [τpre]₀ : AProc → AProc
  [par]₀ , [sum]₀ : AProc →
     AProc → AProc
```

```
  [nu]₀ : (@N ⊸ AProc) → AProc
  [inp]₀ : 1 + Nm → (@N ⊸ AProc) → AProc
  [out]₀ : 1 + Nm → 1 + Nm → AProc → AProc
```

We can then define $\mathsf{nsub}' : \mathsf{Nm} \to \mathsf{AProc} \to \mathsf{Proc}$ by induction on the second argument. Indeed the informal clauses (0), (1) can be translated into formal ones using the $[\mathsf{inp}]_0$ constructor. More precisely, clause (1) uses the function $[\mathsf{inp}]_0\,(\mathsf{inl}\,\mathsf{tt})$ of type $(@\mathsf{N} \multimap \mathsf{AProc}) \to \mathsf{AProc}$ and clause (0) uses the function $\lambda(n : \mathsf{Nm}).\,[\mathsf{inp}]_0\,(\mathsf{inr}\,n)$ of type $\mathsf{Nm} \to (@\mathsf{N} \multimap \mathsf{AProc}) \to \mathsf{AProc}$.

4.2 A HOAS Example

In the example sketched below, we connect the nominal syntax of the untyped lambda calculus (ULC) to a nominal, higher-order abstract syntax (HOAS) representation. The example is performed within PNTT plus a single (standard) binary parametricity axiom. Our proof is a port to PNTT of an existing argument by Atkey [5], which uses external *Kripke* binary parametricity to connect the De Bruijn (non-nominal) syntax of ULC to a (non-nominal) HOAS representation.

This example serves two purposes. The first is to illustrate, by porting a complex argument from one setting to the other, how the combination of nullary (for nominal reasoning) and binary parametricity allows for a reasoning style similar to that of Kripke binary parametricity, and moreover with closely related denotational semantics. This topic is elaborated further at the end of this section.

Secondly, the example illustrates that more often than not, proving the correctness of a function $f : D \to E$ defined by recursion out of a nominal data type D requires computing the parametricity translation of f. Indeed, for $\mathsf{bind} : (@\mathsf{N} \multimap D) \to D$ a binder (constructor) of D, the β-rule of D says that the redex $f(\mathsf{bind}\,d')$ computes to a term involving $@\mathsf{N} \multimap f$, the action of f on bridges (see Section 2.3). So proving f correct often requires characterizing $@\mathsf{N} \multimap f$ up to propositional equality. This characterization is the SAP for f (also in Section 2.3) $[f]_0 \equiv_{[D]_0 \to [E]_0} [f]_0^{\mathcal{O}}$, where $[f]_0$ is the recursive translation of f and $[f]_0^{\mathcal{O}} := \mathsf{SAP}_E^{-1} \circ (@\mathsf{N} \multimap f) \circ \mathsf{SAP}_D$ the observational parametricity of f.

SAP proofs for composite terms like f are unwieldy because one has to show why the SAP instances for the primitives in f compose to the global one and this involves manually reproducing the structure of the term f at the level of SAP proofs. For this reason some instances of the SAP for terms are assumed to hold below, without proof (see paragraph "Roundtrip at Ltm_j"). The unwieldiness of interval-based PTTs is a good selling point for *observational* parametric type theories (see Section 5), where the SAP is not needed since a first-class recursive translation attempts to replace $@\mathsf{N} \multimap -$ altogether.

As a side note regarding the above function $f : D \to E$, its domain D could feature a binder of type $(@\mathsf{N} \multimap \ldots \multimap @\mathsf{N} \multimap D) \to D$ binding m fresh names. In that case proving the correctness of f may involve computing its m-fold iterated nullary parametricity, for the same reasons as above. No such examples are investigated here.

Let us now get to the actual argument. The nominal syntax of ULC is expressed as the following data type family Ltm, parametrized by a natural number $j : \mathtt{nat}$.

```
data Ltm (j : nat) : 𝒰
  env : 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.

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

--projections
|_| : ∀ {j}. HMod j → 𝒰
|_| M = M .fst
envOf, 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 $\mathsf{mkHM}_j, \mathsf{mkNM}_j$ 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 between Ltm_j and the encoding HEnc_j in both ways, and sketch a conditional proof that they compose to the identity at Ltm_j.

```
ubd : ∀{j}. HEnc_j → Ltm_j
toh : ∀{j}. Ltm_j → HEnc_j
rdt-Ltm : ∀{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 recursion. The name and the idea behind the definition come from [5, 6], 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 (see end of section for a comparison with our system/model).

