Native Type Theory

Christian Williams
University of California, Riverside, US
cbwill.math@gmail.com

Michael Stay
Pyrofex Corporation, Utah, US
stay@pyrofex.net

Native type systems are those in which type constructors are derived from term constructors, as well as the constructors of predicate logic and intuitionistic type theory. We present a method to construct native type systems for a broad class of languages, λ-theories with equality, by embedding such a theory into the internal language of its topos of presheaves. Native types provide total specification of the structure of terms; and by internalizing transition systems, native type systems serve to reason about structure and behavior simultaneously. The construction is functorial, thereby providing a shared framework of higher-order reasoning for many languages, including programming languages.

1 Introduction

Type theory is growing as a guiding philosophy in the design of programming languages. However in practice, type systems are heterogeneous, and there are no standard ways to reason across languages. We present a functorial method to enhance a language with "its own internal logic", using tools and ideas of category theory.

Categorical logic unifies languages: virtually any formalism, from monoids to dependent type theory, can be modelled by structured categories [17]. 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.

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

The first is the *presheaf construction* $\mathcal{P}$ [8, Ch. 8]; it preserves product, equality, and function types. The second is the *language of a topos* $\mathcal{L}$ [17, Ch. 11]. The composite is 2-functorial, so that translations between languages induce translations between type systems.

Note — This idea is quite simple; in fact it was considered more than fifty years ago by Scott [31]. Native type theory simply gives a name to the language of presheaves on theories; we aim to demonstrate its utility, and advocate for real-world application of categorical logic.

The type system is *native* in the sense that type constructors are derived from term constructors, plus those of predicate logic and (co/inductive) intuitionistic 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 encourage community development.

K. Kishida (Ed.): Fourth International Conference
on Applied Category Theory (ACT 2021).
EPTCS 372, 2022, pp. 116–132, doi:10.4204/EPTCS.372.9

1.1 Motivation and implementation

As software systems become increasingly complex, it is critical to develop adequate frameworks for reasoning about code systematically across languages. By generating 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.

Native type theory is intended to be implemented as a development environment, based on a library of formal semantics and translations, in which one can program in languages enhanced by their native type systems. Code can be written in the same way, but enriched with predicates and dependent types, both (1) to condition existing codebases and (2) 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 λ*-theories with equality* (§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. 14).

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

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 [32]. For usability, there will need to be 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. The main original contribution is that by internalizing transition systems in λ-theories, native type systems can reason about both the structure and behavior of terms simultaneously.

§2 **Structured λ-theories**. We define λ*-theories with equality* as cartesian closed categories with pullbacks, and we interpret the internal language as simply-typed λ-calculus combined with the syntax of generalized algebraic theories [12].

Theories ordinarily model the structure of programs, while behavior is modelled separately [35]; but in fact transition systems can be modelled as internal categories. The concept of "language equipped with a notion of behavior" motivates the 2-category of *structured λ-theories*. We define the $\rho\pi$-calculus [26], 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).

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.

§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* [17], as in the Calculus of Constructions [13]. 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.

2 Structured lambda-theories

Simply-typed λ-calculus is the language of products and functions. It is regarded as the foundation of computer science [10] and much of modern programming [15].

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. The λ-calculus is the syntax of *cartesian closed categories* [19].

A particular λ-calculus or λ-*theory* is presented by sorts, constructors, and equations. This is just like presenting an algebraic structure such as groups, but with higher-order constructors (ex. 5, `input`). References for the syntax and semantics of simply-typed λ-calculus are [14, Ch. 4] and [17, Ch. 2]. To save space in presentations, we denote products by $\mathtt{S},\mathtt{T}$ and functions by $[\mathtt{S}\to\mathtt{T}]$.

The main rules of a λ-theory define how to construct and use functions.

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

Definition 1. A **λ-theory with equality** is a cartesian closed category with pullbacks, also known as a "properly cartesian closed category" [18]. The 2-category of λ-theories with equality, finitely continuous closed functors, and natural transformations is $\lambda\mathrm{Thy}_{=}$.

The syntax of a λ-theory with equality can be derived from its subobject fibration having fibered equality [17, Ch. 3]. We interpret the language as simply-typed λ-calculus combined with the syntax of *generalized algebraic theories* [12], which provide *indexed sorts*.

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

Indexed sorts are highly expressive. If we take the source map of a graph $s : \mathtt{E}\to\mathtt{V}$ as an indexed sort, then $s(v)$ is the sort of edges out of a vertex — i.e., the behavior of a term.

