Native Type Theory

Christian Williams
University of California, Riverside, US

Michael Stay
Pyrofex Corporation, Utah, US

Abstract

We present a method to construct "native" type systems for a broad class of languages, in which types are built from term constructors by predicate logic and dependent types. Many languages can be modelled as structured λ-theories, and the internal language of their presheaf toposes provides total specification the structure and behavior of programs. The construction is functorial, thereby providing a shared framework of higher-order reasoning for most existing programming languages.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Type Theory, Category Theory, Logic, Topos Theory, Concurrency

Related Version arxiv:2102.04672 (previous)

Acknowledgements We thank John Baez, Greg Meredith, and Mike Shulman for helpful dialogue.

1 Introduction

Type theory is growing as a guiding philosophy in the design of programming languages. In practice, however, type systems are mostly heterogeneous, and there are not standard ways to reason across languages. We present a simple way to enrich a language with its own "internal logic": we construct from a λ-theory its *native type system*, which provides total specification of the structure and behavior of programs.

Categorical logic unifies languages: virtually any formalism, from a simple heap to the calculus of constructions, can be modelled as a structured category [20]. By doing so, we inherit a wealth of tools from category theory. In particular, we can generate expressive type systems by composing two known ideas.

$$\lambda\mathtt{theory} \xrightarrow{\mathcal{P}} \mathtt{topos} \xrightarrow{\mathcal{L}} \mathtt{type\ system}$$

The first is the *presheaf construction* $\mathcal{P}$ [10, Ch. 8]; it preserves product, equality, and function types. The second is the *language of a topos* $\mathcal{L}$ [20, Ch. 11]. The composite is 2-functorial, so that translations between languages induce translations between type systems.

The type system is *native* in the sense that types are built only from term constructors, predicate logic, and a form of dependent type theory. For example, the following predicate on processes in a concurrent language (ex. 5) is effectively a compile-time firewall.

$$\mathsf{sole.in}(\alpha) \;:=\; \nu\mathtt{X}.\ (\mathtt{in}(\alpha, \mathtt{N} \to \mathtt{X}) \mid \mathtt{P}) \wedge \neg[\mathtt{in}(\neg[\alpha], \mathtt{N} \to \mathtt{P}) \mid \mathtt{P}]$$

Can input on channels in type α and cannot input on $\neg\alpha$, and continues as such.

Native type theory is intended to be a practical method to equip programming languages with a shared system of higher-order reasoning. The authors believe that the potential applications are significant and broad, and we advocate for community development.

1.1 Motivation and implementation

As software systems become increasingly complex, it is critical to develop adequate frameworks for reasoning about code. By generating expressive type systems for programming languages, native type theory can improve control, reasoning, and communication of systems.

For example, web browsers use the dynamic, weakly-typed language of JavaScript. Companies have recognized that correct and maintainable code requires static type checking. Microsoft's TypeScript [6], Facebook's Flow [1], and Google's Closure Compiler [2] are multi-million dollar efforts to retrofit JavaScript with a strong, static type system; yet none of these is sound. When presented as a structured λ-theory [5], JavaScript has a native type system which is sound by construction.

We aim to implement native type theory as a development environment, based on a library of formal semantics and translations, for programming in languages enriched by their native type systems. One can then write code with higher-order logic and dependent types, both to condition existing codebases and to expand software capability.

To this end, we plan to leverage progress in language specification. K Framework [4] is a formal verification tool which is used to give complete semantics of many popular languages, including JavaScript, C, Java, Python, Haskell, LLVM, Solidity, and more. These specifications can be presented as *structured λ-theories* (§2), and input to native type theory.

The type system generated can then be used for many purposes, e.g. to query codebases. The search engine Hoogle [3] queries Haskell libraries by function signature. This idea can be expanded to many languages and strengthened by more expressive types. If $\varphi : \mathtt{S} \to \mathsf{Prop}$ is a predicate on $\mathtt{S}$-terms and $\psi : \mathtt{T} \to \mathsf{Prop}$ is one on $\mathtt{T}$-terms, e.g. a security property, we can form the type of programs $\mathtt{S} \to \mathtt{T}$ for which substituting φ entails ψ (§3.1, def. 13).

$$[\varphi, \psi] := \{\lambda x.c : \mathtt{S} \to \mathtt{T} \mid \forall p : \mathtt{S}.\ \varphi(p) \Rightarrow \psi(c[p/x])\}$$

Of course, the full applications of native type systems require substantial development. Most basic is the need for efficient type-checking, but this is well-studied [34]. For usability we need to convert between existing types and native types, as well as libraries of native types, so programmers can express useful ideas without overly complex formulae.

The larger endeavor, to create a framework for reasoning across many languages, calls for developing a public library of both formal semantics and translations between languages.

1.2 Organization and contribution

Our goal is to demonstrate that composing two categorical ideas can be highly useful to computer science. In the process we emphasize many ideas that may be "known" in theory but are not widely known nor used in practice.

§2 **Structured λ-theories**. We define λ-theories with equality as cartesian closed categories with pullbacks, and interpret the internal language as simply-typed λ-calculus combined with the syntax of generalized algebraic theories [15].

Rewriting systems can be presented as internal categories; this motivates the 2-category of *structured λ-theories*. In §A these are used to demonstrate a translation of λ-calculus into π-calculus which respects their operational semantics. We define the $\rho\pi$-calculus, a concurrent language with reflection, as our running example for native types.

§3 **Logic in a presheaf topos**. A λ-theory T embeds into a presheaf topos $\mathcal{P}(\mathsf{T})$, and we develop its internal language. Predicates on the sorts of T form a λ-theory $\omega\mathsf{T}$ which refines the entire language; refined binding is then applied to condition program input (§5.2).

We show that the predicate and codomain fibrations of $\mathcal{P}(\mathsf{T})$ form a "cosmic" *higher-order dependent type theory* (HDT), and this construction is 2-functorial.

Hence *native type theory* is the composite 2-functor

$$\lambda\mathrm{Thy}_{=}^{\mathsf{op}} \xrightarrow{\mathcal{P}} \mathrm{Topos} \xrightarrow{\mathcal{L}} \mathrm{HDT}\Sigma.$$

This extends to structured λ-theories, i.e. the arrow 2-categories over this composite. Monads and comonads are preserved by 2-functors, in particular Moggi's *notions of computation* [31].

§4 **Native type theory**. The native type system of a λ-theory T is presented as the internal language of the presheaf topos, $\mathcal{LP}(\mathsf{T})$. The system is an extension of *higher-order dependent type theory* [20], as in the Calculus of Constructions [16]. We present the system as generated by T, and give the rules for types and terms, as well as those for functoriality.

§5 **Applications**. We explore a few kinds of applications: conditioning term behavior, with subgraphs of rewrite systems and modalities, and deriving behavioral equivalence; conditioning program input with refined binding, and reasoning about contexts with predicate homs; and translating types across programming paradigms.

The scope of applications is beyond what can be given here.

§A **Appendix**. We give an overview of related work, including the project origin.

2 Structured λ-theories

Simply-typed λ-calculus is the language of products and functions. It is regarded as the foundation of computer science [12] and much of modern programming [18].

The syntax of a language can be modelled by a *syntactic category*, in which an object is a sorted variable context, a morphism is a term constructor, and composition is substitution. Simply-typed λ-calculus is the language of *cartesian closed categories* [22].

A particular λ-calculus or λ-*theory* is presented by sorts, constructors, and equations. This is just like algebraic presentation, but with higher-order constructors. Good references for the syntax and semantics of simply-typed λ-calculus are [17, Ch. 4] and [20, Ch. 2]. We denote products $\mathtt{S} \times \mathtt{T}$ by $\mathtt{S}, \mathtt{T}$ and functions $[\mathtt{S}, \mathtt{T}]$ by $[\mathtt{S} \to \mathtt{T}]$.

$$\frac{\Gamma, x{:}\mathtt{S} \vdash t : \mathtt{T}}{\Gamma \vdash \lambda x.t : [\mathtt{S} \to \mathtt{T}]} \text{ abstraction} \qquad \frac{\Gamma \vdash \lambda x.t : [\mathtt{S} \to \mathtt{T}], u : \mathtt{S}}{\Gamma \vdash t[u/x] : \mathtt{T}} \text{ application}$$

▶ **Definition 1.** *A* **λ-theory with equality** *is a cartesian closed category with pullbacks, also known as a "properly cartesian closed category" [21]. The 2-category of λ-theories with equality, finitely continuous closed functors, and cartesian natural transformations is* $\lambda\mathrm{Thy}_=$.

The syntax of a λ-theory with equality can be derived from its subobject fibration having fibered equality [20, Ch. 3]. We interpret the language as simply-typed λ-calculus combined with the syntax of *generalized algebraic theories* [15], which provide *indexed sorts.*

$$\frac{\Gamma \vdash x_1 : \mathtt{S}_1, \ldots, x_n : \mathtt{S}_n}{\Gamma, \vec{x_i} : \vec{\mathtt{S}_i} \vdash \mathtt{A}(x_1, \ldots, x_n) \ \ \mathsf{sort}} \text{ sort symbol} \qquad \frac{\Gamma \vdash s_1 : \mathtt{S}_1, \ldots, s_n : \mathtt{S}_n}{\Gamma \vdash \mathtt{f}(s_1, \ldots, s_n) : \mathtt{S}} \text{ term symbol}$$

These are presented in the same way as λ-theories, plus constructors which may be parameterized by equations, such as composition in the theory of categories. This is our motivation: we represent behavior of terms using internal categories.

Henceforth, "λ-theory" means λ-theory with equality.

λ-theories with structure

What λ-theories do not explicitly represent is the *process* of computation. In practice, computing consists not of equations but transitions. There are many ways to model the behavior of languages [37], but the operational semantics of higher-order languages is still in development [19]. We introduce a method of representing behavior internally.