```
ubd {j} = λ(h : HEnc_j). h (LtmHMod j) where
    LtmHMod : ∀ j. HMod_j
    LtmHMod j = mkHM_j Ltm_j env 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 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.

```
recⱼ : (N : NModⱼ) → Ltmⱼ → |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.

```
envH : Fin j → HEncⱼ
varH : Nm → HEncⱼ
appH : HEncⱼ → HEncⱼ → HEncⱼ
```

```
envH k = λ (M:HModⱼ). envOf M k
varH n = λ (M:HModⱼ). 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* $\mathsf{ebump}_j : (@\mathsf{N} \multimap \mathsf{HEnc}_j) \simeq \mathsf{HEnc}_{j+1}$, $\mathsf{mbump}_j : (@\mathsf{N} \multimap \mathsf{HMod}_j) \simeq \mathsf{HMod}_{j+1}$, $\mathsf{nbump}_j : (@\mathsf{N} \multimap \mathsf{NMod}_j) \simeq \mathsf{NMod}_{j+1}$ *and* $\mathsf{lbump}_j : (@\mathsf{N} \multimap \mathsf{Ltm}_j) \simeq \mathsf{Ltm}_{j+1}$.

Intuitively these results hold because $@\mathsf{N} \multimap -$ preserves all primitive type formers except Nm, which gets translated to a sum $1 + \mathsf{Nm}$. So translating each of the aforementioned types boils down to replacing Nm with $1 + \mathsf{Nm}$, which refactors to incrementing j.

Proof. We only prove mbump_j. The proofs of ebump_j and nbump_j are similar, and lbump_j is proved using an encode-decode argument [10, 36] similar to the proof of $\mathsf{SAP}_{\mathsf{Nm}}$. We sometimes omit types for Σ and $\multimap$, e.g. we write $x \multimap T$ as shorthand for $(x : @\mathsf{N}) \multimap T$.

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

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 [10, 36]. Moving on,

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

where $\mathsf{env}'_+ : \mathsf{Fin}\,(j+1) \to (x \multimap H' x)$ is defined as $\mathsf{env}'_+\, 0 = \mathsf{foo}\,(\mathsf{inl}\,\mathsf{tt})$ and $\mathsf{env}'_+\,(k+1) = \mathsf{env}' k$. 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 $(H' : x \multimap \mathcal{U}) \times \mathsf{env}'_+ \times \mathsf{var}' \times (\mathsf{app}' : (x \multimap H' x) \to (x \multimap H' x) \to (x \multimap H' x)) \times (\mathsf{hlam}' : ((x \multimap H' x) \to (x \multimap H' x)) \to (x \multimap H' x))$. Now since the SAP at $\mathcal{U}$ is $\mathsf{Gel} : \mathcal{U} \xrightarrow{\simeq} (@\mathsf{N} \multimap \mathcal{U})$, we can do a change of variable in the above translation of HMod_j, 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 $(x \multimap \mathsf{Gel}\, K\, x)$, which is equivalent to K by the SAP for Gel types. Hence $@\mathsf{N} \multimap \mathsf{HMod}_j \simeq \mathsf{HMod}_{j+1}$. ◀

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 $M : \mathsf{HMod}_j$ by pushing m onto the list $\mathsf{envOf} M$. In other words, $\mathsf{envOf}\, (m :: M)\, 0 = m$ and $\mathsf{envOf}\, (m :: M)\, (k+1) = \mathsf{envOf}\, M\, k$. All in all we proved that HEnc_j is a nominal model. Since Ltm_j is the initial one we obtain the desired map $\mathsf{toh}_j : \mathsf{Ltm}_j \to \mathsf{HEnc}_j$.

```
toh_j = rec_j (mkNM_j Henc_j envH varH appH
  (λ h' M. lamOf M (λ(m:|M|). ebump_j h' (m::M))))
```