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

λ-theories with behavior

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 [35], but the operational semantics of higher-order languages is still in development [16]. We introduce a method of representing behavior internally.

A language with a rewrite system can be modelled by a λ-theory T with an internal category, which includes constructors and equations to specify the interaction between rewrites and constructors, a.k.a. the *operational semantics*. First, here is the theory of categories.

Definition 2. Th.Cat

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

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

Note — 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.

Operational semantics describes how term constructors interact with the transition system [35]: given a constructor $\mathtt{f} : \prod \mathtt{S}_i \to \mathtt{T}$ and terms $v_i : \mathtt{S}_i$ with edges $e_{ij} : v_i \rightsquigarrow w_{ij}$, what is the behavior of $\mathtt{f}(v_1,\ldots,v_n)$?

$$\frac{\{e_{1j} : v_1 \rightsquigarrow w_{1j}\} \quad \ldots \quad \{e_{nj} : v_n \rightsquigarrow w_{nj}\}}{\{e_k : \mathtt{f}(v_1,\ldots,v_n) \rightsquigarrow \mathtt{g}_k\}}\ \mathsf{R}(\mathtt{f})$$

To specify this interaction, we take the source map $s : \mathtt{E} \to \mathtt{S}$ as an indexed sort $\mathtt{S}^*(x)$; then $\mathtt{S}^*(v)$ are the edges with source v. This allows us to define operational semantics in a λ-theory.

Definition 3. Let T be a λ-theory with sorts $\mathtt{S}_i$, constructors $\mathtt{f}_{ij} : \prod_j \mathtt{S}_{ij} \to \mathtt{S}_i$, and graphs for each sort. A **behavior rule** for a term constructor $\mathtt{f} : \prod \mathtt{S}_i \to \mathtt{S}$ is a constructor

$$\mathsf{R}(\mathtt{f})_{\vec{v}} : \prod \mathtt{S}_i^*(v_i) \to \mathtt{S}^*(\mathtt{f}(\vec{v}))$$

such that $\mathsf{R}(\mathtt{f})_{\vec{v}}(e_1,\ldots,e_n) : \mathtt{f}(\vec{v}) \rightsquigarrow \mathtt{g}(\vec{e})$, where $\mathtt{g} : \prod \mathtt{S}_i^*(v_i) \to \mathtt{S}$.

An **operational semantics** for T is for each sort a family of edges $\{\mathtt{r}_{ij}(\vec{x}) : \mathtt{a}_{ij}(\vec{x}) \rightsquigarrow \mathtt{b}_{ij}(\vec{x}) : \mathtt{E}_{\mathtt{S}_i}\}$ and for some term constructors a behavior rule $\{\langle \mathtt{f}_{ij}, \mathsf{R}(\mathtt{f}_{ij})\rangle\}$. This defines a subtheory $\mathsf{O}(\mathsf{T}) \to \mathsf{T}$.

Theorem 4. Behavior rules correspond to GSOS rules [35] for deterministic labelled transition systems. The general case can be derived using an internal relation on $\mathtt{V},\mathtt{A},\mathtt{V}$, where $\mathtt{A}$ is a sort of actions.

Hence, internal operational semantics are equivalent to GSOS distributive laws. Providing a novel perspective to a basic topic in computer science, this connection warrants exploration in future work.

By representing behavior internally, we will see that

native type systems reason about both the structure and behavior of terms.

For example, there can be a predicate for "contexts $\lambda x.c : \mathtt{S} \to \mathtt{T}$ such that if $a : \mathtt{S}$ satisfies φ then for all $e_i : c[a/x] \rightsquigarrow b$ if $\psi_i(b)$ then no step of e satisfies ε". Combining both kinds of reasoning is extremely expressive, and the applications in §5 provide only a modest glimpse.

Example 5. $\rho\pi$-calculus $\mathsf{Th}.\rho\pi$ (polyadic)

The $\rho\pi$-calculus or **r**eflective **h**igher-**o**rder π-calculus [26] is a concurrent language succeeding the π-calculus [28]. It is the language of the blockchain platform RChain [7].

The $\rho\pi$-calculus has sorts $\mathtt{P}$ and $\mathtt{N}$ for processes and names, 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 $\mathtt{comm}$: an output and input process in parallel on the same name can *communicate*, transfering a list of processes as data.

