Native Type Theory

Author(s) anonymized

***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 a rich form of dependent type theory. Any language with products and functions can be modelled as a higher-order algebraic theory with rewrites, and the internal language of its presheaf topos provides complete specification of the structure and behavior of terms. This construction is functorial, so that translations of languages give translations of type systems. The construction therefore provides a shared framework for higher-order reasoning about most existing programming languages.**

I. Introduction

As society becomes rooted in network computing, it is vital to develop general methods of reasoning about code. Type theory is growing as a guiding philosophy in the design of programming languages; but in practice type systems are mostly heterogeneous, and there are not standard ways to reason across all languages. In this paper we construct for any language a *native type system* which provides complete specification of the structure and behavior of terms.

Categorical logic [1] unifies languages: any formalism, from a simple heap to the calculus of constructions, can be modelled as a structured category. In doing so, we inherit a wealth of tools from category theory. In particular, we construct native type systems by composing ideas known for decades:

$$\mathtt{language} \xrightarrow{\Lambda} \mathtt{category} \xrightarrow{\mathcal{P}} \mathtt{topos} \xrightarrow{\Phi} \mathtt{type\ system}$$

The first Λ forms the *syntactic category* of a language [2]; the second $\mathcal{P}$ is the *presheaf construction* [3]; and the third Φ is the concept of *logic over a type theory* [1]. The composite is 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 and a rich form of dependent type theory. For example, the following predicate on processes in a concurrent language is effectively a compile-time firewall.

$$\mu\mathtt{X}.\ ((\mathtt{in}(\alpha,\mathtt{N.X}) \mid \mathtt{P}) \wedge \neg(\mathtt{in}(\neg\alpha,\mathtt{N.P}) \mid \mathtt{P}))$$

“Can always input from channels in the set α and cannot input from $\neg\alpha$.”

Native type theory is intended to be a practical framework to equip all programming languages with a shared system for higher-order reasoning. The authors believe that the possibilities are significant and broad, and we encourage the development of tools and algorithms to bring them to fruition.

A. Motivation and Implementation

Why do we want native type systems? How can they be implemented practically? What are the intended applications?

ECMAScript is a dynamic, weakly-typed language built into every web browser. When code breaks, it does so at runtime; but software companies prefer to know in advance. They know from experience that static type systems are necessary for correct and maintainable code. Microsoft’s TypeScript [4], Facebook’s Flow [5], and Google’s Closure Compiler [6] are all multi-million dollar efforts to retrofit ECMAScript with a strong, static type system. None of them is sound [7].

By contrast, the native type system of ECMAScript is sound by construction: it conditions code explicitly by its structure and behavior. By specifying both the language and the platforms on which it runs, ECMAScript can be formally integrated into composite systems, to reason in the practical context of communicating with a back-end server (**??**).

We plan to implement native type theory as a method which inputs the formal specification of a language and outputs —. To do so, we plan to utilize the significant progress that has been made in formalization.

K Framework [8] is a popular formal verification tool. The complete semantics of many popular languages have been expressed using K, including ECMAScript, C, Java, Python, Haskell, LLVM, Solidity, and more. While K has its own language for expressing semantics, it corresponds closely to the notion of a “higher-order algebraic theory with rewrites”, which we explore in section II.

The most obvious way to use the resulting type system is to annotate code, but one can also search by types. The search engine Hoogle [9] enables the search of Haskell libraries on Stackage by function signature. This system can be expanded to many languages, and strengthened by more expressive types. For example, if $\varphi : \mathtt{S} \to \mathsf{Prop}$ is a type of program and $\psi : \mathtt{T} \to \mathsf{Prop}$ is a security property, there is a “rely-guarantee” operator to form the type of $\mathtt{S.T}$-contexts for which substituting a program in φ guarantees the property ψ (III-A).

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

Hence we can query “all contexts where this kind of algorithm can be run safely”. Imagine being able to search any public codebase by pre- and postcondition.

Of course, these applications require substantial development. Most basic is the need for efficient type-checking, such as for inductive types, but much has been done for this. To be usable, we need conversion between existing types and native types, as well as libraries of native types, so anyone can say useful things 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.

B. Origin and Connected Ideas

There are many kinds of language-specific logics which share generally the same form. The present work began in seeking to generate these logics for concurrency.

$$\text{T-logic} = \text{T-constructors} + \text{predicate logic} \ + \ \text{recursion}$$

Hennessy-Milner logic [10] uses formulae that determine when processes are bisimilar. Spatial-behavioral logic [11] specifies system properties such as secrecy and usage of resources [12]. Namespace logic [13] for the ρ-calculus [14] (II-C2) expands these by also reasoning about data and channels.

In developing our approach, we found closely related work. Matching μ-Logic [15] is the basis of K Framework [8], "a rewrite-based executable semantic framework in which programming languages, type systems, and formal analysis tools can be defined". It is based on "patterns", formulae made from term constructors, predicate logic, and recursion.

The present work generalizes the logic to include binding operations, and gives a categorical foundation which provides substantial expansions, such as co/limits and dependent types. The precise relation to Matching Logic is yet to be expounded.

In "Functors are Type Refinement Systems" [16], Melliès and Zeilberger introduce a syntax for reasoning about functors $F : \mathrm{D} \to \mathrm{T}$, and use this to distinguish two notions of type.

Intrinsic types are the sorts of a language, $\mathtt{S} : \mathrm{T}$, i.e. the symbols in a BNF grammar or the parts of the total state of a machine. In the π-calculus, there are distinct sorts $\mathtt{N}$ and $\mathtt{P}$ for name and process. In a grammar of a 80x86 CPU, there are distinct sorts for the stack and the heap.

Refinement types are predicates on a sort, $\varphi : \mathtt{S} \to \mathsf{Prop}$. In Hoare logic [17], assertions are predicates on the valuation of a memory heap. The triples $\{\varphi\}C\{\psi\}$, which express that if φ holds and command C is applied then ψ holds, are modelled as the preimage $F^{-1}(C)$.

Static type systems have historically focused on intrinsic types, but refinement types are growing in popularity. Template meta-programming in C++ allows checking propositions at compile time [18]. TypeScript has type guards that refine union types and superclasses to subclasses [19]. Haskell has various libraries that provide compile-time refinement types [**?**].

The natural setting for this type-predicate distinction is a topos, a category with a canonical functor into it called a *predicate fibration*. The modelling of type theories with toposes is well-established; pertinent here is the Hyland–Pitts model of the Calculus of Constructions [20], and more generally the notion of classifying topos [21, X].

Christian: higher-order unification.

C. Contribution and Organization

The primary goal of the paper is simply to show that composing a few well-known ideas in categorical logic can be very useful to computer science. In the process, we emphasize many facts that are known in theory but not used in practice; we also define and prove the existence of several new constructions. (— less tt)

We give two sections to develop the categorical foundation, one section to present the type system, and one section to demonstrate some applications. The reader focused on the latter half may go to IV. The sections are organized as follows.

II. Higher-order algebraic theories [22] are a general framework for modelling languages with product and function types. We define a new category HOAT, and introduce a new concept of theories with rewriting.

III. A theory embeds into a category with rich internal logic, called a presheaf topos. The cartesian closed structure of predicates and types is very expressive; we give a new way to form dependent functions called reification. We prove that the predicate and codomain fibrations form a functor from presheaf toposes to structures for cartesian closed, complete and cocomplete full higher-order dependent type theory.

IV. The native type system is presented as the internal language of the topos structure. The system is an extension of *full higher-order dependent type theory* [1], as in the Calculus of Constructions [23]. We outline the basic features, their semantics and use.

V. There are a few kinds of applications which we demonstrate: homs (rely-guarantee), modalities (must/can), well-formedness (termination/complexity), and functoriality (translations/combined semantics).

Christian: notation, acknowledgements

II. Higher-order algebraic theories

The syntax of a programming 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.[1] Algebraic theories are a class of such categories, which provide a general formalism of algebraic structures [24]. Recently, they have been generalized to include higher-order operations of logic and computer science [22]. We give an overview of higher-order algebraic theories: their syntax and structure, the category they form, and motivating examples.

Algebraic structures, defined by operations and equations, can be defined in any cartesian category. The 2-category of cartesian categories has been studied as the abstract setting of multisorted universal algebra [24]. These categories model many structures found in mathematics and computer science.

Example 1. The algebraic theory of a $\mathtt{stack}$ of sort $\mathtt{A}$ is presented as follows.

$$\begin{array}{rcl rcl} \mathtt{push}: & \mathtt{S}\times\mathtt{A} & \to \mathtt{S} & \mathtt{pop}: & \mathtt{S} & \to \mathtt{S} \\ \mathtt{emp}: & 1 & \to \mathtt{S} & \mathtt{top}: & \mathtt{S} & \to \mathtt{A} \end{array}$$

$$\langle \mathtt{pop}(\mathtt{push}\langle l,i\rangle), \mathtt{top}(\mathtt{push}\langle l,i\rangle)\rangle = \langle l,i\rangle$$

In logic and computer science, there are *higher-order* operations whose operands are themselves operations, such as universal quantification with predicates and λ-abstraction with expressions. Higher-order operations bind the variables of their operands, which establishes the concept of evaluation.