A language with a rewrite system can be modelled by a λ-theory T equipped with an internal category, which includes constructors and equations to specify the interaction between rewrites and constructors, such as forming a congruence.

▶ **Definition 2.** Th.Cat

$$\begin{array}{lllll} \mathtt{Hom}: \; \mathtt{E} \to \mathtt{V},\mathtt{V} & ;_{abc}: & \mathtt{Hom}(a,b),\mathtt{Hom}(b,c) \to \mathtt{Hom}(a,c) & (e_1;e_2);e_3 & = \; e_1;(e_2;e_3) \\ & \mathtt{id}_a: & 1 \to \mathtt{Hom}(a,a) & \mathtt{id}_a;e = e & e;\mathtt{id}_b = e \end{array}$$

Given $e : \Gamma \to \mathtt{E}$ and $a, b : \Gamma \to \mathtt{V}$ we denote $e : \mathtt{Hom}(a,b)$ by $e(\vec{x}) : a(\vec{x}) \rightsquigarrow b(\vec{x})$.

Though composition is useful, we often want to reason about "basic rewrites" or single-step computations. For most of the paper we will simply use an internal graph. It is easy to combine both approaches, by distinguishing one sort for edges and one sort for morphisms.

To specify how basic rewrites interact with constructors, we can take the source map $s : \mathtt{E} \to \mathtt{S}$ as a sort symbol $\mathtt{S}^*(x)$. Then $\mathtt{S}^*(v)$ are the rewrites with source v, i.e. the *behaviors* of the term. This allows us to define operational semantics.

▶ **Definition 3.** *A* **rewrite rule** *for a term constructor* $\mathtt{f} : \prod \mathtt{S}_i \to \mathtt{S}$ *in λ-theory* $\mathsf{T}(\mathtt{E}_\mathtt{S}, \{\mathtt{E}_{\mathtt{S}_i}\})$ *is specified by an edge constructor*

$$\mathsf{R}(\mathtt{f})_{\vec{v}} : \prod \mathtt{S}_i^*(v_i) \to \mathtt{S}^*(\mathtt{f}(\vec{v})) \;\; \textit{such that} \;\; \mathsf{R}(\mathtt{f})(\langle v_1, e_1\rangle, \dots, \langle v_n, e_n\rangle) : \mathtt{f}(\vec{v}) \rightsquigarrow \mathtt{g}$$

where $\mathtt{g} : \prod \mathtt{S}_i^*(v_i) \to \mathtt{S}$. *An* **operational semantics** O *is a set of basic rewrites* $\{\mathtt{r}_i(\vec{x}) : \mathtt{a}_i(\vec{x}) \rightsquigarrow \mathtt{b}_i(\vec{x}) : \mathtt{E}_{\mathtt{S}_i}\}$ *together with a family of pairs* $\{(\mathtt{f}_{ij}, \mathsf{R}(\mathtt{f}_{ij}))\}$.

▶ **Lemma 4.** *These operational semantics correspond to the class of GSOS rules [37] for deterministic labelled transition systems. Nondeterministic systems can be derived using an internal relation* $\mathtt{act} \rightarrowtail \mathtt{V},\mathtt{A},\mathtt{V}$.

By representing behavior internally, *native type systems reason about both the structure and behavior of programs.* For example there is a predicate for "contexts $\lambda x.c : \mathtt{S} \to \mathtt{T}$ such that if $a : \mathtt{S}$ satisfies φ then for all $e : c[a/x] \rightsquigarrow b$ if $\psi(b)$ then no step of e satisfies ϵ".

▶ **Example 5.** ρπ-calculus Th.ρπ (polyadic)

$$\begin{array}{llllll} \mathtt{0}: & 1 \to \mathtt{P} & -|-: & \mathtt{P},\mathtt{P} \to \mathtt{P} & (\mathtt{P}, -|-, \mathtt{0}) & \text{commutative monoid} \\ @: & \mathtt{P} \to \mathtt{N} & \mathtt{out}_k: & \mathtt{N},\mathtt{P}^k \to \mathtt{P} & \mathtt{run}: & \mathtt{P} \to \mathtt{E} \\ *: & \mathtt{N} \to \mathtt{P} & \mathtt{in}_k: & \mathtt{N},[\mathtt{N}^k \to \mathtt{P}] \to \mathtt{P} & \mathtt{comm}_k: & \mathtt{N},\mathtt{P}^k,[\mathtt{N}^k \to \mathtt{P}] \to \mathtt{E} \end{array}$$

$$\mathtt{run}(p) : *(@p) \rightsquigarrow p \qquad \mathtt{comm}_k(n, \vec{q_i}, \lambda\vec{x_i}.p) : \mathtt{out}(n, \vec{q_i})|\mathtt{in}(n, \lambda\vec{x_i}.p) \rightsquigarrow p[@q_i/x_i]$$

$$\begin{array}{lll} \mathsf{Th.Cat} \;+\; — & \mathtt{par}_l : \mathtt{E},\mathtt{P} \to \mathtt{E} & \mathtt{par}_l(\rho, q) : s(\rho)|q \rightsquigarrow t(\rho)|q \\ \mathtt{par}_r : \mathtt{P},\mathtt{E} \to \mathtt{E} & \mathtt{par}_r(p,\rho) = \mathtt{par}_l(\rho,p) & \mathtt{par}_l \;\; \text{c. monoid action of } \mathtt{P} \text{ on } \mathtt{E} \end{array}$$

The ρπ-calculus or **r**eflective **h**igher-**o**rder π-calculus [28] is a concurrent language succeeding the π-calculus [29]. It is the language of the blockchain platform RChain [7].

The ρπ-calculus has processes $\mathtt{P}$ and names $\mathtt{N}$, which act as code and data respectively; reference @ and execute $*$ transform one into the other. Terms are built up from the null process $\mathtt{0}$ by parallel $-|-$, output $\mathtt{out}$, and input $\mathtt{in}$. The basic rule is communication $\mathtt{comm}$: an output and input process connect on a name and transfer a list of processes as data.

The ρπ-calculus is our running example. In the native type system of Th.ρπ (§3.1), a predicate on names $\alpha : y(\mathtt{N}) \to \mathsf{Prop}$ is called a *namespace* [27], and a predicate on processes $\varphi : y(\mathtt{P}) \to \mathsf{Prop}$ is called a *codespace*.

Hence operational semantics can be specified by $\mathsf{Th.Cat} \to \mathsf{T}$, and translations ought to respect this structure. We generalize to define "structure" as any λ-theory morphism into T.

▶ **Definition 6.** *A* **structured λ-theory** *is a λ-theory with equality T equipped with a morphism $\tau : \mathsf{S} \to \mathsf{T}$ in $\lambda\mathrm{Thy}_=$. The 2-category of structured λ-theories is the (strict) arrow 2-category $[\mathrm{I}, \lambda\mathrm{Thy}_=]$, where I is the interval category.*

By distinguishing behavior as internal structure, we can ensure that translations induce the proper homomorphisms of rewriting systems. In §A we exhibit a translation of the name-passing λ-calculus into the π-calculus which respects their operational semantics.

While behavior is our primary example of structure, the concept is very general. Some other examples are *sorting*, e.g. refining the $\rho\pi$-calculus with sorted channels to send and receive certain kinds of data; *embedding* a language into a networked environment; or *encoding* the programs of one language into another.

Because native type theory is functorial, the structure $\tau : \mathsf{S} \to \mathsf{T}$ translates types of T into types of S. For including behavior, this simply distinguishes the "behavioral" types; for more complex structures, the translation may be highly expressive.

We note that the 2-category of structured λ-theories is naturally indexed over the 2-category of λ-theories. Just as an arrow category $[\mathrm{I}, C]$ has co/domain op/fibrations over C, an arrow 2-category is equipped with 2-op/fibrations [14].

▶ **Proposition 7.** *The 2-category of structured λ-theories is 2-op/fibered over λ-theories, by the domain and codomain 2-functors $\delta_0, \delta_1 : [\mathrm{I}, \lambda\mathrm{Thy}_=] \to \lambda\mathrm{Thy}_=$.*

Because $\lambda\mathrm{Thy}_=$ has pushouts and pullbacks, δ_0 and δ_1 are in fact bifibrations. However, it is not locally cartesian closed [33]; though this may be true for complete CCCs.

The domain fiber $\lambda\mathrm{Thy}_0(\mathsf{S}) := \delta_0^*(\mathsf{S})$ is the 2-category of S-structured λ-theories. The codomain fiber $\lambda\mathrm{Thy}_1(\mathsf{T}) := \delta_1^*(\mathsf{T})$ is the 2-category of structures on T.

From a structured λ-theory we derive a native type system, using the presheaf construction, and demonstrate how it can be used to reason about the structure and behavior of terms.

3 The Logic of a Presheaf Topos

Topos theory [23] expands the domain of predicate logic and intuitionistic type theory [25] beyond sets and functions. Most useful is the fact that every category embeds into a topos. For any λ-theory, the internal language of its presheaf topos is its native type system.

Let T be a λ-theory. The category of *presheaves* is the functor category $[\mathsf{T}^{\mathsf{op}}, \mathrm{Set}]$, which we denote $\mathcal{P}(\mathsf{T})$. This defines a 2-functor to elementary toposes and geometric morphisms

$$\mathcal{P} : \lambda\mathrm{Thy}_=^{\mathsf{op}} \to \mathrm{Topos} \qquad \mathcal{P}(\mathsf{F}) = (\exists_{\mathsf{F}} \dashv \mathsf{F}^*) : [\mathsf{T}^{\mathsf{op}}, \mathrm{Set}] \to [\mathsf{S}^{\mathsf{op}}, \mathrm{Set}].$$

where $\exists_{\mathsf{F}}$ is left Kan extension and F^* is precomposition by $\mathsf{F} : \mathsf{S} \to \mathsf{T}$.