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

$$\begin{array}{rcl} \mathtt{comm}_k(n,\vec{q}_i,\lambda\vec{x}_i.p) & : & \mathtt{out}(n,\vec{q}_i) \mid \mathtt{in}(n,\lambda\vec{x}_i.p) \rightsquigarrow p[@q_i/x_i] \\ \mathtt{run}(p) & : & *(@p) \rightsquigarrow p \end{array}$$

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

The $\rho\pi$-calculus is our running example of a λ-theory. In the native type system $\mathcal{LP}(\mathsf{T})$ (§4) of $\mathsf{T} = \mathsf{Th}.\rho\pi$, a predicate on names $\alpha : y\mathtt{N} \to \mathsf{Prop}$ is called a *namespace* [25], and a predicate on processes $\varphi : y\mathtt{P} \to \mathsf{Prop}$ is called a *codespace*.

In distinguishing internal graphs and edge constructors for the behavior of terms, we should also require that morphisms of "theories with behavior" should *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 $\tau : \mathsf{S} \to \mathsf{T}$. The 2-category of S-structured λ-theories is the strict coslice 2-category $\mathsf{S}/\lambda\mathrm{Thy}_=$. The 2-category of all structured λ-theories is the strict arrow 2-category $[\mathrm{I}, \lambda\mathrm{Thy}_=]$.

Because native type theory is functorial, a 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. As the concept is very general, we give a few more examples.

Example 7. Sorting A polyadic language such as the $\rho\pi$-calculus can be refined with many sorts of name, and name sorts given sorted arities for input and output. This is used by Milner [28] to designate channels to send and receive certain kinds of data.

Sorting is a structure $\sigma : S(\mathsf{Th}.\rho\pi) \to \mathsf{Th}.\rho\pi$, where the fiber over $\mathtt{N}$ is the set of name sorts, and the fibers over $\mathtt{in}_k$ and $\mathtt{out}_k$ are the sorted inputs and outputs with total arity k.

The native type system of the structured λ-theory contains and converts between the sorted and unsorted language. By functoriality, the type system expands to the 2-category of all sorted $\rho\pi$-calculi.

Example 8. Encoding Because a structure is simply a morphism of λ-theories, we can consider any translation $\tau : \mathsf{S} \to \mathsf{T}$ as a structured λ-theory. This can be understood as compiling or encoding the programs of S into those of T.

For example if we encode a complex language like C++ into a simpler language like name-passing λ-calculus, then we can identify a type of λ-term with good properties and take its preimage along τ to form a type of well-behaved C++ programs.

Another example is mapping a theory into a computing environment, such as a virtual machine. This allows for reasoning about both languages and their implementation.

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 [22] expands the domain of predicate logic and intuitionistic type theory [23] 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 $[\mathsf{T}^{\mathsf{op}}, \mathrm{Set}]$, denoted $\mathcal{P}(\mathsf{T})$. This defines a 2-functor to elementary toposes and geometric morphisms

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

where $\mathrm{Lan}_{\mathsf{F}}$ is left Kan extension [21, Ch. 10] 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 $\mathrm{c} : \mathsf{T}(-, \Omega) \simeq \mathrm{Sub}(-) : \mathsf{T}^{\mathsf{op}} \to \mathrm{Pos}$. 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 $\Omega(\mathtt{S}) \simeq \{\text{sieves on } \mathtt{S}\}$. A **sieve** on $\mathtt{S}$ is a set of morphisms into $\mathtt{S}$ closed under precomposition. A simple example is a *principal sieve* $\langle \mathtt{f} \rangle : \Omega(\mathtt{T})$ generated by $\mathtt{f} : \mathtt{S} \to \mathtt{T}$.

$$\langle \mathtt{f} \rangle(\mathtt{R}) = \{t : \mathtt{R} \to \mathtt{T} \mid \exists u{:}\mathtt{R} \to \mathtt{S}.\ \mathtt{f}(u) = t\}$$

A sieve can be understood as a set of shapes of abstract syntax tree with (sorted) holes for leaves, closed under substitution. These are the basic objects of reasoning in native type theory, as they are predicates on representable presheaves.

Example 9. In the $\rho\pi$-calculus (ex. 5) we can define a context $c(n) : \mathtt{P} \to \mathtt{P}$ which replicates processes on a name $n : 1 \to \mathtt{N}$.

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

One can check that $!(p)(n) \rightsquigarrow !(p)(n) \mid p$ 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). Finally, sorts and term constructors are identified with their images under y, so $\mathtt{S}$ will mean $y\mathtt{S}$ when applicable.

3.1 The predicate fibration

For any λ-theory T, there is a "category of predicates" $\Omega\mathcal{P}(\mathsf{T})$ over $\mathcal{P}(\mathsf{T})$ where the fiber over each presheaf is its poset of predicates. Quantification gives adjoints to change-of-base between fibers; we show that moreover the domain is cartesian closed, complete and cocomplete. The structure of this fibration provides higher-order predicate logic of the presheaves on T.