[1] Equivalently, a BNF grammar generates a category in which an object is a symbol and a morphism is a production rule.

Binding and capture-avoiding substitution are formalized by monad strength with respect to a monoidal structure on presheaves [25]. But in computer science, it is standard to use *exponentials* to represent variable-binding operators. This practice was formally justified by Fiore and Mahmoud [26], who proved the binding algebraic structure of Fiore, Plotkin, and Turi [25] to be equivalent to exponential structure.

To represent a higher-order operation, a cartesian category is equipped with an exponentiable object $\mathtt{S}$. Then $\mathtt{T}^{\mathtt{S}}$ represents terms of sort $\mathtt{T}$ with free variable of sort $\mathtt{S}$; hence an operation $\mathtt{T}^{\mathtt{S}} \to \mathtt{U}$ inputs terms of the form $\lambda x : \mathtt{S}.f(x) : \mathtt{T}$, and binds x. The definition of $\mathtt{T}^{\mathtt{S}}$ is characterized by the evaluation map, which substitutes terms of sort $\mathtt{S}$ for the free variable.

Definition 2. Let T be a cartesian category. An object $\mathtt{S} \in \mathrm{T}$ is **exponentiable** if the functor $\mathtt{S} \times -$ has a right adjoint $(-)^{\mathtt{S}}$. We may denote $\mathtt{T}^{\mathtt{S}}$ as $[\mathtt{S}, \mathtt{T}]$ for readability.

We denote "currying" $\mathtt{f} : \mathtt{U} \times \mathtt{S} \to \mathtt{T}$ by $\lambda x.\mathtt{f} : \mathtt{U} \to \mathtt{T}^{\mathtt{S}}$, and the counit by $ev_{\mathtt{S},\mathtt{T}} : \mathtt{S} \times \mathtt{T}^{\mathtt{S}} \to \mathtt{T}$, written in syntax as substitution: $ev_{\mathtt{S},\mathtt{T}}(u, \lambda x.\mathtt{f}) := \mathtt{f}[u/x]$.

A cartesian functor $F : \mathrm{T}_1 \to \mathrm{T}_2$ **preserves** an exponentiable object $\mathtt{S}$ if $F(\mathtt{S})$ is exponentiable and for all $\mathtt{T}$

$$\lambda x.(ev_{\mathtt{S},\mathtt{T}} \circ \langle \pi_{\mathtt{S}}, \pi_{\mathtt{T}^{\mathtt{S}}} \rangle^{-1}) : F(\mathtt{T}^{\mathtt{S}}) \simeq F(\mathtt{T})^{F(\mathtt{S})}.$$

We say F is **cartex** if it preserves all exponentiable objects.

A higher-order algebraic theory is essentially a cartesian category equipped with objects which are exponentiable "up to a certain order". For example, the control operators of [27] are third-order operations of the form $\mathtt{T}^{(\mathtt{T}^{\mathtt{S}})} \to \mathtt{S}$ and $\mathtt{S}^{(\mathtt{T}^{\mathtt{S}})} \to \mathtt{S}$.

Hence the context for higher-order algebraic theories is contained in that of algebraic theories, because we require the functors to preserve the exponentiable objects.

Definition 3. The 2-category of cartesian categories, cartex functors, and natural transformations is denoted CartEx.

The 2-category CartEx is one definition of the universe of higher-order algebraic theories. However, more refined structure is necessary for syntactic presentation of theories. In particular, it must be represented explicitly in the theory that every sort is generated by products and exponents from a set of base sorts, denoted $\mathtt{B}_i$.

A. Presentation and syntax

To formalize higher-order presentations, [22, A1] defines the nth-order S-sorted simply typed λ-calculus, meaning that λ-abstraction is limited to order n; we denote this language by $\lambda_n(S)$. Contexts, terms, and substitution define a classifying category $\Lambda_n(S)$, which is a free cartesian category with base sorts in S being exponentiable to order n.

Definition 4. An ***n*th-order *S*-sorted signature** consists of a set of operations O and an arity function $|-| : O \to \Lambda_n(S) \times S$.

The signature gives a syntactic category Λ_O, generated by $\lambda_n(S)$ plus the following axiom schema.

$$\frac{\Gamma \vdash s : \mathtt{S}}{\Gamma \vdash \mathtt{f}(s) : \mathtt{B}} \; (\mathtt{f} \in O, |\mathtt{f}| = (\mathtt{S}, \mathtt{B}))$$

Definition 5. An ***n*th-order *S*-sorted presentation** $\Sigma = (O, |-|, E)$ consists of a signature and a set of equations

$$E \subseteq \textstyle\sum_{(\mathtt{S},\mathtt{T}) \in \Lambda_n(S) \times S} \Lambda_O(\mathtt{S}, \mathtt{T}) \times \Lambda_O(\mathtt{S}, \mathtt{T}).$$

The presentation gives a syntactic category Λ_Σ generated by the signature plus the following axiom schema.

$$\frac{\Gamma \vdash u : \mathtt{S}}{\Gamma \vdash \mathtt{f}_1(u) = \mathtt{f}_2(u) : \mathtt{B}} \; ((\mathtt{S}, \mathtt{B}), (\mathtt{f}_1, \mathtt{f}_2)) \in E$$

Finally, we add the inference rules for equality to be an equivalence relation.

Example 6. The second-order theory of the untyped equational λ-calculus $\mathrm{T}^{\equiv}_{\lambda 2}$ is presented as follows.

$$\begin{array}{lll} \mathtt{l} : & [\mathtt{U}, \mathtt{U}] & \to \mathtt{U} \\ \mathtt{a} : & \mathtt{U} \times \mathtt{U} & \to \mathtt{U} \end{array} \qquad \mathtt{a}(\mathtt{l}(\lambda x.t), u) \equiv t[u/x]$$

Example 7. In a paper which helped to shape functional programming, Wadler introduced a notation for constructing terms of a monad analogous to set comprehension [28]. This idea has been implemented and used in at least Haskell [29], Scala [30], and Python [31], [32]. Monad comprehensions are terms in a second-order algebraic theory.

Let Th be a single-sorted first-order algebraic theory with generating sort $\mathtt{T}$, and let M be a monad on Set. The second-order theory MonComp(Th) of monad comprehensions over Th adjoins to Th a new sort $\mathtt{C}$ and a function symbol for each natural number $i \geq 0$ and object Γ of MonComp(Th):

$$[-|-]_{\mathtt{i},\Gamma} : \quad \mathtt{T}^{\mathtt{T}^i \times \Gamma} \times \mathtt{C}^{\Gamma} \times \mathtt{C}^{\mathtt{T} \times \Gamma} \times \cdots \times \mathtt{C}^{\mathtt{T}^{i-1} \times \Gamma} \to \mathtt{C}.$$

A monad comprehension denotes a morphism into a monadic type. Given a model $F\colon \mathrm{Th} \to \mathrm{Set}$ and a monad M on Set, we can extend F to a model $\tilde{F}\colon \mathrm{MonComp(Th)} \to \mathrm{Set}$ such that $\tilde{F}(\mathtt{C}) = M(F(\mathtt{T}))$.

The expression $[\mathtt{expr}]_{\mathtt{0},\Gamma}\colon \Gamma \to C$, where $\mathtt{expr}$ is any morphism from Γ to T in MonComp(Th), denotes the morphism $\eta^M_{FT} \circ F\mathtt{expr}\colon F\Gamma \to MFT$. The expression

$$\begin{array}{ll} [\mathtt{expr}(\mathtt{x}_1, \ldots \mathtt{x}_n) \mid & \mathtt{x}_1 \leftarrow \mathtt{expr}_0; \\ & \mathtt{x}_2 \leftarrow \mathtt{expr}_1(\mathtt{x}_1); \\ & \cdots \\ & \mathtt{x}_n \leftarrow \mathtt{expr}_{n-1}(\mathtt{x}_1, \ldots, \mathtt{x}_{n-1})]_{\mathtt{n},\Gamma}, \end{array}$$

where $\mathtt{expr}$ is any morphism $\mathtt{T}^n \times \Gamma \to \mathtt{T}$ in Th and $\mathtt{expr}_\mathtt{i}$ is any morphism $T^i \times \Gamma \to C$ in MonComp(Th), denotes the morphism

$$\begin{array}{l} F\Gamma \xrightarrow{\Delta} (F\Gamma)^2 \xrightarrow{f_0 \times F\Gamma} MFT \times F\Gamma \\ \xrightarrow{\Delta} (MFT \times F\Gamma)^2 \xrightarrow{(\mu^M_{FT} \circ Mf_1) \times MFT \times F\Gamma} (MFT)^2 \times F\Gamma \cdots \\ \xrightarrow{\Delta} ((MFT)^{n-1} \times F\Gamma)^2 \xrightarrow{(\mu^M_{FT} \circ Mf_{n-1}) \times MFT^{n-1} \times F\Gamma} (MFT)^n \times F\Gamma \\ \xrightarrow{Mf} MFT \end{array}$$

where $f = \tilde{F}\mathtt{expr}$ and $f_i = \tilde{F}\mathtt{expr}_\mathtt{i}$.