A presheaf is a context-indexed set of data on the sorts of a theory. The canonical example is a *representable* presheaf, of the form $\mathsf{T}(-, \mathtt{S})$, which indexes all terms of sort $\mathtt{S}$. The Yoneda embedding $y : \mathsf{T} \to \mathcal{P}(\mathsf{T}) :: \mathtt{S} \mapsto \mathsf{T}(-, \mathtt{S})$ preserves limits and internal homs.

A *subobject classifier* is an object Ω with a natural isomorphism $\mathrm{c} : [-, \Omega] \simeq \mathrm{Sub}(-)$. We may denote Ω as Prop; this is its role in the type system: a *predicate* is a morphism $\varphi : A \to \Omega$, and the *comprehension* of φ is the subobject $\mathrm{c}(\varphi) := \{a : A \mid \varphi(a)\} \rightarrowtail A$.

A **topos** is a λ-theory with equality with a subobject classifier. For presheaves, the hom and subobject classifier are defined $[P, Q](\mathtt{S}) = \mathcal{P}(\mathsf{T})(y(\mathtt{S}) \times P, Q)$ and $\Omega(\mathtt{S}) = \{\varphi \rightarrowtail y(\mathtt{S})\}$.

The values of Ω can be understood as $\Omega(\mathtt{S}) \simeq \{\text{sieves of sort } \mathtt{S}\}$. A *sieve* of sort $\mathtt{S}$ is a set of terms of sort $\mathtt{S}$ that is closed under substitution. A simple example is a *principal sieve* $\langle \mathtt{f} \rangle : \Omega(\mathtt{T})$ generated by a term $\mathtt{f} : \mathtt{S} \to \mathtt{T}$, defined $\langle \mathtt{f} \rangle(\mathtt{R}) := \Sigma u{:}\mathtt{R} \to \mathtt{S}.\mathtt{f} \circ u$.

▶ **Example 8.** The $\rho\pi$-calculus (ex. 5) can express recursion without the replication operator of the π-calculus. On a name $n : 1 \to \mathtt{N}$ we define a context which replicates processes.

$$c(n) \quad := \mathtt{in}(n, \lambda x.\{\mathtt{out}(n, *x) \mid *x\}) \qquad !(-)(n) \quad := \mathtt{out}(n, \{c(n)|-\}) \mid c(n).$$

One can check that $!(p)(n) \quad \rightsquigarrow \; !(p)(n) \mid p$ for any process p. The sieve $\langle !(-)(n) \rangle : \Omega(\mathtt{P})$ consists of processes which replicate on the name n by the above method.

For simpler formulae, we denote the values of a presheaf by $A_\mathtt{S} := A(\mathtt{S})$, and the action of $u : \mathtt{R} \to \mathtt{S}$ by $- \cdot u := A(u) : A(\mathtt{S}) \to A(\mathtt{R})$. For $\varphi : A \to \mathsf{Prop}$ we denote $\varphi^a_\mathtt{S} := \varphi(\mathtt{S})(a)$; more generally for any $p : P \to A$ we denote $p^a_\mathtt{S} := p_\mathtt{S}^{-1}(a)$ as the *fiber* over a (§3.2).

3.1 The predicate fibration

There is a category over $\mathcal{P}(\mathsf{T})$ for which the fiber over each presheaf is the complete Heyting algebra (CHA) of its predicates. Quantification gives change-of-base adjoints between fibers; we show that moreover the domain is cartesian closed, complete and cocomplete. The fibration encapsulates higher-order predicate logic.

We henceforth use Ω^A to denote the complete Heyting algebra of predicates. The *predicate functor* of $\mathcal{P}(\mathsf{T})$ is defined $\Omega^{(-)} : \mathcal{P}(\mathsf{T})^{\mathsf{op}} \to \mathrm{CHA}$. For $f : A \to B$, precomposition of predicates corresponds to preimage of subobjects. This can be written as substitution $\varphi[f] := \Omega^f(\varphi)$, and understood as *pattern-matching.*

▶ **Example 9.** For a $\rho\pi$-calculus predicate $\varphi : y(\mathtt{P}) \to \mathsf{Prop}$, substitution by $\mathtt{in} : \mathtt{N} \times [\mathtt{N}, \mathtt{P}] \to \mathtt{P}$ is the basic query "inputting on what name-context pairs yield property φ?"

$$\varphi[y(\mathtt{in})]_\mathtt{S} = \{\mathtt{S} \vdash (n, \lambda x.p) : \mathtt{N}, [\mathtt{N} \to \mathtt{P}] \mid \varphi(\mathtt{in}(n, \lambda x.p))\}$$

The complete Heyting algebra structure of Ω^A: predicates are ordered by entailment, meet and join are defined by pointwise intersection and union, $\top = A$ and $\bot = (\mathtt{S} \mapsto \emptyset)$, implication is defined $(\varphi \Rightarrow \psi)^a_\mathtt{S} := \prod_{u:\mathtt{R}\to\mathtt{S}} \varphi^{a\cdot u}_\mathtt{R} \Rightarrow \psi^{a\cdot u}_\mathtt{R}$, and negation is $\neg(\varphi) := (\varphi \Rightarrow \bot)$.

We can assemble the image of $\Omega^{(-)}$ into one category with the Grothendieck construction.

▶ **Definition 10.** *The* category of predicates *of* $\mathcal{P}(\mathsf{T})$ *is denoted* $\Omega\mathcal{P}(\mathsf{T})$*: an object is a pair* $\langle A : \mathcal{P}(\mathsf{T}) \;,\; \varphi : \Omega^A \rangle$*, and a morphism is a pair* $\langle f : A \to B \;,\; \varphi \Rightarrow \Omega^f(\psi) \rangle$*. The projection* $\pi_\Omega : \Omega\mathcal{P}(\mathsf{T}) \to \mathcal{P}(\mathsf{T})$ *is the* **predicate fibration***; the fiber over* A *is* Ω^A*, and the fiber over* $f : A \to B$ *is* $\Omega^f : \Omega^B \to \Omega^A$*, known as a* change-of-base functor*.*

A fibration is a functor with a well-behaved notion of preimage, used in type theory for *indexing*; a reference is [20, Ch. 1]. The predicate fibration is highly structured; each change-of-base functor has adjoints which give dependent sum and product.

▶ **Proposition 11.** $\pi_\Omega : \Omega\mathcal{P}(\mathsf{T}) \to \mathcal{P}(\mathsf{T})$ *has* **indexed sums and products** *[20]: for each* $f : A \to B$*, the functor* $\Omega^f : \Omega^B \to \Omega^A$ *has left and right adjoints* $\exists_f \dashv \Omega^f \dashv \forall_f$*.*

$$\exists_f(\varphi)^b_\mathtt{S} := \quad \Sigma(a{:}A_\mathtt{S}).\Sigma(f_\mathtt{S}(a) = b).\varphi(a) \qquad \forall_f(\varphi)^b_\mathtt{S} := \quad \Pi(u{:}\mathtt{R} \to \mathtt{S}).\Pi(f_\mathtt{R}(a) = b \cdot u).\varphi(a)$$

The left adjoint $\exists_f$ is called **direct image**, because on subobjects it is composition by f; we call the right adjoint $\forall_f$ **secure image**. While Ω^f is a morphism of complete Heyting algebras, $\exists_f$ and $\forall_f$ are only morphisms of join and meet semilattices, respectively.

▶ **Example 12.** Let $\mathsf{Th.Gph} \to \mathsf{T}$ be a λ-theory with a graph, and $\varphi : y(\mathtt{V}) \to \mathsf{Prop}$ be a predicate on terms. Then $\varphi[y(s)] : y(\mathtt{E}) \to \mathsf{Prop}$ are rewrites with $\varphi(\text{source})$, and $\exists_{y(t)}(\varphi[y(s)])$ are the targets of these rewrites. Hence there is a *step-forward* $\mathsf{F}_! : [y(\mathtt{V}), \mathsf{Prop}] \to [y(\mathtt{V}), \mathsf{Prop}]$.

The *secure step-forward* is a more refined operation: $\mathsf{F}_*(\varphi) := \forall_{y(t)}(\varphi[y(s)])$ are terms u for which $(t \rightsquigarrow u) \Rightarrow \varphi(t)$. For security protocols, this can filter agents by past behavior.

The change-of-base adjoints satisfy the *Beck–Chevalley condition*: this means that quantification commutes with substitution, and implies that $\Omega^{(-)} : \mathcal{P}(\mathsf{T})^{\mathsf{op}} \to \mathrm{CHA}$ is a *first-order hyperdoctrine* [24] and a **higher-order fibration** [20, section 5.3].

This concept leaves implicit additional structure: there is an *internal hom* of predicates.

▶ **Proposition 13.** *$\Omega\mathcal{P}(\mathsf{T})$ is cartesian closed, as is π_Ω. Let $\varphi : A \to \mathsf{Prop}$, $\psi : B \to \mathsf{Prop}$, and let $\langle \pi_1, \pi_2, ev\rangle : A \times [A,B] \to A \times [A,B] \times B$. Then $[\varphi, \psi] : [A,B] \to \mathsf{Prop}$ is defined $[\varphi,\psi] := \forall_{\pi_2}(\varphi[\pi_1] \Rightarrow \psi[ev])$.*

The cartesian closed structure of $\Omega\mathcal{P}(\mathsf{T})$ is significant, because the category of predicates on T is itself a λ-theory, the refinement of the language. We explore applications in §5.2.