We use Ω^A to denote the poset of predicates $\varphi : A \to \Omega$, ordered by entailment. The *predicate functor* of $\mathcal{P}(\mathsf{T})$ is $\Omega^{(-)} : \mathcal{P}(\mathsf{T})^{\mathsf{op}} \to \mathrm{Pos}$. For $f : A \to B$, precomposition of predicates corresponds to preimage of subobjects. This is written as substitution $\varphi[f] := \Omega^f(\varphi)$.

Substitution can be understood as *pattern-matching*.

Example 10. For a $\rho\pi$-calculus predicate $\varphi : y(\mathtt{P}) \to \mathsf{Prop}$, substitution by $y(\mathtt{in}) : y(\mathtt{N}) \times y([\mathtt{N},\mathtt{P}]) \to y(\mathtt{P})$ is the query "inputting on what name-context pairs yield property φ?"

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

Each poset Ω^A is in fact a complete Heyting algebra: meet and join are intersection and union, $\top = A$ and $\bot = (\mathtt{S} \mapsto \emptyset)$, implication is

$$(\varphi \Rightarrow \psi)_{\mathtt{S}}(a) := \ \Pi u{:}\mathtt{R} \to \mathtt{S}.\ \varphi_{\mathtt{R}}(a \cdot u) \Rightarrow \psi_{\mathtt{R}}(a \cdot u)$$

and negation is $\neg[\varphi] := (\varphi \Rightarrow \bot)$.

We can assemble the image of $\Omega^{(-)}$ into one category, with the Grothendieck construction [17, 1.10].

Definition 11. The *category of predicates* of $\mathcal{P}(\mathsf{T})$, denoted $\Omega\mathcal{P}(\mathsf{T})$, is defined as follows.

Object	a pair	$\langle A\ ,\ \varphi : \Omega^A \rangle$
Morphism	a pair	$f : \langle A, \varphi \rangle \to \langle B, \psi \rangle$
	$=$	$\langle f : A \to B\ ,\ \varphi \Rightarrow \psi[f] \rangle$
Composition		$f;g : \langle A, \varphi \rangle \to \langle B, \psi \rangle \to \langle C, \chi \rangle$
	$=$	$\langle f;g : A \to C, \varphi \Rightarrow \psi[f] \Rightarrow \chi[g][f] \rangle$

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 [17, Ch. 1]. The predicate fibration is highly structured: each change-of-base functor has adjoints which are *dependent sum* and *dependent product*.

Proposition 12. The projection $\pi_\Omega : \Omega\mathcal{P}(\mathsf{T}) \to \mathcal{P}(\mathsf{T})$ has **indexed sums and products** [17]: for each $f : A \to B$, the functor $\Omega^f : \Omega^B \to \Omega^A$ has left and right adjoints $\exists_f \dashv \Omega^f \dashv \forall_f$.

$$\exists_f(\varphi)^b_{\mathtt{S}} := \ \exists a{:}A_{\mathtt{S}}.\ (f_{\mathtt{S}}(a) = b) \wedge \varphi(a) \qquad \forall_f(\varphi)^b_{\mathtt{S}} := \ \forall(u{:}\mathtt{R} \to \mathtt{S}).\forall a{:}A.\ (f_{\mathtt{R}}(a) = b) \Rightarrow \varphi(a)$$

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 13. Let $\mathsf{Th.Gph} \to \mathsf{T}$ be a λ-theory with a graph, and $\varphi : \mathtt{V} \to \mathsf{Prop}$ be a predicate on terms. Then $\varphi[s] : \mathtt{E} \to \mathsf{Prop}$ are rewrites with φ(source), and $\exists_t(\varphi[s])$ are the targets of these rewrites. Hence there is a *step-forward* operation $\mathsf{F}_! := [s];\exists_t : [\mathtt{V}, \mathsf{Prop}] \to [\mathtt{V}, \mathsf{Prop}]$.

The *secure step-forward* is a more refined operation: $\mathsf{F}_*(\varphi) := \forall_t(\varphi[s])$ determines the 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* [20] and a **higher-order fibration** [17, section 5.3].

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

Proposition 14. $\Omega\mathcal{P}(\mathsf{T})$ 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 $[\varphi, \psi] := \forall_{\pi_2}(\varphi[\pi_1] \Rightarrow \psi[ev])$. This determines maps $f : A \to B$ for which $\varphi(a) \Rightarrow \psi(f(a))$.

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.

Definition 15. 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 $\mathtt{f}_*$.