Roundtrip at Ltm_j. We sketch a conditional proof that $\forall j\, (t : \mathsf{Ltm}_j).\, t \equiv \mathsf{ubd}_j\, (\mathsf{toh}_j t)$, by (nominal) induction, i.e. using the dependent eliminator of Ltm_j. Our proof is conditional since it relies on two axioms. (1) We use a specific axiom called binx that is an instance of standard binary parametricity. It is natural to rely on such an axiom here since our proofs attempt to emulate Kripke *binary* parametricity (the Kripke aspect is in fact emulated by nullary parametricity). An expected model for this standard binary parametricity statement is $\mathsf{Psh}(◪_2 \times □_0 \times □_2)$ where $□_2$ is the original binary affine cube category used by Cavallo and Harper (CH) [10] to model binary parametricity, see Section 2.4. In fact, we expect binx to be provable in PNTT extended with CH internal binary parametricity operators. (2) Some steps require characterizing the observational parametricity $[-]_0^{\mathcal{O}}$ (defined in Section 2.3) of certain terms. These steps are written with a gray background below and are *assumed*. As mentioned above characterizing the observational parametricity $[f]_0^{\mathcal{O}}$ of a complex term f is unwieldy because such characterizations must prove that $[-]_0^{\mathcal{O}}$ commutes with the primitives used in f and reproduce by hand the structure of the term f. We currently have a partial paper proof of these steps. In the future we would like to implement our theory (or maybe an observational version of it) to fully validate this example.

The proofs for constructors other than lam_j are easy. For $t = \mathsf{lam}_j(g : @\mathsf{N} \multimap \mathsf{Ltm}_j)$ we must prove that if the induction hypothesis holds $(g^\bullet : (z : @\mathsf{N}) \multimap (g\, z \equiv \mathsf{ubd}_j\, (\mathsf{toh}_j(g\, z))))$ then $\mathsf{lam}_j\, g \equiv \mathsf{ubd}_j\, (\mathsf{toh}_j\, (\mathsf{lam}_j\, g))$. Let $g^\bullet$ be in context and let us compute the right-hand side *RHS* 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.

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

The latter model is of type HMod_{j+1} and we observe that it looks similar to $\mathsf{LtmHMod}_{j+1} = \mathsf{mkHM}_{j+1}\, \mathsf{Ltm}_{j+1}\, \mathsf{env}_{j+1}\, \mathsf{var}_{j+1}\, \mathsf{app}_{j+1}\, \mathsf{hlamLtm}_{j+1} : \mathsf{HMod}_{j+1}$. More formally, for $(x : @\mathsf{N})$ in context, the graph of $\mathsf{plug}_x : \mathsf{Ltm}_{j+1} \to \mathsf{Ltm}_j : u \mapsto \mathsf{lbump}_j^{-1}\, u\, x$ turns out be a structure-preserving relation between the two models $\mathsf{LtmHMod}_{j+1}$, $(\mathsf{var}(\mathsf{c}\, x) :: \mathsf{LtmHMod}_j) : \mathsf{HMod}_{j+1}$. This suggests using the binary parametricity of the dependent function $\mathsf{inEnc}_g := (\mathsf{toh}_{j+1} \circ \mathsf{lbump}_j)\, g : (M : \mathsf{HMod}_{j+1}) \to M$ which appears in our computed *RHS*. The binary parametricity of inEnc_g is $[\mathsf{inEnc}_g]_2 : \forall (M_0\, M_1 : \mathsf{HMod}_{j+1})(R : [\mathsf{HMod}_{j+1}]_2\, M_0\, M_1).\, |R|(\mathsf{inEnc}_g\, M_0)(\mathsf{inEnc}_g\, M_1)$, where $[\mathsf{HMod}_{j+1}]_2\, M_0\, M_1$ is defined to be the type of structure-preserving relations between M_0, M_1, and $|R| : |M_0| \to |M_1| \to \mathcal{U}$ extracts the carrier of R. The specific binary parametricity axiom we use is $\mathsf{binx} := [\mathsf{inEnc}_g]_2\, \mathsf{LtmHMod}_{j+1}\, (\mathsf{var}(\mathsf{c}\, x) :: \mathsf{LtmHMod}_j)\, (\mathsf{Graph}(\mathsf{plug}_x))$, i.e. the dependent function inEnc_g commutes with the

function plug_x. We then have:

$$\begin{aligned}
RHS &\equiv \mathsf{lam}_j\,\$\,\lambda(x : @\mathsf{N}).\,(\mathsf{toh}_{j+1} \circ \mathsf{lbump}_j)\,g\,(\mathsf{var}(\mathsf{c}\,x) :: \mathsf{LtmHMod}_j) \\
&\equiv \mathsf{lam}_j\,\$\,\lambda(x : @\mathsf{N}).\,\mathsf{plug}_x\,\$\,(\mathsf{toh}_{j+1} \circ \mathsf{lbump}_j)\,g\,\mathsf{LtmHMod}_{j+1} && (\mathsf{binx}) \\
&\equiv \mathsf{lam}_j\,\$\,\lambda(x : @\mathsf{N}).\,\mathsf{plug}_x\,\$\,(\mathsf{ubd}_{j+1} \circ \mathsf{toh}_{j+1})(\mathsf{lbump}_j\,g) && (\text{def. } \mathsf{ubd}_{j+1}) \\
&\equiv \mathsf{lam}_j\,\$\,\lambda(x : @\mathsf{N}).\,(\mathsf{lbump}_j^{-1}\$\,(\mathsf{ubd}_{j+1} \circ \mathsf{toh}_{j+1})(\mathsf{lbump}_j\,g))\,x && (\text{def. } \mathsf{plug}_x)
\end{aligned}$$

Now it remains to show that the latter $\equiv \mathsf{lam}_j\,\$\,\lambda(x : @\mathsf{N}).\,g\,x$, i.e. that the following equality holds $\mathsf{eqNextWorld} : \mathsf{lbump}_j\,g \equiv (\mathsf{ubd}_{j+1} \circ \mathsf{toh}_{j+1})(\mathsf{lbump}_j\,g)$. Let us write $r_j := \mathsf{ubd}_j \circ \mathsf{toh}_j$. (1) The induction hypothesis tells us $g^\bullet : (z : @\mathsf{N}) \multimap (g\,z \equiv r_j(g\,z))$ 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) The observational parametricity of r_j is such that $[r_j]_0^{\mathcal{O}} \equiv [\mathsf{ubd}_j \circ \mathsf{toh}_j]_0^{\mathcal{O}} \equiv [\mathsf{ubd}_j]_0^{\mathcal{O}} \circ [\mathsf{toh}_j]_0^{\mathcal{O}} \equiv \mathsf{ubd}_{j+1} \circ \mathsf{toh}_{j+1} = r_{j+1}$. The function r_j has type $\mathsf{Ltm}_j \to \mathsf{Ltm}_j$ so $[r_j]_0^{\mathcal{O}} \equiv r_{j+1}$ entails (by def. of $[r_j]_0^{\mathcal{O}}$, see Section 2.3) $r_{j+1} \circ \mathsf{lbump}_j \equiv \mathsf{lbump}_j \circ (@\mathsf{N} \multimap r_j)$. Applying both sides to $g : @\mathsf{N} \multimap \mathsf{Ltm}_j$ we get $r_{j+1}(\mathsf{lbump}_j\,g) \equiv \mathsf{lbump}_j(\lambda(z : @\mathsf{N}).\,r_j(g\,z))$. (3) Using step (1), (2) and transitivity of $\equiv$ we get $\mathsf{lbump}_j g \equiv r_{j+1}(\mathsf{lbump}_j\,g) = \mathsf{ubd}_{j+1}\,(\mathsf{toh}_{j+1}\,(\mathsf{lbump}_j\,g))$. Hence $\mathsf{eqNextWorld}$ holds and this concludes our proof.

Synthetic Kripke parametricity. Kripke parametricity is a strengthened form of binary parametricity that was introduced in a denotational model built by Atkey [5] in order to prove an adequacy result for HOAS: the De Bruijn syntax of ULC is equivalent to the HOAS Π-encoding $\forall A.(A \to A \to A) \to ((A \to A) \to A) \to A$. One way to explain Kripke binary parametricity is to compare it to standard binary parametricity. The latter asserts that polymorphic functions $f : (A : \mathcal{U}) \to T(A)$ map relations R to proofs of relatedness over R (this is explained for $T(A) = A \to A$ in Section 1). By contrast, briefly speaking, the *Kripke* binary parametricity of a polymorphic function $f : (A : \mathcal{U}) \to T(A)$ is written $[f]_{\mathsf{K2}}$ and asserts that for every (*) preorder W, f maps any W-monotonic family of relations $(w \mapsto R_w) : W \to A_0 \to A_1 \to \mathcal{U}$ to a monotonic function/proof that for every $w : W$ the outputs $f\,A_0, f\,A_1$ are related over R_w ($\mathcal{U}$ is implicitly ordered by logical implication).