A **transliteration** of presentations $f : (O_1, |-|_1, E_1) \to (O_2, |-|_2, E_2)$ is a function of operations $f : O_1 \to O_2$ which preserves arities and equations. For order n and sorts S, presentations and transliterations form a category $\mathrm{Pre}_n(S)$.

For each presentation Σ, the syntactic category Λ_Σ is equipped with a canonical functor $\tau_\Sigma : \Lambda_n(S) \to \Lambda_\Sigma$, which we take to be the categorical definition of an nth-order S-sorted algebraic theory.

B. The category of higher-order algebraic theories

We define the category of all higher-order algebraic theories, to be the domain of the native type theory construction. To do this, we define the category of n-S theories and show that this is functorial in n and S. We then assemble these into one category, using a tool called the Grothendieck construction.

The definition of single-sorted algebraic theory, a cartesian identity-on-objects functor $\tau : \mathbb{F}^{\mathrm{op}} = \Lambda_0(1) \to \mathrm{T}$, generalizes to many sorts and higher orders.

Definition 8. An ***n*th-order *S*-sorted algebraic theory** or ***n*-*S* theory** (T, τ) is a cartesian category T equipped with a strict cartex identity-on-objects functor $\tau : \Lambda_n(S) \to \mathrm{T}$.

An ***n*-*S* theory morphism** $F : (\mathrm{T}_1, \tau_1) \to (\mathrm{T}_2, \tau_2)$ is a cartex functor $F : \mathrm{T}_1 \to \mathrm{T}_2$ such that $F \circ \tau_1 = \tau_2$. Composition is given by pasting commutative triangles.

We denote the category of n-S theories by $\mathbb{T}_n(S)$.

Example 9. A first-order theory of arithmetic $\mathrm{T}_{\mathtt{Ar}}$ can be presented by $0, \mathtt{s}, +, \times$, and the equations for a commutative rig. There is a well-known translation into the λ-calculus [33]:

$$\begin{array}{lll} 0 \mapsto & \mathtt{l}(\lambda f.\mathtt{l}(\lambda x.x)) & : 1 \to \mathtt{U} \\ \mathtt{s} \mapsto & \mathtt{l}(\lambda f.\mathtt{l}(\lambda x.\mathtt{a}(f, \mathtt{a}(\mathtt{a}(-, f), x)))) & : \mathtt{U} \to \mathtt{U} \\ + \mapsto & \mathtt{l}(\lambda f.\mathtt{l}(\lambda x.\mathtt{a}(\mathtt{a}(-, f), \mathtt{a}(-, \mathtt{a}(f, x))))) & : \mathtt{U}^2 \to \mathtt{U} \\ \times \mapsto & \mathtt{l}(\lambda f.\mathtt{a}(-, \mathtt{a}(-, f))) & : \mathtt{U}^2 \to \mathtt{U}. \end{array}$$

This is a 2-$\{\mathtt{U}\}$ theory morphism $F : (\mathrm{T}_{\mathtt{Ar}}, \tau_{\mathtt{Ar}}) \to (\mathrm{T}^{=}_{\lambda 2}, \tau^{=}_{\lambda 2})$.

To understand the category of n-S theories more concretely, we clarify its relation to presentations.

A transliteration is a map of base operations, but in practice one maps base operations to arbitrary terms. A **translation** of presentations $f : (O_1, |-|_1, E_1) \to (O_2, |-|_2, E_2)$ is a function $f : \prod_{o \in O_1} \Lambda_{O_2}(|o|)$ which preserves equations.

Presentations and translations form a category $\mathrm{Pres}_n(S)$. The functor $\xi : \mathrm{Pre}_n(S) \to \mathbb{T}_n(S)$ which sends Σ to Λ_Σ has a right adjoint $\upsilon : \mathbb{T}_n(S) \to \mathrm{Pre}_n(S)$, which sends a theory $\tau : \Lambda_n(S) \to \mathrm{T}$ to the signature whose operations are the morphisms of T. This adjunction is monadic:

$$\mathbb{T}_n(S) \simeq \mathrm{Kl}(\upsilon \circ \xi) \simeq \mathrm{Pres}_n(S).$$

In [22] this fact is proved and used to show that n-S theories are a coreflective subcategory of $(n+1)$-S theories: there is an adjunction which adds and removes exponentiable structure, and the unit is an isomorphism.

$$\lceil - \rceil \dashv \lfloor - \rfloor : \mathbb{T}_n(S) \simeq \mathrm{Pres}_n(S) \leftrightarrows \mathrm{Pres}_{n+1}(S) \simeq \mathbb{T}_{n+1}(S)$$

This implies that $(n+1)$-S theories correspond to a class of monads on n-S theories, generalizing the theory-monad correspondence which is central to categorical algebra.

We show that $\mathbb{T}$ is functorial in n and S. Let CR be the free coreflection: the free 2-category with $l : a \to b$, $r : b \to a$, $\varepsilon : lr \Rightarrow 1_b$, $\eta : 1_a \simeq rl$ satisfying the triangle identities [34].

Let CR_ω be an ω-chain of copies of CR connected end-to-end. We take $[\mathrm{CR}_\omega, \mathrm{Cat}]$ to be a category of 2-functors and 2-natural transformations.

Proposition 10. There is a functor

$$\lambda n.\mathbb{T} : \mathrm{Set} \to [\mathrm{CR}_\omega, \mathrm{Cat}]$$

such that $\lambda n.\mathbb{T}(S)(n) = \mathbb{T}_n(S)$, the category of n-S theories.

Proof. For each set S of base sorts, $\lambda n.\mathbb{T}(S) : \mathrm{CR}_\omega \to \mathrm{Cat}$ gives an ω-chain of coreflective subcategories.

$$\ldots \leftrightarrows \mathbb{T}_n(S) \leftrightarrows \mathbb{T}_{n+1}(S) \leftrightarrows \ldots$$

For each change of base sorts $f : S_1 \to S_2$, we define $\lambda n.\mathbb{T}(f)(n)$ in terms of presentations

$$\lambda n.\mathbb{T}(f)(n) : \mathbb{T}_n(S_1) \simeq \mathrm{Pres}_n(S_1) \xrightarrow{\bar{f}} \mathrm{Pres}_n(S_2) \simeq \mathbb{T}_n(S_2),$$

given by substitution of base sorts in operations and equations. Then $\lambda n.\mathbb{T}(f)$ is a morphism of adjunctions [?]: one may check for $\mathrm{T} \in \mathbb{T}_n(S_1), \mathrm{U} \in \mathbb{T}_{n+1}(S_1)$ that

$$\begin{array}{rcl} \mathbb{T}_{n+1}(f)(\lceil \mathrm{T} \rceil) & = & \lceil \mathbb{T}_n(f)(\mathrm{T}) \rceil \\ \lfloor \mathbb{T}_{n+1}(f)(\mathrm{U}) \rfloor & = & \mathbb{T}_n(f)(\lfloor \mathrm{U} \rfloor) \\ \mathbb{T}_n(f)(\eta_{S_1}(\mathrm{T})) & = & \eta_{S_2}(\mathbb{T}_{n+1}(f)(\mathrm{T})), \end{array}$$

simply because substitution of base sorts commutes with adding and removing exponentiable structure. □

Uncurrying, we have an indexed category

$$\mathbb{T} : \mathrm{Set} \times \mathrm{CR}_\omega \to \mathrm{Cat}.$$

The image of $\mathbb{T}$ assembles into a single category as follows.

Definition 11. The **Grothendieck construction** [35] of an indexed category $F : C^{\mathrm{op}} \to \mathrm{Cat}$ is a category $C \ltimes F$, defined:

object	$\langle c : C, x : F(c) \rangle$
morphism	$\langle f : c \to d, \alpha : F(f)(x) \to y \rangle$
composition	$\langle g, \beta \rangle \circ \langle f, \alpha \rangle := \langle g \circ f, \beta \circ F(g)(\alpha) \rangle$

This category is equipped with a *fibration* (III) over C which projects onto the first coordinate.

$$\pi_F : C \ltimes F \to C$$

Definition 12. The category of higher-order algebraic theories **HOAT** is defined to be $(\mathrm{CR}_\omega \times \mathrm{Set}) \ltimes \mathbb{T}$.

Hence a **higher-order algebraic theory** $\langle (n, S), (\mathrm{T}, \tau) \rangle$ consists of an order n, base sorts S, and cartesian category with a strict cartex identity-on-objects functor $\tau : \Lambda_n(S) \to \mathrm{T}$.