▶ **Definition 14.** *The* **predicate theory** *of T, denoted $\omega\mathsf{T}$, is the pullback of the predicate fibration along the embedding $y : \mathsf{T} \to \mathcal{P}(\mathsf{T})$; it is a λ-theory fibered over T.*

Note. We emphasize the idea of having "lifted" the language by an abuse of notation: for any operation $\mathtt{f} : \mathtt{S} \to \mathtt{T}$, we may denote $\exists_{y(\mathtt{f})} : [y(\mathtt{S}), \mathsf{Prop}] \to [y(\mathtt{T}), \mathsf{Prop}]$ simply by $\mathtt{f}$, and $\forall_{y(\mathtt{f})}$ by $\mathtt{f}_*$. Similarly, we may write $y(\mathtt{S})$ as $\mathtt{S}$, when the context is clear.

▶ **Example 15.** As an example of contexts which ensure implications across substitution, we can construct the "magic wand" of separation logic [26]. Let T_h be the theory of a commutative monoid $(H, \cup, e)$, plus constructors for the elements of a heap. If we define $(\varphi \mathrel{-\!\!*} \psi) := [\varphi, \psi][\lambda x.x \cup -]$, then $(\varphi \mathrel{-\!\!*} \psi)(h_1)$ means that $\varphi(h_2) \Rightarrow \psi(h_1 \cup h_2)$.

There is a more expressive way to form hom predicates, which provides *predicate binding*.

▶ **Proposition 16.** *Let $A, B : \mathcal{P}(\mathsf{T})$, and let $\mathsf{L}_{A,B} : [[A,B], \mathsf{Prop}] \to [[A, \mathsf{Prop}], [B, \mathsf{Prop}]]$ be curried evaluation. There is a right adjoint which we call* **reification***. The predicate $\mathsf{R}_{A,B}(F)$, denoted $\chi.F$, determines $f : [A,B]$ whose images are contained in those of F:*

$$[\chi.F]^f_{\mathtt{S}} \;=\; \Pi\chi{:}[A \to \mathsf{Prop}].\ \exists_f(y(\mathtt{S}) \times \chi) \Rightarrow F(\chi).$$

Using reification, separation logic can be generalized from pairs of predicates to *functions* of predicates. We are not aware if this has been studied.

In addition, the category of predicates has all limits and colimits, by a result of [36]. These can be used to form modalities, inductive and coinductive types, and more.

▶ **Proposition 17.** *$\Omega\mathcal{P}(\mathsf{T})$ is complete and cocomplete, and π_Ω preserves limits and colimits. They are computed pointwise; letting π, ι represent the cone and cocone:*

$$\lim_i\langle A_i, \varphi_i\rangle = \langle \lim_i(A_i), \lim_i(\Omega^{\pi_i}\varphi_i)\rangle \qquad \mathrm{colim}_i\langle A_i, \varphi_i\rangle = \langle \mathrm{colim}_i(A_i), \mathrm{colim}_i(\Sigma_{\iota_i}\varphi_i)\rangle.$$

To summarize the rich structure present, we allude to a term from category theory: a *cosmos* is a monoidal closed category which is complete and cocomplete [35].

▶ **Proposition 18.** *The predicate fibration $\pi_\Omega : \Omega\mathcal{P}(\mathsf{T}) \to \mathcal{P}(\mathsf{T})$ is a higher-order fibration which is* **cosmic***: cartesian closed, complete and cocomplete.*

3.2 The codomain fibration

Predicates $\varphi : A \to \mathsf{Prop}$ correspond to subobjects $\mathrm{c}(\varphi) \rightarrowtail A$. More generally, any $p : P \to A$ can be understood as a *dependent type*. Like subsets to indexed sets, this expands the fibers over A from truth values to sets, and the fibers over $\mathcal{P}(\mathsf{T})$ from posets to categories.

▶ **Proposition 19.** *Let* CCT *be the category of co/complete toposes and logical functors. There is a functor* $\Delta : \mathcal{P}(\mathsf{T})^{\mathsf{op}} \to \mathrm{CCT}$ *that maps* A *to* $\mathcal{P}(\mathsf{T})/A$ *and* $f : A \to B$ *to pullback.*

We can denote pullback by substitution, $p[f]^a_{\mathsf{S}} := \Delta^f(p)^a_{\mathsf{S}} = p^{f_{\mathsf{S}}(a)}_{\mathsf{S}}$. Dependent sum Σ_f and dependent product Π_f are given by the same formulae as those for predicates, and these satisfy the Beck-Chevalley condition. The Grothendieck construction of Δ determines a category over $\mathcal{P}(\mathsf{T})$.

▶ **Definition 20.** *The* category of dependent types *of* $\mathcal{P}(\mathsf{T})$*, denoted* $\Delta\mathcal{P}(\mathsf{T})$*, is equivalent to the arrow category of* $\mathcal{P}(\mathsf{T})$*. The* **codomain fibration** *is the projection* $\pi_\Delta : \Delta\mathcal{P}(\mathsf{T}) \to \mathcal{P}(\mathsf{T})$.

▶ **Proposition 21.** *The codomain fibration* π_Δ *is a closed comprehension category [20, Sec 10.5] which is cosmic, i.e. cartesian closed, complete and cocomplete.*

The two fibrations are connected by an adjunction $\mathrm{c} \dashv \mathrm{i} : \pi_\Delta \leftrightarrows \pi_\Omega$: comprehension interprets a predicate as a dependent type, and factorization takes a dependent type to its image predicate. This fibered adjunction is a *higher-order dependent type theory* [20, Sec. 11.6]. These form a sub-2-category of adjunctions in the 2-category of fibrations.

Geometric morphisms of toposes preserve pullbacks, inducing morphisms of predicate and codomain fibrations. But they are not locally cartesian closed, nor do they preserve the subobject classifier; it is future work to consider theory translations which induce locally connected morphisms of presheaf toposes [21, C 3.3].

We denote by $\mathrm{HDT}\Sigma$ the 2-category of higher-order dependent type theories and maps of adjunctions of fibrations.

▶ **Theorem 22.** *The construction which sends a topos to its* **internal language** $\mathcal{L}(\mathcal{E}) = \langle \pi_{\Omega\mathcal{E}}, \pi_{\Delta\mathcal{E}}, \mathrm{i}_{\mathcal{E}}, \mathrm{c}_{\mathcal{E}} \rangle$*, consisting of the predicate and codomain fibrations connected by the image-comprehension adjunction, defines a 2-functor* $\mathcal{L} : \mathrm{Topos} \to \mathrm{HDT}\Sigma$.

We note that 2-functors preserve monads and comonads, so the native types construction $\mathcal{LP} : \lambda\mathrm{Thy}^{\mathsf{op}}_{=} \to \mathrm{HDT}\Sigma$ extends to λ-theories equipped with "notions of computation" [31].

4 Native Type Theory

We present the **native type system** $\mathcal{LP}(\mathsf{T})$ of a λ-theory with equality T (§2). As $y : \mathsf{T} \to \mathcal{P}(\mathsf{T})$ is full and faithful, $\mathsf{L}\mathcal{P}(\mathsf{T})$ is a conservative extension of T.

The system is *higher-order dependent type theory* [20, Sec. 11.5] "parameterized" by T. We do not present Equality and Quotient types. We encode Subtyping, Hom, and Inductive types, which we use in applications.

The type system has **predicates** $\mathsf{x{:}\Gamma} \vdash \varphi : \mathsf{Prop}$ and **types** $\mathsf{x{:}\Gamma \vdash A : Type}$, interpreted as $\varphi : \Gamma \to \Omega$ and $\mathsf{p : A \to \Gamma}$. A term judgement is of the form $\mathsf{x{:}\Gamma, a{:}A \vdash N : B[M]}$, interpreted as a morphism $\langle \mathsf{M}, \mathsf{N} \rangle : (\mathsf{A} \to \Gamma) \to (\mathsf{B} \to \Delta)$ in the total category of the codomain fibration.

For details on the semantic interpretation of the type system, in particular handling coherence when interpreting substitution as pullback, see Awodey's *natural models* [11].

We present the type system as generated from the λ-theory T, so a programmer can start in the ordinary language and use the ambient logical structure as needed.

Y **Representables** are given in the type system as axioms.

$$\frac{[\![\mathtt{S} : \mathtt{T}]\!]}{\mathsf{y}\mathtt{S} : \mathsf{Type}}\,\mathsf{T}_S \qquad \frac{[\![\mathtt{S}_1 \vdash \mathtt{f} : \mathtt{S}_2]\!]}{\mathsf{x{:}y}\mathtt{S}_2 \vdash \mathsf{y}\mathtt{f} : \mathsf{Type}}\,\mathsf{T}_O \qquad \frac{[\![\mathtt{S}_1 \vdash \mathtt{f} = \mathtt{g} : \mathtt{S}_2]\!]}{\mathsf{x{:}y}\mathtt{S}_2 \vdash \mathsf{y}\mathtt{f} = \mathsf{y}\mathtt{g}}\,\mathsf{T}_E$$

The type $\mathsf{y}\mathtt{S}$ indexes all terms of sort $\mathtt{S}$. Because the Yoneda embedding preserves limits and internal hom, we have $\mathsf{y}(\mathtt{S}_1, \mathtt{S}_2) = (\mathsf{y}\mathtt{S}_1, \mathsf{y}\mathtt{S}_2)$ and $\mathsf{y}[\mathtt{S} \to \mathtt{T}] = [\mathsf{y}\mathtt{S} \to \mathsf{y}\mathtt{T}]$.

Σ **Dependent Pair** is an indexed sum generalizing existential quantification.