Example 16. As an example of contexts which ensure implications across substitution, we can construct the "magic wand" of separation logic [24]. 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 17. 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:

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

The authors do not know of existing literature on this right adjoint; we do not yet how it is connected with dependent products of $\mathcal{P}(\mathsf{T})$. We know it is highly expressive: using reification, separation logic can be generalized from pairs to *functions of predicates*. We do not know if this has been studied.

In addition, the category of predicates has all limits and colimits, by a result of [34]. These can be used to form modalities, inductive and coinductive types, and more.

Proposition 18. $\Omega\mathcal{P}(\mathsf{T})$ is complete and cocomplete, and π_Ω preserves limits and colimits. They are computed pointwise; letting π, ι represent the cone and cocone:

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

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 [33].

Proposition 19. 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*. This generalizes subsets to indexed sets: fibers over A are expanded from truth values to sets, denoted $p^a_{\mathtt{S}}$ or $P_{\mathtt{S}}[a]$, and fibers over $\mathcal{P}(\mathsf{T})$ are expanded from posets to categories.

Each term constructor $\mathtt{f} : \mathtt{S} \to \mathtt{T}$ defines a dependent type $y(\mathtt{f}) : y(\mathtt{S}) \to y(\mathtt{T})$, still denoted $\mathtt{f}$. Its terms are like the principal sieve $\langle \mathtt{f} \rangle : y(\mathtt{T}) \to \mathsf{Prop}$, except that the substituted terms u are recorded.

$$\mathtt{f}_{\mathtt{R}}[t] = \Sigma u{:}\mathtt{R} \to \mathtt{S}.\, (f(u) = t)$$

Proposition 20. Let CCT be the category of complete and cocomplete 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, $p[f]^a_{\mathtt{S}} := \Delta^f(p)^a_{\mathtt{S}} = p_{\mathtt{S}}^{f_{\mathtt{S}}(a)} = p_{\mathtt{S}}^{-1}(f_{\mathtt{S}}(a))$. Dependent sum Σ_f and dependent product Π_f have the same formulae as predicates, and they commute with substitution.

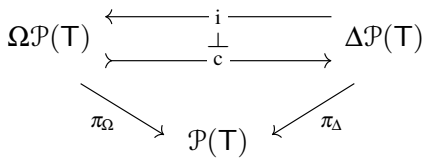
$$\Sigma_f(\varphi)^b_{\mathtt{S}} := \quad \Sigma a{:}A_{\mathtt{S}}.\ (f_{\mathtt{S}}(a) = b) \wedge \varphi(a) \qquad \Pi_f(\varphi)^b_{\mathtt{S}} := \quad \Pi(u{:}\mathtt{R} \to \mathtt{S}).\Pi a{:}A.\ (f_{\mathtt{R}}(a) = b) \Rightarrow \varphi(a)$$

The Grothendieck construction of Δ determines a category over $\mathcal{P}(\mathsf{T})$.

Definition 21. 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 22. The codomain fibration π_Δ is a *closed comprehension category* [17, Sec. 10.5] which is cosmic, i.e. cartesian closed, complete and cocomplete.

The two fibrations are connected by an adjunction: 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** [17, Sec. 11.6].



We denote by HDTΣ the 2-category of higher-order dependent type theories, a full sub-2-category of adjunctions in the 2-category of fibrations.

The reason for the notation HDTΣ is as follows. Recall that the 2-functor

$$\mathcal{P} : \lambda\mathrm{Thy}^{\mathsf{op}}_{=} \to \mathrm{Topos}$$

sends $\mathsf{F} : \mathsf{S} \to \mathsf{T}$ to precomposition by F, $\mathcal{P}(\mathsf{F}) := - \circ \mathsf{F} : [\mathsf{T}^{\mathsf{op}}, \mathrm{Set}] \to [\mathsf{S}^{\mathsf{op}}, \mathrm{Set}]$. This functor preserves pullbacks, inducing morphisms of predicate and codomain fibrations; but it is not locally cartesian closed, nor does it preserve the subobject classifier. It is future work to consider theory translations for which $\mathcal{P}(\mathsf{F})$ preserves Π and Ω [18, C 3].

Theorem 23. 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 $\mathcal{L} : \mathrm{Topos} \to \mathrm{HDT}\Sigma$.

4 Native Type Theory

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

The system is *higher-order dependent type theory* [17, Sec. 11.5] "parameterized" by T. We do not present Equality and Quotient types. We encode Subtyping, Hom, Reification, and Inductive types, which we use in applications.