A **theory morphism** $\langle (i, f), F \rangle$ is a relation of orders $i : n_1 \to n_2$, a function of base sorts $f : S_1 \to S_2$, and a morphism of n_2-S_2 theories $F : \mathbb{T}_i(f)(\mathrm{T}_1, \tau_1) \to (\mathrm{T}_2, \tau_2)$.

Composition of theory morphisms can be inferred from the definition of the Grothendieck construction.

The category HOAT contains all languages from product theories to simple type theories, and all translations between them. Most existing programming languages, data structures, and computing systems are represented in HOAT.

Finally, we define models of a theory: actual implementations of the syntactic structure.

Definition 13. For a higher order algebraic theory $\langle (n, S), (\mathrm{T}, \tau)\rangle$ and a cartesian category C, the **category of models** is defined to be $\mathrm{CartEx}(\mathrm{T}, \mathrm{C})$. These assemble into a category of all higher-order models.

In the present work we focus on native type theory for higher-order algebraic theories, and not yet for models thereof. However, we note that given an n-S theory $\tau : \Lambda_n(S) \to \mathrm{T}$, a model $M : \mathrm{T} \to \mathrm{C}$ can be converted into (a non-evil notion of) an n-S theory by taking the bijective-on-objects factor [36] of the composite $M \circ \tau : \Lambda_n(S) \to \mathrm{C}$.

The category HOAT is the domain in which to generate native type theories. Prior to the construction, we describe motivating examples of higher-order theories.

C. Application: theories with rewrites

One major aspect of computing that higher-order algebraic theories do not explicitly represent is *dynamics*. In practice, function evaluation is not an equation but a *computation*. There are several ways to incorporate dynamics into languages [37]; we introduce a new method, which is unusual but beneficial.

Rewrite systems can be modelled by theories that include an internal graph, *i.e.* two sorts $\mathtt{E}$ for edges and $\mathtt{V}$ for vertices, together with two operations $s, t \colon \mathtt{E} \to \mathtt{V}$ that encode the source and target of each edge. Such theories construct terms as vertices and rewrites as edges.

Let $\{\mathtt{S}_i\}_i$ be a family of sorts. Given an n-ary edge constructor and two n-ary morphisms into $\mathtt{V}$

$$\begin{array}{rl} \mathtt{r}: & \prod_{i=0}^{n} \mathtt{S}_i \to \mathtt{E} \\ \mathtt{f}, \mathtt{g}: & \prod_{i=0}^{n} \mathtt{S}_i \to \mathtt{V}, \end{array}$$

we write an abbreviation for the pair of equations

$$\begin{array}{rcrcl} \mathtt{r}(\vec{u}) : \mathtt{f}(\vec{u}) \rightsquigarrow \mathtt{g}(\vec{u}) & := & s(\mathtt{r}(\vec{u})) & = & \mathtt{f}(\vec{u}) \\ & \wedge & t(\mathtt{r}(\vec{u})) & = & \mathtt{g}(\vec{u}). \end{array}$$

By representing a rewrite as an operation within a theory, we gain control of its implementation: we can add sorts, operations, and equations which account for resource usage, reduction strategies, and other aspects of computing.

We focus on the second: since not all rewrites happen at the top level of a term, a theory may include equations that propagate the reduction context inward.

1) λ-calculus: The second-order theory of the untyped λ-calculus $\mathrm{T}_{\lambda 2}$ has two sorts: $\mathtt{U}$ for terms and $\mathtt{R}$ for rewrites.

Example 14. λ-calculus

$$\begin{array}{rlcrrl} \mathtt{a}: & \mathtt{U} \times \mathtt{U} & \to \mathtt{U} & \beta: & [\mathtt{U},\mathtt{U}] \times \mathtt{U} & \to \mathtt{R} \\ \mathtt{l}: & [\mathtt{U},\mathtt{U}] & \to \mathtt{U} & \mathtt{r}_{\mathtt{a}}: & \mathtt{R} \times \mathtt{U} & \to \mathtt{R} \\ & & & s,t: & \mathtt{R} & \to \mathtt{U} \end{array}$$

$$\begin{array}{rl} \beta(\lambda x.t, u): & \mathtt{a}(\mathtt{l}(\lambda x.t), u) \rightsquigarrow t[u/x] \\ \mathtt{r}_{\mathtt{a}}(r, u): & \mathtt{a}(s(r), u) \rightsquigarrow \mathtt{a}(t(r), u) \end{array}$$

The rewrite constructor $\mathtt{r}_{\mathtt{a}}$ restricts rewrites to the first coordinate, and propagates reduction contexts inward to head position. This prohibits reductions under an abstraction and on the right side of an application, giving a "head normal form" for terminating terms.

2) ρ-calculus: The present work originated in an effort to develop logics for concurrent languages. RChain [38] is a distributed computing system based on the ρ-calculus, or **r**eflective **h**igher-**o**rder π-calculus.

We can model the ρ-calculus[2] [14] as a second-order theory T_ρ with two sorts, $\mathtt{P}$ for processes and $\mathtt{N}$ for names. The sorts act as code and data respectively, and the operations of reference @ and dereference $*$ transform one into the other. There is a third sort $\mathtt{R}$ for rewrites.[3]

Terms are built up from one constant, the null process $\mathtt{0}$. The two basic actions of a process are output $\mathtt{out}$ and input $\mathtt{in}$, and parallel $-|-$ gives binary interaction: these earn their names in the communication rule.

Example 15. ρ-calculus

$$\begin{array}{rlcrlc} \mathtt{0}: & 1 & \to \mathtt{P} & -|-: & \mathtt{P} \times \mathtt{P} & \to \mathtt{P} \\ @: & \mathtt{P} & \to \mathtt{N} & \mathtt{out}: & \mathtt{N} \times \mathtt{P} & \to \mathtt{P} \\ *: & \mathtt{N} & \to \mathtt{P} & \mathtt{in}: & \mathtt{N} \times [\mathtt{N},\mathtt{P}] & \to \mathtt{P} \\ \\ s,t: & \mathtt{R} & \to \mathtt{P} & \mathtt{comm}: & \mathtt{N} \times \mathtt{P} \times [\mathtt{N},\mathtt{P}] & \to \mathtt{R} \\ -\rfloor-: & \mathtt{R} \times \mathtt{P} & \to \mathtt{R} & -\lfloor-: & \mathtt{P} \times \mathtt{R} & \to \mathtt{R} \end{array}$$

$$\begin{array}{rcl} \mathtt{comm}(n, q, \lambda x.p) & : & \mathtt{out}(n, q)|\mathtt{in}(n, \lambda x.p) \rightsquigarrow p[@q/x] \\ (\mathtt{P}, -|-, \mathtt{0}) & & \text{commutative monoid} \end{array}$$

$$* \circ @ \equiv 1_{\mathtt{P}}$$

$$\begin{array}{rcl} r\rfloor p & : & s(r)|p \rightsquigarrow t(r)|p \\ -\rfloor- & & \text{commutative monoid action} \end{array}$$

$$p\lfloor r \;\equiv\; swap_{\mathtt{P},\mathtt{R}} \circ r\rfloor p \;\circ swap_{\mathtt{R},\mathtt{P}}$$

In the predicate logic of $\mathcal{P}(\mathrm{T}_\rho)$, a predicate on names $\alpha : \Omega(\mathtt{N})$ is called a **namespace** [13], and a predicate on processes $\varphi : \Omega(\mathtt{P})$ is called a **codespace**.

In the following sections we derive a native type system from such higher-order algebraic theories with rewrites, to reason about the structure and behavior of terms.

III. The Logic of a Presheaf Topos

Topos theory [21] revolutionizes mathematics by expanding the domain of higher-order predicate logic and dependent type theory beyond sets and function. Most useful is the fact that every category embeds into a topos, called the topos of presheaves. For any higher-order algebraic theory, the internal language of its presheaf topos is its native type system.

Let T be a higher-order algebraic theory. A **presheaf** on T is a functor $A : \mathrm{T}^{\mathrm{op}} \to \mathrm{Set}$. The category of presheaves is the functor category $[\mathrm{T}^{\mathrm{op}}, \mathrm{Set}]$, which we denote $\mathcal{P}(\mathrm{T})$.

[2]This theory differs slightly from the original presentation in that we use bound variables rather than semantic substitution [14, section 2.7].

[3]When the ρ-calculus is in a system which includes a parallel processor, instead of $\rfloor, \lfloor$ we can have $-|_{\mathtt{R}}- : \mathtt{R}^2 \to \mathtt{R}$ which puts rewrites in parallel.

This defines a functor $\mathcal{P} : \mathrm{HOAT}^{\mathrm{op}} \to \mathrm{CAT}$, the category of large categories. We define **Psh** to be the essential image.

Objects	$\mathcal{P}(\mathrm{T})$ for each $\mathrm{T} \in$ HOAT
Morphisms	$\mathcal{P}(F) = - \circ F^{\mathrm{op}} : \mathcal{P}(\mathrm{T}_2) \to \mathcal{P}(\mathrm{T}_1)$

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 $\mathrm{T}(-, \mathtt{S})$, which indexes all terms of sort $\mathtt{S}$. The basic logical objects are to be predicates on representables.