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma, \mathsf{x{:}A} \vdash \mathsf{B} : \mathsf{Type}}{\Gamma \vdash \Sigma \mathsf{x{:}A.B} : \mathsf{Type}}\,\Sigma_{\mathrm{F}} \qquad \frac{\Gamma \vdash \mathsf{a} : \mathsf{A} \qquad \Gamma \vdash \mathsf{u} : \mathsf{B}[\mathsf{a}/\mathsf{x}]}{\Gamma \vdash \langle \mathsf{a}, \mathsf{u}\rangle : \Sigma \mathsf{x{:}A.B}}\,\Sigma_{\mathrm{I}}$$

$$\frac{\Gamma, \mathsf{z}{:}\Sigma \mathsf{x{:}A.B} \vdash \mathsf{C} : \mathsf{Type} \qquad \Gamma, \mathsf{a{:}A}, \mathsf{u{:}B} \vdash \mathsf{Q} : \mathsf{C}[\langle \mathsf{a}, \mathsf{u}\rangle/\mathsf{z}]}{\Gamma, \mathsf{z} : \Sigma \mathsf{x{:}A.B} \vdash (\mathsf{z} \text{ as } \langle \mathsf{a}, \mathsf{u}\rangle \text{ in } \mathsf{Q}) : \mathsf{C}}\,\Sigma_{\mathrm{E}}$$

$$\begin{array}{lcll} \langle \mathsf{M}, \mathsf{N}\rangle \text{ as } \langle \mathsf{a}, \mathsf{u}\rangle \text{ in } \mathsf{Q} & = & \mathsf{Q}[\mathsf{M}/\mathsf{a}, \mathsf{N}/\mathsf{u}] & (\Sigma_\beta) \\ \mathsf{P} \text{ as } \langle \mathsf{a}, \mathsf{u}\rangle \text{ in } \mathsf{Q}[\langle \mathsf{a}, \mathsf{u}\rangle/\mathsf{z}] & = & \mathsf{Q}[\mathsf{P}/\mathsf{z}] & (\Sigma_\eta) \end{array}$$

Π **Dependent Function** is an indexed product generalizing universal quantification.

$$\frac{\Gamma \vdash \mathsf{A} : \mathsf{Type} \qquad \Gamma, \mathsf{x{:}A} \vdash \mathsf{B} : \mathsf{Type}}{\Gamma \vdash \Pi \mathsf{x{:}A.B} : \mathsf{Type}}\,\Pi_{\mathrm{F}} \qquad \frac{\Gamma, \mathsf{x{:}A} \vdash \mathsf{t} : \mathsf{B}}{\Gamma \vdash \lambda \mathsf{x{:}A.t} : \Pi \mathsf{x{:}A.B}}\,\Pi_{\mathrm{I}}$$

$$\frac{\Gamma \vdash \mathsf{f} : \Pi \mathsf{x{:}A.B} \qquad \Gamma \vdash \mathsf{u} : \mathsf{B}}{\Gamma \vdash \mathsf{f}(\mathsf{u}) : \mathsf{B}[\mathsf{u}/\mathsf{x}]}\,\Pi_{\mathrm{E}} \qquad \begin{array}{lcll} (\lambda \mathsf{x{:}A.t})(\mathsf{a}) & = & \mathsf{t}(\mathsf{a}) & (\Pi_\beta) \\ \mathsf{f} & = & \lambda \mathsf{x{:}A.f} & (\Pi_\eta) \end{array}$$

We derive existential $\exists$ from Σ and universal $\forall$ from Π by image factorization. The rest of predicate logic $\bot, \top, \vee, \wedge, \Rightarrow, \neg$ is also encoded in terms of Σ and Π.

{} **Comprehension** converts a predicate to the type of its satisfying terms. The rules which convert a type to its image predicate can be derived from Σ and Equality.

$$\frac{\Gamma, \mathsf{x{:}A} \vdash \varphi : \mathsf{Prop}}{\Gamma \vdash \{\mathsf{x{:}A} \mid \varphi\} : \mathsf{Type}}\,\mathsf{c}_{\mathrm{F}} \qquad \frac{\Gamma, \mathsf{x{:}A} \vdash \varphi : \mathsf{Prop} \qquad \Gamma \vdash \mathsf{M} : \mathsf{A} \qquad \Gamma \vdash \varphi[\mathsf{M}/\mathsf{x}]}{\Gamma \vdash \mathsf{i}(\mathsf{M}) : \{\mathsf{x{:}A} \mid \varphi\}}\,\mathsf{c}_{\mathrm{I}}$$

$$\frac{\Gamma \vdash \mathsf{N} : \{\mathsf{x{:}A} \mid \varphi\}}{\Gamma \vdash \mathsf{o}(\mathsf{N}) : \mathsf{A}}\,\mathsf{c}_{\mathrm{E}} \qquad \begin{array}{ll} \mathsf{o}(\mathsf{i}(\mathsf{M})) = \mathsf{M} & (\mathsf{c}_\beta) \\ \mathsf{i}(\mathsf{o}(\mathsf{N})) = \mathsf{N} & (\mathsf{c}_\eta) \end{array} \qquad \begin{array}{c} \Gamma_1, \mathsf{x{:}A}, \Gamma_2, \varphi \vdash \psi \\ \hline\hline \Gamma_1, \mathsf{a} : \{\mathsf{x{:}A} \mid \varphi\}, \Gamma_2[\mathsf{o}(\mathsf{a})/\mathsf{x}] \vdash \psi[\mathsf{o}(\mathsf{a})/\mathsf{x}] \end{array}\,\mathsf{c}^{\circ}_{\mathrm{E}}$$

⊆ **Subtyping** of predicates is defined $(\varphi \subseteq \psi) := \forall \mathsf{a{:}A}.\ \varphi(\mathsf{a}) \Rightarrow \psi(\mathsf{a})$.
→ **Hom type** (def. 13) of $\mathsf{A}_1 \vdash \mathsf{B}_1 : \mathsf{Type}$ and $\mathsf{A}_2 \vdash \mathsf{B}_2 : \mathsf{Type}$ is defined $\Pi \mathsf{x{:}A}_1.\mathsf{B}_1[\pi] \Rightarrow \mathsf{B}_2[\mathsf{ev}]$.
R **Reification** (def. 16) $\chi.\mathsf{F} : [\mathsf{A}, \mathsf{B}] \to \mathsf{Prop}$ is defined $\Pi \varphi{:}[\mathsf{A} \to \mathsf{Prop}].\varphi \Rightarrow \mathsf{F}(\varphi[-])$.
μ **Inductive type** of $\mathsf{F} : [\mathsf{A}, \mathsf{Prop}] \to [\mathsf{A}, \mathsf{Prop}]$: the least and greatest fixed points are defined $\mu\varphi.\mathsf{F}(\varphi) := \exists \varphi{:}[\mathsf{A}, \mathsf{Prop}].\ (\varphi \subseteq \mathsf{F}(\varphi)) \Rightarrow \varphi$ and $\nu\varphi.\mathsf{F}(\varphi) := \forall \varphi{:}[\mathsf{A}, \mathsf{Prop}].\ (\mathsf{F}(\varphi) \subseteq \varphi) \Rightarrow \varphi$. These are used to form data structures and modalities; we can generalize to W-types [30].

These rules constitute the native type system $\mathcal{LP}(\mathsf{T})$, abridged for a first presentation. By functoriality, translations of λ-theories induce translations of native type systems.

F **Translation** is given by precomposing types and "whiskering" terms.

$$\frac{[\![\mathsf{F} : \mathsf{T}_1 \to \mathsf{T}_2]\!] \qquad \Gamma \vdash \mathsf{A} : \mathsf{Type}_2}{\Gamma \circ \mathsf{F} \vdash \mathsf{A} \circ \mathsf{F} : \mathsf{Type}_1}\,\mathsf{F}_{\mathsf{Ty}} \qquad \frac{[\![\mathsf{F} : \mathsf{T}_1 \to \mathsf{T}_2]\!] \qquad \mathsf{x}{:}\Gamma, \mathsf{y{:}A} \vdash \mathsf{N} : \mathsf{B}[\mathsf{M}]}{\mathsf{x}{:}(\Gamma \circ \mathsf{F}), \mathsf{y}{:}(\mathsf{A} \circ \mathsf{F}) \vdash \mathsf{N} \cdot \mathsf{F} : (\mathsf{B} \circ \mathsf{F})[\mathsf{M} \cdot \mathsf{F}]}\,\mathsf{F}_{\mathsf{Tm}}$$

We include rules that $\mathsf{F}^* : \mathcal{P}(\mathsf{T}_2) \to \mathcal{P}(\mathsf{T}_1)$ is a functor which preserves substitution, dependent pair, and limits and colimits. To further research we leave the question of the rules for the colax preservation of Π and Prop, and the rules for the two covariant functors $\exists_\mathsf{F}, \forall_\mathsf{F} : \mathcal{P}(\mathsf{T}_1) \to \mathcal{P}(\mathsf{T}_2)$ given by left and right Kan extension.

As a small demonstration, suppose we have a program $\mathtt{f} : \mathtt{S} \to \mathtt{T}$, and we want to construct the predicate which checks whether a term of sort $\mathtt{T}$ has been processed by $\mathtt{f}$. The type is formed and terms are introduced as follows.

$$\frac{\mathsf{yT} \vdash \mathsf{y}\mathtt{f} : \mathsf{Type} \qquad \mathsf{yT}, \mathsf{y}\mathtt{f} \vdash \mathsf{y}\mathtt{S} : \mathsf{Type}}{\mathsf{yT} \vdash \langle \mathtt{f} \rangle := \Sigma \mathsf{g}{:}\mathsf{y}\mathtt{f}.\mathsf{y}\mathtt{S} : \mathsf{Type}} \qquad\qquad \frac{\mathsf{yT} \vdash \mathsf{g} : \mathsf{y}\mathtt{f} \qquad \mathsf{yT}, \mathsf{x}{:}\mathsf{y}\mathtt{f} \vdash \mathsf{u} : \mathsf{yS}[\mathsf{g}/\mathsf{x}]}{\mathsf{yT} \vdash \langle \mathsf{g}, \mathsf{u} \rangle : \langle \mathtt{f} \rangle.}$$

This is the principal sieve $\langle \mathtt{f} \rangle$ (ex. 8), which determines terms of the form $\mathtt{g} = \mathtt{f} \circ \mathtt{u}$ for some $\mathtt{u} : \mathtt{R} \to \mathtt{S}$. We can then write protocols based on this precondition in the native type system.