The type system has **predicates** $\mathsf{x}{:}\Gamma \vdash \varphi : \mathsf{Prop}$ and **types** $\mathsf{x}{:}\Gamma \vdash \mathsf{A} : \mathsf{Type}$, interpreted as $\varphi : \Gamma \to \Omega$ and $\mathsf{p} : \mathsf{A} \to \Gamma$. A term judgement is of the form $\mathsf{x}{:}\Gamma, \mathsf{a}{:}\mathsf{A} \vdash \mathsf{N} : \mathsf{B}[\mathsf{M}]$, 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* [9].

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.

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

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

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

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

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

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

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

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

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

We derive existential ∃ from Σ and universal ∀ 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.

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

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

⊆ **Subtyping** of predicates is defined $(\varphi \subseteq \psi) := \forall a{:}A.\ \varphi(a) \Rightarrow \psi(a)$.

→ **Hom type** (def. 14) of $A_1 \vdash B_1 : \mathsf{Type}$ and $A_2 \vdash B_2 : \mathsf{Type}$ is defined $\Pi x{:}A_1.\ B_1[\pi] \Rightarrow B_2[\mathsf{ev}]$.

R **Reification** (def. 17) $\chi.F : [A, B] \to \mathsf{Prop}$ is defined $\Pi\varphi{:}[A \to \mathsf{Prop}].\ \varphi \Rightarrow F(\varphi[-])$.

μ **Inductive type** of $F : [A, \mathsf{Prop}] \to [A, \mathsf{Prop}]$: the least and greatest fixed points are defined

$$\begin{array}{rcl} \mu\varphi.F(\varphi) & := & \exists\varphi{:}[A, \mathsf{Prop}].\ (\varphi \subseteq F(\varphi)) \Rightarrow \varphi \\ \nu\varphi.F(\varphi) & := & \forall\varphi{:}[A, \mathsf{Prop}].\ (F(\varphi) \subseteq \varphi) \Rightarrow \varphi \end{array}$$

These are used to form data structures and modalities; we can generalize to W-types [29].

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

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

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

We add rules that $\mathsf{F}^* : \mathcal{P}(\mathsf{T}_2) \to \mathcal{P}(\mathsf{T}_1)$ preserves substitution, dependent pair, and co/limits.

To further research we leave the rules for the colax preservation of Π and Prop, and the rules for the two covariant functors $\mathrm{Lan}_{\mathsf{F}}, \mathrm{Ran}_{\mathsf{F}} : \mathcal{P}(\mathsf{T}_1) \to \mathcal{P}(\mathsf{T}_2)$ given by left and right Kan extension.

As a small demonstration of the type system, 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 $\mathtt{f}$.

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

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 $\mathcal{P}(\mathsf{T})$.

Behavior subsystems

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

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

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 24. Let $\mathsf{Th.Gph} \to \mathsf{Th}.\rho\pi$ be the structured λ-theory of the $\rho\pi$-calculus (ex. 5). In $\mathcal{LP}(\mathsf{Th}.\rho\pi)$, suppose we have a name predicate $\alpha : \mathtt{N} \to \mathsf{Prop}$, a process predicate $\varphi : \mathtt{P} \to \mathsf{Prop}$, and a constructor of predicates $F : [\mathtt{N} \to \mathsf{Prop}] \to [\mathtt{P} \to \mathsf{Prop}]$. Then $\mathtt{comm}(\alpha, \varphi, \chi.F) : [\mathtt{E}, \mathsf{Prop}]$ determines communications

- on channels in namespace α
- sending data in codespace φ
- and continuing in contexts $\lambda x.c : [\mathtt{N}, \mathtt{P}]$ such that $\chi(@p) \Rightarrow F(\chi)(c[@p/x])$.

This provides highly expressive specification and conditioning of communication on a network. In particular, these predicates could be used to analyze, distribute, or enforce rules for execution on a public computing platform, such as a blockchain.

Modalities

We can express *temporal modalities* to reason about past and future behavior. Applying the "step" operators of ex. 13 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.

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

Similarly for F, we can condition past behavior. These modalities can also be restricted to subsystems.

Example 25. We can use modalities to express system requirements, such as the capacity to receive and process input on certain channels, or the guarantee to only communicate on certain channels.

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

By proving $\mathtt{in}(n, \lambda x.c) : \mathtt{in}(\mathtt{N}, \chi.\mathsf{safe})$, we know the program will be secure on the channel it receives.