The **Yoneda embedding** $y : \mathrm{T} \to \mathcal{P}(\mathrm{T})$ is defined

$$y_{\mathrm{T}}(\mathtt{S}) = \mathrm{T}(-, \mathtt{S}) \qquad y_{\mathrm{T}}(\mathtt{f}) = - \circ \mathtt{f}.$$

Products and exponentials in $\mathcal{P}(\mathrm{T})$ ensure that y is cartex.

A subobject of a presheaf A is a natural indexed injection $\alpha_{\mathtt{S}} : \varphi(\mathtt{S}) \rightarrowtail A(\mathtt{S})$. Subobjects and commuting triangles form a category; isomorphism classes form the poset $\mathrm{Sub}(A)$.

A **subobject classifier** in a category with pullbacks C is an object $\Omega : \mathrm{C}$ equipped with a natural isomorphism

$$\mathrm{c} : \Omega^{(-)} \simeq \mathrm{Sub}(-).$$

A **predicate** in C is a morphism in C whose codomain is Ω. In the type system, Ω is to play the role of the type Prop. The **comprehension** of a predicate $\varphi : A \to \Omega$ is the subobject

$$\mathrm{c}(\varphi) \;:=\; \{a : A \mid \varphi(a)\} \rightarrowtail A.$$

A **topos** is a cartesian closed category which has finite limits and a subobject classifier. The presheaf category $\mathcal{P}(\mathrm{T})$ is a topos: exponents and subobject classifier are defined

$$\begin{array}{rcl} Q^P(\mathtt{S}) & = & \mathcal{P}(\mathrm{T})(y_{\mathrm{T}}(\mathtt{S}) \times P, Q) \\ \Omega(\mathtt{S}) & = & \{\varphi \rightarrowtail y_{\mathrm{T}}(\mathtt{S})\}. \end{array}$$

In fact $\mathcal{P}(\mathrm{T})$ has all limits and colimits, evaluated pointwise.

$$\begin{array}{rcl} (\lim_i A_i)(\mathtt{S}) & = & \lim_i \; A_i(\mathtt{S}) \\ (\mathrm{colim}_i A_i)(\mathtt{S}) & = & \mathrm{colim}_i \; A_i(\mathtt{S}) \end{array}$$

The values of Ω can be understood more concretely as

$$\Omega(\mathtt{S}) \simeq \{\text{sieves of sort } \mathtt{S}\}.$$

An **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** generated by a term $\mathtt{f} : \mathtt{S} \to \mathtt{T}$.

$$[\mathtt{f}] : \Omega(\mathtt{T}) \qquad [\mathtt{f}](\mathtt{R}) := \textstyle\sum_{u:\mathtt{R}\to\mathtt{S}} \mathtt{f} \circ u.$$

Example 16. The ρ-calculus is a concurrent language which can express recursive processes without a replication operator, as in the π-calculus. On a given name $n : 1 \to \mathtt{N}$, we may define a context which replicates processes as follows.

$$\begin{array}{ll} c(n) & := \mathtt{in}(n, \lambda x.\{\mathtt{out}(n, *x) \mid *x\}) \\ !(-)_n & := \mathtt{out}(n, \{c(n)|-\}) \mid c(n). \end{array}$$

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

For simpler formulae, we introduce some notation. 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 a predicate $\varphi : \Omega^A$, we denote $\varphi^a_{\mathtt{S}} := \varphi(\mathtt{S})(a)$. More generally for any indexed presheaf $p : P \to A$, we denote $p^a_{\mathtt{S}} := p_{\mathtt{S}}^{-1}(a)$ as the *fiber* over a (III-B).

A. The predicate fibration

There is a functor into $\mathcal{P}(\mathrm{T})$ called the predicate fibration, where the fiber over each presheaf is the complete Heyting algebra of its predicates. We show that quantification gives change-of-base adjoints between fibers; and moreover the domain is cartesian closed, complete and cocomplete. The fibration encapsulates the higher-order predicate logic of the topos.

Definition 17. The **predicate functor** of $\mathcal{P}(\mathrm{T})$ is defined

$$\Omega^{(-)} : \mathcal{P}(\mathrm{T})^{\mathrm{op}} \to \mathrm{CHA}.$$

For every $f : A \to B$, precomposition $\Omega^f : \Omega^B \to \Omega^A$ corresponds to preimage $\mathrm{Sub}(f) : \mathrm{Sub}(B) \to \mathrm{Sub}(A)$. For any $\psi : \Omega^B$, this can be expressed as *substitution*

$$\Omega^f(\psi)^a_{\mathtt{S}} = \psi_{\mathtt{S}}^{f_{\mathtt{S}}(a)}.$$

The fact that Ω^f is a morphism in CHA is known as *Frobenius reciprocity* [21].

The complete Heyting algebra structure of Ω^A is given: predicates are partially ordered by implication, meet and join are defined by pointwise intersection and union, $\top = A$ and $\bot = \emptyset$, 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 $\neg(\varphi) := (\varphi \Rightarrow \bot)$.

Hence, $\Omega^{(-)}$ is an indexed category, and we can assemble its image into one category by applying the Grothendieck construction (II).

Definition 18. The **category of predicates** of $\mathcal{P}(\mathrm{T})$ is denoted $\Omega\mathcal{P}(\mathrm{T})$, and is defined to be $\mathcal{P}(\mathrm{T}) \ltimes \Omega^{(-)}$.

Objects	$\langle A \;,\; \varphi : \Omega^A \rangle$
Morphisms	$f : \langle A, \varphi \rangle \to \langle B, \psi \rangle$
	$\langle f : A \to B \;,\; \varphi \Rightarrow \Omega^f(\psi) \rangle$

It is equipped with a projection called the **predicate fibration**

$$\pi_\Omega : \Omega\mathcal{P}(\mathrm{T}) \to \mathcal{P}(\mathrm{T}),$$

so that 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**.

The authors of [16] give a syntax for functors to be understood as *type refinement systems*. Let $p : \mathrm{D} \to \mathrm{T}$ be a functor. We denote $p(\psi) = B$ by $\psi \sqsubset B$, read as "ψ refines B", and we denote $p(\varphi \xrightarrow{\alpha} \psi) = (A \xrightarrow{f} B)$ by $\varphi \xrightarrow[f]{\alpha} \psi$; this can be understood as precondition-command-postcondition (I-B).

Definition 19. A **fibration** is a functor $p : \mathrm{D} \to \mathrm{T}$ with inference rules and axioms

$$\frac{f : A \to B \qquad \psi \sqsubset B}{\Omega^f(\psi) \xrightarrow[f]{} \psi}\ \mathrm{L} \qquad \frac{\varphi \xrightarrow[f\circ g]{} \psi}{\varphi \xrightarrow[g]{} \Omega^f(\psi)}\ \mathrm{R}$$

$$\text{for all} \quad \varphi \xrightarrow[f\circ g]{\beta} \psi, \quad \mathrm{L}(f) \circ \mathrm{R}(\beta) = \beta$$
$$\text{for all} \quad \varphi \xrightarrow[g]{\alpha} \Omega^f(\psi), \ \mathrm{R}(\mathrm{L}(f) \circ \alpha) = \alpha.$$

The projection $\pi_\Omega : \Omega\mathcal{P}(\mathrm{T}) \to \mathcal{P}(\mathrm{T})$ is a fibration: for each $f : A \to B$ and $\psi \sqsubset B$ there is $\mathrm{L}(f) : \Omega^f(\psi) \to \psi$ which maps the preimage into its image; this morphism is **cartesian**, meaning the equations above ensure that derivations of type ψ over $f \circ g$ factor uniquely through $\mathrm{L}(f)$.

In addition, the change-of-base functors have adjoints which generalize existential and universal quantification.

Proposition 20. $\pi_\Omega : \Omega\mathcal{P}(\mathrm{T}) \to \mathcal{P}(\mathrm{T})$ is a **$*$-bifibration** [39]: for each $f : A \to B$, the functor $\Omega^f : \Omega^B \to \Omega^A$ has left and right adjoints $\Sigma_f \dashv \Omega^f \dashv \Pi_f$, defined as follows.

$$\Sigma_f(\varphi)^b_{\mathtt{S}} := \textstyle\sum_{a:A_{\mathtt{S}}} \sum_{f_{\mathtt{R}}(a)=b} \varphi^a_{\mathtt{R}} \tag{1}$$
$$\Pi_f(\varphi)^b_{\mathtt{S}} := \textstyle\prod_{u:\mathtt{R}\to\mathtt{S}} \prod_{f_{\mathtt{R}}(a)=b\cdot u} \varphi^a_{\mathtt{R}} \tag{2}$$

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

These adjoints are intuited in the case of Set. For $\varphi : 2^A$ and $f : A \to B$, the subset $\mathrm{c}(\Sigma_f(\varphi))$ is the image of $\mathrm{c}(\varphi)$, and $\mathrm{c}(\Pi_f(\varphi))$ is the subset of elements of B with preimage contained in φ. Existential and universal quantification are special cases of these adjoints, by taking f to be a projection.