5 Applications

Native type systems are highly expressive and versatile. We demonstrate a few small examples. Notation is simplified by identifying sorts and constructors of T with their image in $\mathcal{P}(\mathsf{T})$.

5.1 Rewrite subsystems, modalities, and behavioral equivalence

In section §2 we motivated structured λ-theories by demonstrating that an internal category $\mathsf{Th.Cat} \to \mathsf{T}$ can be used to represent the operational semantics of T. We now apply this idea, with a subtle change: for morphisms to be *lists* of basic rewrites, we do not want composition. Instead we simply use a graph $\mathtt{G} := \langle s, t \rangle : \mathtt{E} \to \mathtt{V}, \mathtt{V}$ and implicitly consider the free category, *i.e.* we use pullbacks to construct lists of edges.

Let $\mathsf{Th.Gph} \to \mathsf{T}$ be a λ-theory with internal graph $\mathtt{G}$. Then $\mathsf{y}\mathtt{G} : \mathcal{P}(\mathsf{T})$ is the (dependent) type of rewrites over terms. The fiber over each pair is the set of rewrites between terms.

$$\mathtt{S}, a{:}\mathtt{V}, b{:}\mathtt{V} \vdash \mathtt{G}(a, b) : \mathsf{Type} \qquad\qquad \mathtt{G}(a, b) = \{\mathtt{S} \vdash e : a \rightsquigarrow b\}$$

This object is the space of all computations in language T. The native type system can be used to construct predicates which specify subgraphs of computations.

▶ **Example 23.** Let $\mathsf{Th.Gph} \to \mathsf{Th.}\rho\pi$ be the structured λ-theory of the $\rho\pi$-calculus (ex. 5), without composition of rewrites. In the presheaf topos $\mathcal{P}(\mathsf{Th.}\rho\pi)$, suppose we have a name predicate $\alpha : \mathtt{N} \to \mathsf{Prop}$, a process predicate $\varphi : \mathtt{P} \to \mathsf{Prop}$, and $F : [\mathtt{N} \to \mathsf{Prop}] \to [\mathtt{P} \to \mathsf{Prop}]$. Then $\mathtt{comm}(\alpha, \varphi, \chi.F) : [\mathtt{E}, \mathsf{Prop}]$ determines the communications

$$\mathtt{comm}(a, p, \lambda x.c) : \mathtt{out}(a, p) \mid \mathtt{in}(a, \lambda x.c) \rightsquigarrow c[@p/x]$$

on channels in namespace α, sending data in codespace φ, and continuing in contexts $\lambda x.c : [\mathtt{N}, \mathtt{P}]$ such that $\chi(@p) \Rightarrow F(\chi)(c[@p/x])$. Then $\Sigma e{:}\mathtt{G}.\mathtt{comm}(\alpha, \varphi, \chi.F)$ is the graph of these computations. This can be used to condition protocols or identify parts of a network.

We can express *temporal modalities* to reason about past and future behavior. Applying the "step" operators of ex. 12 to a predicate $\varphi : \mathtt{V} \to \mathsf{Prop}$ on terms, $\mathsf{B}_!(\varphi)$ are terms which *possibly* rewrite to φ, and $\mathsf{B}_*(\varphi)$ are terms which *necessarily* rewrite to φ. By iterating, we can form each kind of modality.

$$\begin{array}{llll} \mathsf{B}_!^\circ(\varphi) := \exists n{:}\mathbb{N}.\mathsf{B}_!^n(\varphi) & \text{can become } \varphi & \mathsf{B}_!^\bullet(\varphi) := \forall n{:}\mathbb{N}.\mathsf{B}_!^n(\varphi) & \text{always can become } \varphi \\ \mathsf{B}_*^\circ(\varphi) := \exists n{:}\mathbb{N}.\mathsf{B}_*^n(\varphi) & \text{will become } \varphi & \mathsf{B}_*^\bullet(\varphi) := \forall n{:}\mathbb{N}.\mathsf{B}_*^n(\varphi) & \text{always will become } \varphi \end{array}$$

Similarly for F, we can condition past behavior. Moreover, these operators and modalities can be restricted to subgraphs.

▶ **Example 24.** We can use the always modality to express constant system requirements, such as the capacity to receive and process input on certain channels, or the guarantee to only communicate on certain channels.

$$\mathsf{live}(\alpha) := \quad \mathsf{B}^{\bullet}_{*}(\mathtt{in}(\alpha, [\mathtt{N} \to \mathtt{P}]) \mid \mathtt{P})) \qquad\qquad \mathsf{safe}(\alpha) := \quad \mathsf{B}^{\bullet}_{*}(\neg[\mathtt{in}(\neg[\alpha], [\mathtt{N} \to \mathtt{P}]) \mid \mathtt{P}])$$

By checking that a program $\mathtt{in}(n, \lambda x.c) : \mathtt{in}(\mathtt{N}, \chi.\mathsf{safe})$, we ensure that the program will remain secure on the channel it receives.

Our rewrite graphs are deterministic, because each edge specifies all data in the term vertices. In operational semantics, rewrites are "silent reductions" which occur in a closed system, while more generally *transitions* allow for interaction with the environment. This can be expressed using substitution as pattern-matching, to construct a nondeterministic labelled transition system in which to derive behavioral equivalence.

▶ **Example 25.** Processes in the ρπ-calculus interact by parallel composition $-|-$. The basic actions are input and output. To construct the transition system of these observable behaviors, we first define the interaction contexts.

$$\mathsf{obs} := [\lambda x.x] \vee [\lambda x.(\mathtt{in}(\mathtt{N}, \mathtt{N} \to \mathtt{P}) \mid x)] \vee [\lambda x.(\mathtt{out}(\mathtt{N}, \mathtt{P}) \mid x)] : [\mathtt{P} \to \mathtt{P}] \to \mathsf{Prop}$$

We can then define the labelled transition system $\mathsf{act} : \mathtt{P}, [\mathtt{P} \to \mathtt{P}], \mathtt{P} \to \mathsf{Prop}$ as

$$p{:}\mathtt{P}, \lambda x.c{:}[\mathtt{P} \to \mathtt{P}], q{:}\mathtt{P} \vdash \mathsf{act}(p, \lambda x.c, q) := \mathtt{G}(ev[p, \mathsf{obs}(\lambda x.c)], q)$$

the predicate which is usually written as $p \xrightarrow{\lambda x.c} q$ we define to be $\exists e : \mathtt{G}.\ e : c[p/x] \rightsquigarrow q$. We can now construct new modalities relative to this observational graph, denoted with $(-)_{\mathsf{act}}$.

From this relation, many kinds of behavioral equivalence can be written explicitly as types. For example, bisimulation is the inductive type $\mathsf{Bisim} := \mu\varphi.\mathsf{S}(\varphi)$ for

$$\begin{aligned} \mathsf{S}(\varphi)(p, q) \quad := \quad & \forall y{:}\mathtt{P}.\ \forall \lambda x.c{:}[\mathtt{P}, \mathtt{P}].\ \mathsf{act}(p, \lambda x.c, y) \Rightarrow \exists z{:}\mathtt{P}.\ \mathsf{act}(q, \lambda x.c, z) \wedge \varphi(y, z) \wedge \\ & \forall z{:}\mathtt{P}.\ \forall \lambda x.c{:}[\mathtt{P}, \mathtt{P}].\ \mathsf{act}(q, \lambda x.c, z) \Rightarrow \exists y{:}\mathtt{P}.\ \mathsf{act}(p, \lambda x.c, y) \wedge \varphi(y, z) \end{aligned}$$

By constructing bisimilarity as a native type, we can reason up to behavioral equivalence.

5.2 Refined binding and reasoning about contexts

Hom types provide *refined binding*, i.e. using predicates to condition what can be substituted into a context. To formalize this idea, we need to restrict rewrite rules to require that a term satisfies the predicate which the context binds.

▶ **Example 26.** In the ρπ-calculus, an input process $\mathtt{in}(n, \lambda x.c)$ receives whatever is sent on the name n. We can refine input to receive only data which satisfies a predicate.

Consider the predicate theory (def. 14) of the ρπ-calculus. For each namespace α, define

$$\mathtt{comm}_{\alpha} : \mathtt{N}, \alpha[@], [\alpha \to \mathtt{P}] \to \mathtt{E} \qquad \mathtt{comm}_{\alpha}(n, p, \lambda x.c) : \mathtt{out}_{\alpha}(n, p) | \mathtt{in}_{\alpha}(n, \lambda x.c) \rightsquigarrow c[@p/x]$$

where $\alpha[@]$ is the preimage of α under $@ : \mathtt{P} \to \mathtt{N}$. This extends to polyadic communication.

The **refinement** of the ρπ-calculus is defined to be the subtheory $\rho\pi_{\omega} \subset \omega\mathsf{Th}.\rho\pi$ in which the only rewrite constructors are $\mathtt{comm}_{\alpha}$ for each namespace. In this theory, $\mathtt{in}_{\alpha} : \mathtt{N}, [\alpha \to \mathtt{P}] \to \mathtt{P}$ constructs processes which only receive data on α.

The namespace $\alpha : \mathtt{N} \to \mathsf{Prop}$ could be a predicate on structured data, a set of trusted addresses, or the implementations of an algorithm. Then $\mathtt{in}(n, \lambda x{:}\alpha.p)$ can be understood as a *query* for α. In the refined language, we can search by both structure and behavior.