Behavioral equivalence

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 *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 26. Processes in the $\rho\pi$-calculus interact in parallel $-|-$. The basic actions are input and output. To construct the transition system of these observable behaviors, we define 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

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

Hence the predicate which is usually written as $p \xrightarrow{\lambda x.c} q$, meaning "substituting p into c rewrites to 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

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

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

Refined binding

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

Example 27. In the $\rho\pi$-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. 15) of the $\rho\pi$-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 $\rho\pi$-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.

Reasoning about contexts

A ubiquitous 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 28. 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 similarly detect if a program may not remain single-threaded. Let $\mathsf{s.thr} := \neg[0] \wedge \neg[\neg[0] \mid \neg[0]]$. 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 for the act transition system (ex. 26).

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.

These kind of predicates in the $\rho\pi$-calculus have been studied for object capabilities [27], advocating for better security by determining authority purely through object references.

Example 29. A key concept in concurrency is that of a *race condition*, in which multiple communications on one channel are possible simultaneously.

$$\mathsf{race.out} \quad := \exists n : \mathtt{N}.\ \mathtt{out}(n, \mathtt{P}) \mid \mathtt{out}(n, \mathtt{P}) \mid \mathtt{in}(n, \mathtt{N.P}) \mid \mathtt{P}$$

We can use native types to design algorithms which identify and manage these conditions. Let $\mathtt{comm}(n) := \lambda p.(p \mid s(\mathtt{comm}(n, \mathtt{P}, \mathtt{N} \to \mathtt{P})))$, contexts with potential communication on n.

$$\mathtt{comm}(n) \quad \subseteq \quad [\mathtt{out}(n, \mathtt{P}), \mathsf{race.out}] \qquad (\mathtt{comm}(n) \mid \mathtt{out}(n, \mathtt{P})) \quad \subseteq \quad \mathsf{race.out}$$

This is useful especially in applications such as blockchain.

Translating across language paradigms

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

We give a translation $\tau : \mathsf{Th.N}\lambda \to \mathsf{Th}.\pi$ from the name-passing λ-calculus into the π-calculus.

Example 30. Name-passing λ-calculus [11] (abridged)

$$\mathtt{V} \quad \text{variables} \qquad \mathtt{T} \quad \text{terms} \qquad \mathtt{E} \quad \text{rewrites of terms } (+\mathsf{Th.Cat})$$

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

$$\begin{array}{llll} \beta: & [\mathtt{V} \to \mathtt{T}], \mathtt{V} \to \mathtt{E} & \beta(Q, y): & \mathtt{app}(\mathtt{lam}(Q), y) \rightsquigarrow Q(y) \\ \phi: & \mathtt{V}, \mathtt{T}, \mathtt{T} \to \mathtt{E} & \phi(x, Q): & \mathtt{C}(x, Q, \mathtt{var}(x)) \rightsquigarrow Q \end{array}$$

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.

Example 31. Polyadic asynchronous π-calculus [30] (abridged)

$$\mathtt{N} \quad \text{names} \qquad \mathtt{P} \quad \text{processes} \qquad \mathtt{E} \quad \text{rewrites between processes } (+\mathsf{Th.Cat})$$

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

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

The π-calculus [28] models concurrent processes which compute via *communication*, or the exchange of "names". It is like the $\rho\pi$-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 32. There is a translation $[\![-]\!] : \mathsf{Th.N}\lambda \to \mathsf{Th}.\pi$ as follows.

On sorts, $[\![\mathtt{V}]\!] = \mathtt{N}$, $[\![\mathtt{T}]\!] = [\mathtt{N} \to \mathtt{P}]$, and $[\![\mathtt{Hom}_{\mathtt{V}}]\!] = \mathtt{Hom}_{\mathtt{P}}$.

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

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

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

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

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

The translation preserves equations and rewrites. This induces a functor $\mathcal{P}(\mathsf{Th}.\pi) \to \mathcal{P}(\mathsf{Th}.\mathsf{N}\lambda)$, 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 interaction between agents in a network that is highly nondeterministic. By the translation, ir is mapped to a λ-calculus predicate by precomposition; 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 can allow for significant optimization in network computing; e.g. agents trying to reach consensus about side effects. 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 few small examples are only a modest selection of the applications of native type theory. Native types are practical because they are basic: they are made by logic from languages we already use. We encourage the reader to explore what native types can do for you.

6 Conclusion

Native type theory is a method to generate expressive type systems for a broad class of languages. We believe that integrating native type systems in software can help 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*. Available at https://flow.org/.

[2] *Google Closure Compiler*. Available at https://developers.google.com/closure/compiler.

[3] *Hoogle*. Available at https://hoogle.haskell.org/.

[4] *K Framework*. Available at http://www.kframework.org/.