Example 21. Let T be a theory with rewrites (II-C), and suppose $\varphi : \Omega^{\mathtt{V}}$ is a predicate on terms. Then $\mathrm{c}(\Omega^s(\varphi))$ are the rewrites with source in φ; and $\mathrm{c}(\Sigma_t(\Omega^s(\varphi)))$ are the targets of these rewrites.

Hence there are *step-forward* and *step-back* operators

$$F_! := \Sigma_t\Omega^s : \Omega^{\mathtt{V}} \to \Omega^{\mathtt{V}} \qquad B_! := \Sigma_s\Omega^t : \Omega^{\mathtt{V}} \to \Omega^{\mathtt{V}}.$$

The *secure step-forward* performs a more refined operation: $F_*(\varphi) := \Pi_t(\Omega^s(\varphi))$ are terms u for which $(t \rightsquigarrow u) \Rightarrow \varphi(t)$. This should be useful in security protocols, to filter agents by past behavior.

Proposition 22. The change-of-base adjoints satisfy the **Beck–Chevalley condition**: if $\langle a, b\rangle : D \to A \times B$ is a pullback of $A \xrightarrow{f} C \xleftarrow{g} B$, then there are natural isomorphisms:

$$\Omega^g\Sigma_f(\varphi) \simeq \Sigma_b\Omega^a(\varphi)$$
$$\Omega^g\Pi_f(\varphi) \simeq \Pi_b\Omega^a(\varphi).$$

In type theory, the Beck–Chevalley condition means that quantification commutes with substitution. It implies that $\Omega^{(-)} : \mathcal{P}(\mathrm{T}) \to \mathrm{CHA}$ is a *first-order hyperdoctrine* [40].

Proposition 23. The projection $\pi_\Omega : \Omega\mathcal{P}(\mathrm{T}) \to \mathcal{P}(\mathrm{T})$ is a **higher-order fibration** [1, section 5.3]: a preordered $*$-bifibration satisfying the Beck–Chevalley condition, with cartesian closed fibers and a *generic object* $\Omega : \mathcal{P}(\mathrm{T})$.

This concept leaves implicit additional important structure: in predicate logic, there is implication between predicates on the same set; but more generally, the category of predicates is cartesian closed. Given $\varphi : 2^A$ and $\psi : 2^B$, we can define

$$[\varphi, \psi](f) := \forall a \in A.\ \varphi(a) \Rightarrow \psi(f(a)).$$

Proposition 24. Let $\varphi : \Omega^A$, $\psi : \Omega^B$ in $\mathcal{P}(\mathrm{T})$, and let

$$\langle \pi_1, \pi_2, ev\rangle : A \times [A, B] \to A \times [A, B] \times B.$$

Define the **hom predicate** $[\varphi, \psi] : \Omega^{[A,B]}$ to be

$$[\varphi, \psi] := \Pi_{\pi_2}(\Omega^{\pi_1}(\varphi) \Rightarrow \Omega^{ev}(\psi)).$$

With this structure, $\Omega\mathcal{P}(\mathrm{T})$ is cartesian closed, as is π_Ω.

The above gives maps $(\Omega^A)^{\mathrm{op}} \times \Omega^B \to \Omega^{[A,B]}$, which assemble into a global closed structure; but there is another way to form hom predicates which is much more fine-grained.

Proposition 25. Let $\mathcal{L}_{A,B} : [\Omega^A, \Omega^B] \to \Omega^{[A,B]}$ be the canonical morphism given by curried evaluation. There is a right adjoint which we call **reification**.

$$\mathcal{R}_{A,B} : [\Omega^A, \Omega^B] \to \Omega^{[A,B]}$$

$\mathcal{R}_{A,B}(F)$ are the maps which "respect the rule of F":

$$\mathcal{R}_{A,B}(F)^f_{\mathtt{S}} = \prod_{\varphi:\Omega^A} \Sigma_f(y(\mathtt{S}) \times \varphi) \Rightarrow F(\varphi). \tag{3}$$

The cartesian closed structure of $\Omega\mathcal{P}(\mathrm{T})$ is significant, because this means that $\pi_\Omega^{-1}(y_{\mathrm{T}}(\mathrm{T}))$ can be understood as a higher-order theory which refines the entire language T.

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 denote $\Sigma_{\mathtt{f}} : \Omega^{y_{\mathrm{T}}(\mathtt{S})} \to \Omega^{y_{\mathrm{T}}(\mathtt{T})}$ simply by $\mathtt{f}$, and $\Pi_{\mathtt{f}}$ by $\mathtt{f}_*$.

Example 26. Let T_ρ be the theory of the ρ-calculus (II-C2). The operation $\mathtt{in} : \mathtt{N} \times [\mathtt{N}, \mathtt{P}] \to \mathtt{P}$ constructs a process which inputs on a name $n : \mathtt{N}$ and continues as $\lambda x.p : [\mathtt{N}, \mathtt{P}]$.

Hence for $\alpha, \chi : \Omega(\mathtt{N})$ and $\varphi : \Omega(\mathtt{P})$ there is a *refined input*

$$\mathtt{in}_{\alpha,[\chi,\varphi]} : \alpha \times [\chi, \varphi] \to \mathtt{P}$$

which constructs a process that inputs on namespace α and upon receiving a name $n : \chi$ continues as $p[n/x] : \varphi$.

Moreover, this generalizes to "dependent processes": given a rule $F : \Omega^{\mathtt{N}} \to \Omega^{\mathtt{P}}$, then

$$\mathtt{in}_{\alpha,\mathcal{R}_{\mathtt{N},\mathtt{P}}(F)} : \alpha \times \mathcal{R}_{\mathtt{N},\mathtt{P}}(F) \to \mathtt{P}$$

constructs a process that inputs on namespace α and upon receiving $n : \chi$ continues as $p[n/x] : F(\chi)$.

As a more widely known example of contexts which ensure implications across substitution, we can construct the "magic wand" of separation logic [16]:

Example 27. Let T_h be the theory of a commutative monoid $(H, \cup, e)$, with a set of constants $\{h\} : 1 \to H$ adjoined as the elements of a heap. If we define

$$(\varphi{-\!\!*}\psi) := \Omega^{\lambda x.\cup}[\varphi, \psi]$$

then $h_1 : (\varphi{-\!\!*}\psi)$ asserts that $h_2 : \varphi \Rightarrow h_1 \cup h_2 : \psi$.

Using reification as described above, this can be generalized from pairs of predicates to functions of predicates. The authors are not aware if this has been studied.

In addition, the category of predicates has all limits and colimits, by a result of [41]. These can be used to form modalities, inductive and coinductive types, and more.

Proposition 28. $\Omega\mathcal{P}(\mathrm{T})$ is complete and cocomplete, and π_Ω preserves all limits and colimits. They are computed pointwise; letting π, ι represent the cone and cocone:

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

Example 29. In the ρ-calculus, we can construct a predicate for *safety and liveness* $\mathtt{sl}(\alpha)$, which asserts that a process that can always input on namespace α and cannot input on $\neg\alpha$.

There is a functor $SL_\alpha : \Omega^{y_{\mathrm{T}}(\mathtt{P})} \to \Omega^{y_{\mathrm{T}}(\mathtt{P})}$ defined

$$SL_\alpha(\varphi) = (\mathtt{in}(\alpha, [\mathtt{N}, \varphi]) \mid \mathtt{P}) \wedge \neg(\mathtt{in}(\neg\alpha, [\mathtt{N}, \mathtt{P}]) \mid \mathtt{P}).$$

The free monad SL^*_α is constructed as a colimit of iterated composites, and its values are greatest fixed points. Then

$$\mu\varphi.SL_\alpha(\varphi) = \mathsf{sl}(\alpha) := SL^*_\alpha([0]).$$

Definition 30. A **cartesian closed fibration** $F : \mathrm{D} \to \mathrm{T}$ is a fibration which is a strict cartesian closed functor, such that F is a morphism of the adjunction $- \times \varphi \dashv [\varphi, -]$ [42, Def. 3.12].

Similarly, the fibration is **complete and cocomplete** if it preserves limits and colimits [43].

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 [44].

Proposition 31. The predicate fibration $\pi_\Omega : \Omega\mathcal{P}(\mathrm{T}) \to \mathcal{P}(\mathrm{T})$ is a higher-order fibration which is **cosmic**: cartesian closed, complete and cocomplete.

B. The codomain fibration

Predicates $\varphi : \Omega^A$ correspond to subobjects $\mathrm{c}(\varphi) \rightarrowtail A$. While predicates suffice for certain modes of reasoning, it is natural to consider any $p : P \to A$ as a *dependent type*. This expands the fibers over A from truth values to sets, and the fibers over $\mathcal{P}(\mathrm{T})$ from posets to categories.