The notion of refinement generalizes to any λ-theory with rewrites which have distinguished "substitution sort pairs", like those for the output process and the input bound name.

A common question in software is "what contexts ensure this implication?" For example, "where can this protocol be executed without security leaks?" Hom types provide this expressive power for reasoning contextually in codebases.

By composing the hom type with modalities, we can extend contextual reasoning over term behavior. In particular, $\varphi \rhd \psi := [\varphi, \mathsf{B}_*^\circ(\psi)]$ are contexts for which substituting φ can *eventually* lead to some condition, desired or otherwise.

▶ **Example 27.** An arrow can be used to detect security leaks: given a trusted channel $a : \mathtt{N}$ and an untrusted $n : \mathtt{N}$, then the following program will not preserve safety on a.

$$\lambda p.(p \mid \mathtt{out}(a, \mathtt{in}(n, \lambda x.c))) : \mathsf{safe}(a) \rhd \neg[\mathsf{safe}](a)$$

We can also detect if a program may not remain single-threaded: if $\mathsf{s.thr} := \neg[0] \wedge \neg[\neg[0] \mid \neg[0]]$, then $\lambda p.\mathtt{out}(a, (p \mid q)) : \mathsf{s.thr} \rhd_{\mathsf{act}} \neg[\mathsf{s.thr}]$, where $\rhd_{\mathsf{act}}$ is the arrow relative to the observational transition system (ex. 25).

In this way, the process of finding bugs can be automated as a form of type-checking. The query time depends only on the system complexity and the efficiency of the type checker. Moreover, with subtyping this reasoning expands to collections of programs.

5.3 Translating across language paradigms

The native types construction is functorial, allowing us to reason across translations. We sketch a simple example of the benefits of relating across programming paradigms.

▶ **Example 28** (Translations)**.** In the appendix §A, we give a translation $\tau : \mathsf{Th.N}\lambda \to \mathsf{Th.}\pi$ from the name-passing λ-calculus into the π-calculus. This induces a functor $\mathcal{P}(\tau) : \mathcal{P}(\mathsf{Th.}\pi) \to \mathcal{P}(\mathsf{Th.N}\lambda)$, which in turn induces a translation of the native type systems.

A π-calculus predicate $\varphi : \mathtt{P} \to \mathsf{Prop}$ contains processes which may involve highly nondeterministic interaction between agents in a network. In the translation, it is mapped to a λ-calculus predicate $\mathcal{P}(\tau)(\varphi) : \mathtt{T} \to \mathsf{Prop}$ by preimage; this has the effect of restricting φ to its "functional" processes.

Because λ-terms have no side-effects and execute deterministically, restricting to functional terms allows significant optimization in network computing; e.g. agents trying to reach consensus about side effects. Similar to how a compiler can optimize a tail call in a functional language, a compiler could recognize that a π-term can be implemented functionally and run the consensus protocol on not the details of the execution but only the result.

These are a few small examples, which hardly scratch the surface of native type theory. Native types are practical because they are basic: they are made by logic from the languages we already use. We encourage the reader to explore what native types can do for you.

6 Conclusion

Native type theory is a functorial way to generate expressive type systems for a broad class of languages. We have presented the construction and given directions for application.

The authors believe that integrating native type systems in software is a viable way to provide a shared framework of higher-order reasoning in everyday computing. Most of the tools necessary for implementation already exist.

References

1 Flow: A static type checker for javascript. URL: https://flow.org/.

2 Google closure compiler. URL: https://developers.google.com/closure/compiler.

3 Hoogle. URL: https://hoogle.haskell.org/.

4 K framework. URL: http://www.kframework.org/.

5 Kjs: A complete formal semantics of javascript. URL: https://github.com/kframework/javascript-semantics.

6 Microsoft typescript. URL: https://www.typescriptlang.org/.

7 Rchain. URL: https://www.rchain.coop/.

8 A spatial logic model checker. URL: http://ctp.di.fct.unl.pt/SLMC/.

9 Samson Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51(1-2):1–77, mar 1991. URL: https://doi.org/10.1016%2F0168-0072%2891%2990065-t, doi:10.1016/0168-0072(91)90065-t.

10 Steve Awodey. *Category Theory*. Oxford University Press, Inc., USA, 2nd edition, 2010.

11 Steve Awodey. Natural models of homotopy type theory. *Mathematical Structures in Computer Science*, 28(2):241–286, nov 2016. doi:10.1017/s0960129516000268.

12 H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. Elsevier, 1984.

13 Gérard Boudol. The π-calculus in direct style. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '97*. ACM Press, 1997. URL: https://doi.org/10.1145%2F263699.263726, doi:10.1145/263699.263726.

14 Mitchell Buckley. Fibred 2-categories and bicategories. *Journal of Pure and Applied Algebra*, 218(6):1034–1074, jun 2014. URL: https://doi.org/10.1016%2Fj.jpaa.2013.11.002, doi:10.1016/j.jpaa.2013.11.002.

15 John Cartmell. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986. URL: https://www.sciencedirect.com/science/article/pii/0168007286900539, doi:https://doi.org/10.1016/0168-0072(86)90053-9.

16 Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, feb 1988. URL: https://doi.org/10.1016%2F0890-5401%2888%2990005-3, doi:10.1016/0890-5401(88)90005-3.

17 Roy L. Crole. *Categories for Types*. Cambridge University Press, 1994. doi:10.1017/CBO9781139172707.

18 Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2 edition, 2016. doi:10.1017/CBO9781316576892.

19 André Hirschowitz, Tom Hirschowitz, and Ambroise Lafont. Modules over monads and operational semantics, 2020. arXiv:2012.06530.

20 B. Jacobs. *Categorical Logic and Type Theory*. Elsevier, Amsterdam, 1998. doi:10.1016/s0049-237x(98)x8028-6.

21 Peter T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium: 2 Volume Set*. Oxford University Press UK, 2002.

22 J. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, USA, 1986.

23 Saunders Mac Lane and Ieke Moerdijk. *Sheaves in Geometry and Logic*. Springer New York, 1994. URL: https://doi.org/10.1007%2F978-1-4612-0927-0, doi:10.1007/978-1-4612-0927-0.

24 F. William Lawvere. Adjointness in foundations. *dialectica*, 23(3-4):281–296, dec 1969. URL: https://doi.org/10.1111%2Fj.1746-8361.1969.tb01194.x, doi:10.1111/j.1746-8361.1969.tb01194.x.

25 Per Martin-Löf. An intuitionistic theory of types. In *Twenty Five Years of Constructive Type Theory*. Oxford University Press, oct 1998. URL: https://doi.org/10.1093%2Foso%2F9780198501275.003.0010, doi:10.1093/oso/9780198501275.003.0010.

26 Paul-André Melliès and Noam Zeilberger. Functors are type refinement systems. *ACM SIGPLAN Notices*, 50(1):3–16, may 2015. URL: https://doi.org/10.1145%2F2775051.2676970, doi:10.1145/2775051.2676970.

27 L. G. Meredith and Matthias Radestock. Namespace logic: A logic for a reflective higher-order calculus. In *Trustworthy Global Computing*, pages 353–369. Springer Berlin Heidelberg, 2005. URL: https://doi.org/10.1007%2F11580850_19, doi:10.1007/11580850_19.

28 L.G. Meredith and Matthias Radestock. A reflective higher-order calculus. *Electronic Notes in Theoretical Computer Science*, 141(5):49–67, dec 2005. URL: https://doi.org/10.1016%2Fj.entcs.2005.05.016, doi:10.1016/j.entcs.2005.05.016.

29 Robin Milner. The polyadic π-calculus: a tutorial. In *Logic and Algebra of Specification*, pages 203–246. Springer Berlin Heidelberg, 1993. URL: https://doi.org/10.1007%2F978-3-642-58041-3_6, doi:10.1007/978-3-642-58041-3_6.

30 Ieke Moerdijk and Erik Palmgren. Wellfounded trees in categories. *Annals of Pure and Applied Logic*, 104(1-3):189–218, jul 2000. URL: https://doi.org/10.1016%2Fs0168-0072%2800%2900012-9, doi:10.1016/s0168-0072(00)00012-9.

31 Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, jul 1991. URL: https://doi.org/10.1016%2F0890-5401%2891%2990052-4, doi:10.1016/0890-5401(91)90052-4.

32 Davide Sangiorgi. Communicating and mobile systems: the π-calculus,. *Science of Computer Programming*, 38(1-3):151–153, aug 2000. URL: https://doi.org/10.1016%2Fs0167-6423%2800%2900008-3, doi:10.1016/s0167-6423(00)00008-3.

33 Mike Shulman. exponentials of cartesian closed categories. http://ncatlab.org/nlab/show/cartesian%20closed%20category#exponentials_of_cartesian_closed_categories.

34 Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq coq correct! verification of type checking and erasure for coq, in coq. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371076.

35 Ross Street. Elementary cosmoi i. In Gregory M. Kelly, editor, *Category Seminar*, pages 134–180, Berlin, Heidelberg, 1974. Springer Berlin Heidelberg.

36 Andrzej Tarlecki, Rod M. Burstall, and Joseph A. Goguen. Some fundamental algebraic tools for the semantics of computation: Part 3. indexed categories. *Theoretical Computer Science*, 91(2):239 – 264, 1991. URL: http://www.sciencedirect.com/science/article/pii/030439759190085G, doi:https://doi.org/10.1016/0304-3975(91)90085-G.

37 D. Turi and G. Plotkin. Towards a mathematical operational semantics. In *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science*. IEEE Comput. Soc. URL: https://doi.org/10.1109%2Flics.1997.614955, doi:10.1109/lics.1997.614955.