[5] *KJS: A Complete Formal Semantics of JavaScript*. Available at https://github.com/kframework/javascript-semantics.

[6] *Microsoft TypeScript*. Available at https://www.typescriptlang.org/.

[7] *RChain*. Available at https://www.rchain.coop/.

[8] Steve Awodey (2010): *Category Theory*, 2nd edition. Oxford University Press, Inc., USA.

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

[10] H. P. Barendregt (1984): *The Lambda Calculus: Its Syntax and Semantics*. Elsevier.

[11] Gérard Boudol (1997): *The π-calculus in direct style*. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL*, ACM Press, doi:10.1145/263699.263726.

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

[13] Thierry Coquand & Gérard Huet (1988): *The calculus of constructions*. *Information and Computation* 76(2-3), pp. 95–120, doi:10.1016/0890-5401(88)90005-3.

[14] Roy L. Crole (1994): *Categories for Types*. Cambridge University Press, doi:10.1017/CBO9781139172707.

[15] Robert Harper (2016): *Practical Foundations for Programming Languages*, 2 edition. Cambridge University Press, doi:10.1017/CBO9781316576892.

[16] André Hirschowitz, Tom Hirschowitz & Ambroise Lafont (2020): *Modules over monads and operational semantics*. arXiv:2012.06530.

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

[18] Peter T. Johnstone (2002): *Sketches of an Elephant: A Topos Theory Compendium: 2 Volume Set*. Oxford University Press UK.

[19] J. Lambek & P. J. Scott (1986): *Introduction to Higher Order Categorical Logic*. Cambridge University Press, USA.

[20] F. William Lawvere (1969): *Adjointness in Foundations*. *dialectica* 23(3-4), pp. 281–296, doi:10.1111/j.1746-8361.1969.tb01194.x.

[21] Saunders MacLane (1971): *Categories for the Working Mathematician*. Springer New York, doi:10.1007/978-1-4612-9839-7.

[22] Saunders MacLane & Ieke Moerdijk (1994): *Sheaves in Geometry and Logic*. Springer New York, doi:10.1007/978-1-4612-0927-0.

[23] Per Martin-Löf (1998): *An intuitionistic theory of types*. In: *Twenty Five Years of Constructive Type Theory*, Oxford University Press, doi:10.1093/oso/9780198501275.003.0010.

[24] Paul-André Melliès & Noam Zeilberger (2015): *Functors are Type Refinement Systems*. *ACM SIGPLAN Notices* 50(1), pp. 3–16, doi:10.1145/2775051.2676970.

[25] L. G. Meredith & Matthias Radestock (2005): *Namespace Logic: A Logic for a Reflective Higher-Order Calculus*. In: *Trustworthy Global Computing*, Springer Berlin Heidelberg, pp. 353–369, doi:10.1007/11580850_19.

[26] L.G. Meredith & Matthias Radestock (2005): *A Reflective Higher-order Calculus*. *Electronic Notes in Theoretical Computer Science* 141(5), pp. 49–67, doi:10.1016/j.entcs.2005.05.016.

[27] Lucius G Meredith, Mike Stay & Sophia Drossopoulou (2013): *Policy as Types*. arXiv:1307.7766.

[28] Robin Milner (1993): *The Polyadic π-Calculus: a Tutorial*. In: *Logic and Algebra of Specification*, Springer Berlin Heidelberg, pp. 203–246, doi:10.1007/978-3-642-58041-3_6.

[29] Ieke Moerdijk & Erik Palmgren (2000): *Wellfounded trees in categories*. *Annals of Pure and Applied Logic* 104(1-3), pp. 189–218, doi:10.1016/s0168-0072(00)00012-9.

[30] Davide Sangiorgi (2000): *Communicating and Mobile Systems: the π-calculus,*. *Science of Computer Programming* 38(1-3), pp. 151–153, doi:10.1016/s0167-6423(00)00008-3.

[31] Dana S. Scott (1980): *Relating Theories of the Lambda Calculus*. Available at https://www.andrew.cmu.edu/user/awodey/dump/Scott/ScottRelating.pdf.

[32] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau & Théo Winterhalter (2019): *Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq*. Proc. ACM Program. Lang. 4(POPL), doi:10.1145/3371076.

[33] Ross Street (1974): *Elementary cosmoi I*. In Gregory M. Kelly, editor: Category Seminar, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 134–180, doi:10.1016/0022-4049(72)90019-9.

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

[35] D. Turi & G. Plotkin: *Towards a mathematical operational semantics*. In: Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science, IEEE Comput. Soc, doi:10.1109/lics.1997.614955.