Definition 32. The **slice category** over A is denoted $\mathcal{P}(\mathrm{T})/A$. An object is a pair $\langle P, p : P \to A\rangle$, a morphism $h : p \to q$ is a morphism $h : P \to Q$ such that $q \circ h = p$.

The slice category $\mathcal{P}(\mathrm{T})/A$ subsumes the poset $\mathrm{Sub}(A)$, and it has all of the same rich structure.

Proposition 33. $\mathcal{P}(\mathrm{T})/A$ is equivalent to a presheaf topos.

$$\begin{array}{rcl} k : \mathcal{P}(\mathrm{T})/A & \leftrightharpoons & \mathcal{P}(\mathrm{el}(A)) : l \\ k(p)(\mathtt{S}, x) = p_{\mathtt{S}}^{-1}(x) & & l(F)^x_{\mathtt{S}} = F(\mathtt{S}, x) \end{array}$$

Here $\mathrm{el}(A)$ is the *category of elements*, i.e. the Grothendieck construction of $\iota \circ A : \mathrm{T}^{\mathrm{op}} \to \mathrm{Set} \to \mathrm{Cat}$. [21]

Hence $\mathcal{P}(\mathrm{T})/A$ is a complete and cocomplete topos. Given $p : P \to A$, $q : Q \to A$, the hom $[p, q] : [P, Q] \to A$ is given

$$[p, q]^z_{\mathtt{S}} = \prod_{u:\mathtt{R}\to\mathtt{S}} \prod_{a:A_{\mathtt{S}}} p_{\mathtt{R}}^{a\cdot u} \Rightarrow q_{\mathtt{R}}^{z_{\mathtt{R}}(u, a\cdot u)}.$$

The category has all limits and colimits, evaluated pointwise. Let CCT be the category of complete and cocomplete toposes.

Proposition 34. There is a functor $\Delta : \mathcal{P}(\mathrm{T})^{\mathrm{op}} \to \mathrm{CCT}$ that maps A to $\mathcal{P}(\mathrm{T})/A$ and $f : A \to B$ to pullback.

$$\Delta^f : \mathcal{P}(\mathrm{T})/B \to \mathcal{P}(\mathrm{T})/A \qquad \Delta_f(P)^a_{\mathtt{S}} = P_{\mathtt{S}}^{f_{\mathtt{S}}(a)}$$

Definition 35. The **category of dependent types** of $\mathcal{P}(\mathrm{T})$ is denoted $\Delta\mathcal{P}(\mathrm{T})$, and is defined to be $\mathcal{P}(\mathrm{T}) \ltimes \Delta$.

Objects	$\langle A\ ,\ p : P \to A\rangle$
Morphisms	$f : \langle A, p\rangle \to \langle B, q\rangle$
	$\langle f : A \to B\ ,\ \alpha : p \to \Delta^f(q)\rangle$
Composition	$\langle g, \beta\rangle \circ \langle f, \alpha\rangle = \langle g \circ f, \beta \circ \Delta^g(\alpha)\rangle$

This is equivalent to the arrow category of $\mathcal{P}(\mathrm{T})$.

The **codomain fibration** is the projection, which we denote $\pi_\Delta : \Delta\mathcal{P}(\mathrm{T}) \to \mathcal{P}(\mathrm{T})$. Change-of-base adjoints are given by the same formulae as predicates (1,2), and these satisfy the Beck–Chevalley condition.

Proposition 36. The codomain fibration π_Δ is a *closed comprehension category* [1, Sec 10.5] which is cosmic, i.e. cartesian closed, complete and cocomplete.

The two fibrations are connected by a reflection: comprehension interprets a predicate as a dependent type, and factorization takes a dependent type to its image predicate.

Definition 37. The **native structure** of a presheaf topos $\mathcal{P}(\mathrm{T})$, denoted $\Phi(\mathcal{P}(\mathrm{T})) = \langle \pi_{\Omega\mathrm{T}}, \pi_{\Delta\mathrm{T}}, \mathrm{i}_{\mathrm{T}}, \mathrm{c}_{\mathrm{T}}\rangle$, consists of the predicate and codomain fibrations of $\mathcal{P}(\mathrm{T})$, connected by the image-comprehension adjunction.

$$\begin{tikzcd} \Omega\mathcal{P}(\mathrm{T}) \arrow[rr, tail, "\mathrm{c}"', shift right] \arrow[dr, "\pi_\Omega"'] & & \Delta\mathcal{P}(\mathrm{T}) \arrow[ll, "\mathrm{i}"', shift right] \arrow[ll, phantom, "\perp"] \arrow[dl, "\pi_\Delta"] \\ & \mathcal{P}(\mathrm{T}) & \end{tikzcd}$$

This is a *full higher-order dependent type theory with comprehension* [1, Sec. 11.6]. This is a very rich structure which we cannot fully expound here.

There is a category of *Fho-DTT structures*, in which a morphism is a morphism of predicate fibrations and a morphism of type fibrations, such that the pair is a morphism of reflections.

If moreover these fibrations are cosmic, we demand that the morphisms preserve this structure. We denote by CHDT the category of cosmic FhoDTT-structures.

Theorem 38. The native structure of a presheaf topos defines a functor $\Phi : \mathrm{Psh} \to \mathrm{CHDT}$.

Proof. The functor forms the native structure

$$\Phi(\mathcal{P}(\mathrm{T})) := \langle \pi_{\Omega \mathrm{T}}, \pi_{\Delta \mathrm{T}}, \mathrm{i}_{\mathrm{T}}, \mathrm{c}_{\mathrm{T}} \rangle$$

and for $\mathcal{P}(F) : \mathcal{P}(\mathrm{T}_2) \to \mathcal{P}(\mathrm{T}_1)$, a morphism in CHDT

$$\Phi(F) : \langle \pi_{\Omega \mathrm{T}_2}, \pi_{\Delta \mathrm{T}_2}, \mathrm{i}_{\mathrm{T}_2}, \mathrm{c}_{\mathrm{T}_2} \rangle \to \langle \pi_{\Omega \mathrm{T}_1}, \pi_{\Delta \mathrm{T}_1}, \mathrm{i}_{\mathrm{T}_1}, \mathrm{c}_{\mathrm{T}_1} \rangle$$

which is simply precomposition by F.

$$\begin{array}{rcl} \Phi(F)_\Omega(\langle A, \varphi \rangle) & = & \langle A \circ F \ , \ \mathrm{c}_{\mathrm{T}_1}^{-1}(\mathrm{c}_{\mathrm{T}_2}(\varphi) \circ F) \rangle \\ \Phi(F)_\Delta(\langle A, p \rangle) & = & \langle A \circ F \ , \ p \circ F \rangle). \end{array}$$

These functors are continuous and cocontinuous because $\mathcal{P}(F)$ is both a left and right adjoint. They are morphisms of fibrations because the cartesian morphisms are pullback squares. Finally, they are morphisms of reflections because whiskering units and counits commutes with precomposition. □

In summary, we have constructed a functor

$$\mathrm{Pres}^{\mathrm{op}} \xrightarrow{\Lambda} \mathrm{HOAT}^{\mathrm{op}} \xrightarrow{\mathcal{P}} \mathrm{Psh} \xrightarrow{\Phi} \mathrm{CHDT}$$

which elucidates a wealth of structure in every language given by a higher-order algebraic theory.

We now present the native type system.

IV. Native Type Theory

We present the **native type system** $\Phi(\mathcal{P}(\mathrm{T}))$ of a higher-order algebraic theory T given by a presentation Σ (II).

The system is a *full higher-order dependent type theory* [1, Sec. 11.5] parameterized by the theory T. We do not present Equality and Quotient types. We encode Subtyping, Hom, and Inductive types, which we use in applications.

The system has two basic entities: **predicates** and **types**

$$A \vdash \varphi : \mathsf{Prop} \qquad A \vdash B : \mathsf{Type}$$

interpreted as $\varphi : A \to \Omega$ and $p : B \to A$, where Ω is the subobject classifier. The **higher-order axiom** Prop : Type enables quantification over predicates of a type.

Substitution is interpreted as pullback: $B[t] := \Delta^t(p)$. A term $\Gamma \vdash a : A$ is a morphism $a : X \to \Gamma$ together with a section $x : 1_X \to A[a]$ in $\mathcal{P}(\mathrm{T})/X$. For details see [45].

For usability, we present the type system as generated from the theory. This way, a programmer can start in the ordinary language, and use the ambient logical structure as needed.