We think that the example discussed above externalizes to the fragment of Atkey's Kripke model where the W preorder is systematically replaced by $(\mathbb{N}, \leq)$, natural numbers with the usual ordering.[6] Indeed the example above is performed within PNTT with an additional binary parametricity axiom binx. The idea is then that the nullary parametricity aspect of our argument gets translated to the monotonic quantification aspect of Atkey's Kripke model. That is, $(0, 2)$-ary parametricity emulates $(\mathbb{N}, \leq)$-restricted Kripke 2-ary parametricity.

Regarding 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 Atkey's proof of the roundtrip at $\forall A.(A \to A \to A) \to ((A \to A) \to A) \to A$. Indeed, within Atkey's model, the proof introduces a variable $B : \mathcal{U}$ and uses $[f]_{\mathsf{K2}}$ at preorder $W := \mathsf{List}\,B$ with prefix ordering, and at type $A_0 := B$ (and some other type for A_1). So B intervenes both to determine the kind of parametricity being used ($W := \mathsf{List}\,B$), and as an input for this parametricity ($A_0 := B$). We see this as a form of diagonalization, which we were not able to internalize.

[6] A mismatch: our model uses the category $\square_0$ (Section 2.4) while Atkey's uses *preorders* like $(\mathbb{N}, \leq)$.

5 Related and Future Work

We have already discussed the most closely related work in internally parametric type theory [8, 10, 36] and in nominal type theory [33, 31, 32, 11, 28] in detail in Sections 2 and 3 respectively. Note that a subtle error in the construction of the model shown in [8] was recently spotted in [2] and investigated in [29].

Regarding the semantics of nominal frameworks, PNTT is modelled (see Section 2.4) in $\mathsf{Psh}(\boxslash_2 \times \square_0)$, i.e. bicubical sets over the binary cartesian cube category $\boxslash_2$ and the nullary affine cube category $\square_0$. The latter is also the base category of the Schanuel sheaf topos [27, §6.3], which is in turn equivalent to the category of nominal sets [27], used to model FreshMLTT [28]. FreshML [33] is modelled in the Fraenkel-Mostowski model of set theory, which inspired the development of nominal sets. Schöpp and Stark's bunched system [32] is modelled in a class of categories, including the Schanuel topos, that are locally cartesian closed and also equipped with a semicartesian closed structure. Finally, $\lambda^{\Pi\mathcal{U}}$ [11] has a syntactic soundness proof.

$\square_0$ is a category of cubes without diagonals, so the internal language of its associated presheaf topos is among the first candidates to get a transpension type [21] with workable typing rules [19]. 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. The relationship between Gel and transpension is explained in more generality in [21].

Regarding the implementation of PNTT, given that the binary CH theory was implemented in [36] on top of Cubical Agda [37], implementing PNTT (essentially nullary CH + name induction) is feasible. We think that such an implementation would be usable but not particularly scalable by default to perform parametricity proofs. The issue is that such proofs require the SAP, which cannot be proved as a single internal theorem (see Section 4.2 and [36]). A practical solution could be to automate the process of building SAP proofs. Alternatively, one could instead build PNTT on top of *observational* [2] nullary PTT, where the SAP is not needed since a first-class recursive translation attempts to replace the Bridge type. The Narya proof assistant in progress [18] implements observational n-ary PTT (and more).

References

1 C. B. Aberlé. Parametricity via cohesion. *CoRR*, abs/2404.03825, 2024. 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, 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.

7 Jesper Bengtson and Joachim Parrow. Formalising the pi-calculus using nominal logic. *Log. Methods Comput. Sci.*, 5(2), 2009. arXiv:0809.3960.

8 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. doi:10.1016/J.ENTCS.2015.12.006.

9 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.

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

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

12 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.

13 Martin Hofmann. *Syntax and semantics of dependent types*, pages 13–54. Springer London, London, 1997. doi:10.1007/978-1-4471-0963-1_2.

14 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.

15 Martin Hofmann and Thomas Streicher. Lifting grothendieck universes. Unpublished note, 1997.

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

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

20 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.

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

22 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.

23 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.

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

25 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.

26 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.

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

28 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. doi:10.1016/J.ENTCS.2015.04.003.

29 Sarah Reboullet. *Universe discrepancies in nominal presheaf models of parametric type theory*. Theses, Université Paris Cité, September 2025. URL: https://theses.hal.science/tel-05584055.

30 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.

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

32 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.

33 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.

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

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

36 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.

37 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.

38 Philip Wadler. Theorems for free! In Joseph E. Stoy, editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359. ACM, 1989. doi:10.1145/99370.99404.