A Appendix

Origin and related work

The present work began with Greg Meredith seeking a method to generate logics for concurrent languages, motivated by Abramsky's Domain Theory in Logical Form [9]. In 2005 Meredith developed Namespace Logic [27], an expressive logic for data and code in the ρ-calculus, just as Cardelli was developing Spatial-Behavioral logic [8] for the π-calculus.

Intuiting a general method, Meredith later began collaboration with Stay, who explored approaches in category theory. In 2018 they brought in Williams and after extensive discussion of the vision, it became clear that categorical logic offered a powerful method of generating type systems for λ-theories, including most concurrent languages.

Native type theory is an entire world to explore, both in theory and practice. Yet there are desiderata for a comprehensive logic for concurrency which may not be addressed by the language of toposes, and the project continues to expand.

Translation of structured λ-theories

The translation of the name-passing λ-calculus into the π-calculus.

▶ **Example 29.** Name-passing λ-calculus [13]

V	variables	T	terms	E	rewrites of terms (+Th.Cat)

$$\begin{array}{llllll} \mathtt{lam}: & [\mathtt{V}\to\mathtt{T}]\to\mathtt{T} & \mathtt{var}: & \mathtt{V}\to\mathtt{T} & \mathtt{C}: & \mathtt{V},\mathtt{T},\mathtt{T}\to\mathtt{T} \\ \mathtt{app}: & \mathtt{T},\mathtt{V}\to\mathtt{T} & \mathtt{def}: & \mathtt{T},[\mathtt{V}\to\mathtt{T}]\to\mathtt{T} & & \end{array}$$

$$\begin{array}{rlrl} \beta: & [\mathtt{V}\to\mathtt{T}],\mathtt{V}\to\mathtt{E} & \beta(Q,y): & \mathtt{app}(\mathtt{lam}(Q),y)\rightsquigarrow Q(y) \\ \phi: & \mathtt{V},\mathtt{T},\mathtt{T}\to\mathtt{E} & \phi(x,Q): & \mathtt{C}(x,Q,\mathtt{var}(x))\rightsquigarrow Q \\ \mathtt{app_e}: & \mathtt{E},\mathtt{V}\to\mathtt{E} & \mathtt{app_e}(\rho,N): & \mathtt{app}(s(\rho),N)\rightsquigarrow\mathtt{app}(t(\rho),N) \\ \mathtt{def_e}: & \mathtt{T},[\mathtt{V}\to\mathtt{E}]\to\mathtt{E} & \mathtt{def_e}(M,\lambda x.\rho): & \mathtt{def}(M,\lambda x.s(\rho))\rightsquigarrow\mathtt{def}(M,\lambda x.t(\rho)) \end{array}$$

$$\begin{array}{lcl} \mathtt{def}(Q,\lambda x.R) & = & \mathtt{def}(Q,\lambda x.\mathtt{C}(x,Q,R)) \\ \mathtt{C}(x,Q,\mathtt{def}(R,\lambda y.S)) & = & \mathtt{def}(R,\lambda y.\mathtt{C}(x,Q,S)) \\ \mathtt{C}(x,Q,\mathtt{app}(R,y)) & = & \mathtt{app}(\mathtt{C}(x,Q,R),y) \end{array}$$

The name-passing λ-calculus uses references to avoid copying large data structures. It is a restriction of the λ-calculus in that terms may only be applied to variables; while it is an enrichment in that it introduces an environment `def` that records binding. There is also a carrier `C`, which serves to transport the recorded binding from its declaration to its use.

The usual β reduction splits into two reductions. The first, denoted β, replaces variables in a term with other variables. The second, denoted ϕ (for "fetch"), replaces a variable in head position with the term to which it is bound in the environment.

The edge constructors $\mathtt{app_e}$ and $\mathtt{def_e}$ describe the propagation of reduction contexts into the term: reductions may only occur in the head position of an application or under a `def`.

▶ **Example 30.** Polyadic asynchronous π-calculus [32]

N	names	P	processes	E	rewrites between processes (+Th.Cat)

$$\begin{array}{rlrll} \mathtt{0}: & 1\to\mathtt{P} & \mathtt{in}_k: & \mathtt{N},[\mathtt{N}^k\to\mathtt{P}]\to\mathtt{P} & \\ -|-: & \mathtt{P},\mathtt{P}\to\mathtt{P} & \mathtt{out}_k: & \mathtt{N},\mathtt{N}^k\to\mathtt{P} & \\ !: & \mathtt{P}\to\mathtt{P} & \nu: & [\mathtt{N}\to\mathtt{P}]\to\mathtt{P} & \text{syntactic sugar: } \nu x.p \text{ means } \nu(\lambda x.p) \end{array}$$

$$\begin{array}{rlrl} \mathtt{comm}_k: & \mathtt{N},\mathtt{N}^k,[\mathtt{N}^k\to\mathtt{P}]\to\mathtt{E} & \mathtt{comm}_k(n,\vec{a_i},\lambda\vec{y_i}.Q): & \mathtt{out}_k(n;\vec{a_i})|\mathtt{in}_k(n,\lambda\vec{y_i}.Q)\rightsquigarrow Q[a_i/y_i] \\ \mathtt{par}_l: & \mathtt{E},\mathtt{P}\to\mathtt{E} & \mathtt{par}_l(\langle p,e\rangle,q): & p|q\rightsquigarrow t(e)|q \\ \nu_\mathtt{e}: & [\mathtt{N}\to\mathtt{E}]\to\mathtt{E} & \nu_\mathtt{e}x.\rho: & \nu x.s(\rho)\rightsquigarrow\nu x.t(\rho) \end{array}$$

$$\begin{array}{lcll} !Q & = & !Q|Q & \\ (\mathtt{P},|,0) & & \text{commutative monoid} & \\ \nu x.\nu y.Q & = & \nu y.\nu x.Q & \\ \nu x.0 & = & 0 & \\ Q|\nu x.R & = & \nu x.(Q|R) & \text{“scope extrusion”} \end{array}$$

The π-calculus [29] models concurrent processes which compute via *communication*, or the exchange of "names". It is like the ρ-calculus of this paper, without reflection and with two added constructors. The replication operator ! makes infinitely many copies of a process. The ν operator introduces a new scope in which a fresh name has been made available to the contained process. Scopes can expand via scope extrusion to absorb other processes running in parallel with the scope.

▶ **Proposition 31.** *There is a translation* $[\![-]\!] : \mathsf{Th.N\lambda} \to \mathsf{Th.}\pi$, *given below.*

sorts
$[\![\mathtt{V}]\!] = \mathtt{N}$
$[\![\mathtt{T}]\!] = [\mathtt{N} \to \mathtt{P}]$
$[\![\mathtt{Hom_V}]\!] = \mathtt{Hom_P}$

constructors
$[\![\mathtt{var}]\!] : \mathtt{N} \to [\mathtt{N} \to \mathtt{P}]$
$[\![\mathtt{var}(x)]\!] = \lambda u.\mathtt{out}_1(x, u)$

$[\![\mathtt{lam}]\!] : [\mathtt{N} \to [\mathtt{N} \to \mathtt{P}]] \to [\mathtt{N} \to \mathtt{P}]$
$[\![\mathtt{lam}(\lambda x.Q)]\!] = \lambda u.\mathtt{in}_2(u, \lambda x.[\![Q]\!])$

$[\![\mathtt{app}]\!] : [\mathtt{N} \to \mathtt{P}], \mathtt{N} \to [\mathtt{N} \to \mathtt{P}]$
$[\![\mathtt{app}(Q, x)]\!] = \lambda u.\nu v.([\![Q]\!](v)|\mathtt{out}_2(v; x, u))$

$[\![\mathtt{def}]\!] : [\mathtt{N} \to \mathtt{P}], [\mathtt{N} \to [\mathtt{N} \to \mathtt{P}]] \to [\mathtt{N} \to \mathtt{P}]$
$[\![\mathtt{def}(Q, \lambda x.R)]\!] = \lambda u.\nu x.([\![R]\!](u)|!\mathtt{in}_1(x, [\![Q]\!])))$

$[\![\mathtt{C}]\!] : \mathtt{N}, [\mathtt{N} \to \mathtt{P}], [\mathtt{N} \to \mathtt{P}] \to [\mathtt{N} \to \mathtt{P}]$
$[\![\mathtt{C}(x, Q, R)]\!] = \lambda u.([\![R]\!](u)|\mathtt{in}_1(x, [\![Q]\!]))$

The translation preserves equations and rewrites; we give the computation for β-reduction.

rewrites

$$\begin{array}{rll}
[\![\beta]\!] : & [\mathtt{N} \to [\mathtt{N} \to \mathtt{P}]], \mathtt{N} \to \mathtt{E} & \\
[\![\beta(Q, x)]\!] : & [\![\mathtt{app}(\mathtt{lam}(Q), x)]\!] & \\
& = \lambda u.\nu v.([\![\mathtt{lam}(Q)]\!](v)|\mathtt{out}_2(v; x, u)) & [\![\mathtt{app}]\!] \\
& = \lambda u.\nu v.([\![\mathtt{lam}(\lambda y.Q(y))]\!](v)|\mathtt{out}_2(v; x, u)) & \text{extensionality} \\
& = \lambda u.\nu v.(\mathtt{in}_2(v, \lambda y.[\![Q(y)]\!]|\mathtt{out}_2(v; x, u)) & [\![\mathtt{lam}]\!] \\
& \rightsquigarrow \lambda u.\nu v.[\![Q(x)]\!](u) & \mathtt{comm}_2 \\
& = \lambda u.([\![Q(x)]\!](u)|\nu v.0) & \text{scope extrusion} \\
& = \lambda u.([\![Q(x)]\!](u)|0) & \\
& = \lambda u.[\![Q(x)]\!](u) & \\
& = [\![Q(x)]\!] & \text{extensionality}
\end{array}$$