Representables are given in the type system as axioms, because $y : \mathrm{T} \to \mathcal{P}(\mathrm{T})$ preserves products and exponents.

$$\frac{[\![\mathtt{S} : \mathrm{T}]\!]}{y(\mathtt{S}) : \mathsf{Type}}\ \mathrm{T}_S \qquad \frac{[\![\mathtt{S}_1 \vdash \mathtt{f} : \mathtt{S}_2]\!]}{y(\mathtt{S}_2) \vdash y(\mathtt{f}) : \mathsf{Type}}\ \mathrm{T}_O$$

$$\frac{[\![\mathtt{S}_1 \vdash \mathtt{f} = \mathtt{g} : \mathtt{S}_2]\!]}{y(\mathtt{S}_2) \vdash y(\mathtt{f}) = y(\mathtt{g})}\ \mathrm{T}_E$$

Subobject type of a predicate. Using comprehension, a predicate is converted to the type of its satisfying terms.

$$\frac{\Gamma, x{:}A \vdash \varphi : \mathsf{Prop}}{\Gamma \vdash \{x{:}A \mid \varphi\} : \mathsf{Type}}\ \mathrm{c_F}$$

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

$$\frac{\Gamma \vdash N : \{x{:}A \mid \varphi\}}{\Gamma \vdash \mathsf{o}(N) : A}\ \mathrm{c_E} \qquad \begin{array}{rcll} \mathsf{o}(\mathsf{i}(M)) & = & M & (\mathrm{c}_\beta) \\ \mathsf{i}(\mathsf{o}(N)) & = & N & (\mathrm{c}_\eta) \end{array}$$

$$\frac{\Gamma_1, x{:}A, \Gamma_2, \varphi \vdash \psi}{\overline{\Gamma_1, y : \{x{:}A \mid \varphi\}, \Gamma_2[\mathsf{o}(y)/x] \vdash \psi[\mathsf{o}(y)/x]}}\ \mathrm{c_E^\circ}$$

Image type rules, which convert a type to its image predicate, can be derived from Σ and Equality.

Because predicates are a reflective subcategory of types, it suffices to give the inference rules for types.

Dependent sum type is a form of indexed sum, which generalizes product types and existential quantification.

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x{:}A \vdash B : \mathsf{Type}}{\Gamma \vdash \Sigma x{:}A.B : \mathsf{Type}}\ \Sigma_{\mathrm{F}}$$

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

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

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

Dependent product type is a form of indexed product, which generalizes function types and universal quantification.

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

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

$$\frac{\Gamma \vdash f : \Pi x{:}A.B \qquad \Gamma \vdash u : B}{\Gamma \vdash f(u) : B[u/x]}\ \Pi_{\mathrm{E}}$$

$$\begin{array}{lcll} (\lambda x{:}A.t)(a) & = & t(a) & (\beta) \\ f & = & \lambda x{:}A.f & (\eta) \end{array}$$

We can 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 as special cases of Σ and Π.

Hom type is defined

$$\frac{A_1 \vdash B_1 : \mathsf{Type} \qquad A_2 \vdash B_2 : \mathsf{Type}}{[A_1, A_2] \vdash [B_1, B_2] := \Pi x{:}A_1.B_1[\pi] \Rightarrow B_2[ev]}$$

where $\langle \pi, ev \rangle : A_1 \times [A_1, A_2] \to A_1 \times A_2$. This encoding represents functions of "independent types".

Subtyping of predicates is defined

$$(\varphi \leq \psi) := \forall a{:}A.\ \varphi(a) \Rightarrow \psi(a).$$

Inductive type for any endofunctor $F : \Omega^A \to \Omega^A$, the greatest fixed point predicate is defined:

$$\mu\varphi.F(\varphi) \quad := \quad \forall\varphi{:}[A, \mathsf{Prop}].\ (F(\varphi) \leq \varphi) \Rightarrow \varphi$$

Coinductive type is defined dually.

These types and rules constitute the native type system $\Phi(\mathcal{P}(\mathrm{T}))$, abridged for a first presentation. By adding rules of functoriality, we can incorporate translations of languages.

$$\frac{[\![F : \mathrm{T}_1 \to \mathrm{T}_2]\!] \qquad \Gamma \vdash A : \mathsf{Type}_1}{\Gamma \circ F \vdash A \circ F : \mathsf{Type}_2}\ F_{\mathsf{Ty}}$$

$$\frac{[\![F : \mathrm{T}_1 \to \mathrm{T}_2]\!] \qquad \Gamma \vdash a : A}{\Gamma \circ F \vdash a(F) : A \circ F}\ F_{\mathsf{Tm}}$$

How do we use the system to express simple, useful ideas? Suppose we are working in a language T; we have constructed a program $\mathtt{f} : \mathtt{S} \to \mathtt{T}$, and we have a protocol $\mathtt{p} : \mathtt{T} \to \mathtt{U}$ for which we need terms to have been already processed by $\mathtt{f}$. We construct the type

$$\frac{y(\mathtt{T}) \vdash y(\mathtt{f}) : \mathsf{Type} \qquad y(\mathtt{T}), y(\mathtt{f}) \vdash y(\mathtt{S}) : \mathsf{Type}}{y(\mathtt{T}) \vdash \Sigma x{:}y(\mathtt{f}).y(\mathtt{S}) : \mathsf{Type}}$$

for which a term must be derived by the rule

$$\frac{y(\mathtt{T}) \vdash \mathtt{g} : y(\mathtt{f}) \qquad y(\mathtt{T}) \vdash u : y(\mathtt{S})[\mathtt{g}/x]}{y(\mathtt{T}) \vdash \langle \mathtt{g}, u\rangle : \Sigma x{:}y(\mathtt{f}).y(\mathtt{S}).}$$

Then the image (III-A) $\exists_{\mathtt{f}}(y(\mathtt{S})) : [\mathtt{T}, \mathsf{Prop}]$ accepts terms of the form $\mathtt{g} = \mathtt{f}(u)$ for some $u : \mathtt{R} \to \mathtt{S}$. We can then refine the protocol $y(\mathtt{p})_{\mathtt{f}} : \exists_{\mathtt{f}}(y(\mathtt{S})) \to y(\mathtt{T})$ to enforce the condition.

Of course, a first paper uses simple examples to explain. Exploring the full scope and utility of native type systems is really a community activity. For now we demonstrate some applications to indicate only a few possibilities.

V. Applications

A. Behavior and Modalities

Let T be a higher-order algebraic theory with rewriting. The graph of rewrites over terms is a dependent type. The fiber over each pair is the set of rewrites between terms.

$$\begin{array}{lcl} \mathrm{g} & := & y(\langle s,t\rangle) : y(\mathtt{E}) \to y(\mathtt{V}^2) \\ \mathrm{g}_{\mathtt{S}}^{(v_1,v_2)} & = & \{e \mid \mathtt{S} \vdash e : v_1 \rightsquigarrow v_2\} \end{array}$$

By incorporating dynamics in a theory with rewriting (II-C), a native type system can express predicates for future and past behaviors, or *temporal modalities*.

By using adjoints to preimage along $s, t : \mathtt{E} \to \mathtt{V}$, we defined step-forward and backward operators $F_!, B_!$ and "secure-step" operators F_*, B_*. For a predicate on terms $\varphi : \Omega(\mathtt{V})$, $B_!(\varphi)$ are terms which *possibly* rewrite to φ, and $B_*(\varphi)$ are terms which *necessarily* rewrite to φ.

By iterating, we can form each kind of modality.

$B_!^\circ(\varphi) =$	$\bigvee_{\mathbb{N}} B_!^n(\varphi)$	can become φ
$B_*^\circ(\varphi) =$	$\bigvee_{\mathbb{N}} B_*^n(\varphi)$	will become φ
$B_!^\bullet(\varphi) =$	$\bigwedge_{\mathbb{N}} B_!^n(\varphi)$	always can become φ
$B_*^\bullet(\varphi) =$	$\bigwedge_{\mathbb{N}} B_*^n(\varphi)$	always will become φ

(also for F, past behavior) For example, when φ is a capacity to receive and process input, $B_*^\bullet(\varphi)$ is "liveness". If it is a guarantee to only communicate on the proper channels, $B_*^\bullet(\varphi)$ is "safety".

B. Firewall

The namespace φ might be a collection of trusted addresses for an organization or it could be a datatype, such as the XML schema.

We can express these conditions as an inductive type.

$$\mathsf{sole.in}(\alpha) := \mu\varphi.\ (\mathtt{in}(\alpha, \mathtt{N}.\varphi) \mid \mathtt{P}) \wedge \neg(\mathtt{in}(\neg(\alpha), \mathtt{N}.\mathtt{P}) \mid \mathtt{P})$$

In effect, this is a *compile-time firewall*: a process satisfies this predicate if and only if it can only communicate within

The continuing process $\mathtt{q}$ has a free name – how do we know that it can't receive a name $\mathtt{b} : \neg\alpha$, and then input on $\mathtt{b}$? While negation (§III) is boolean for closed terms, it is strictly intuitionistic for general contexts: the algorithm above, formalized in the presheaf topos, will "detect" if there exists a substitution which allows a process to input on $\neg\alpha$. The correctness of the type theory is that of the internal logic.

By constructing a process which uses this type, such as $\mathtt{in}(\mathtt{n}, \mathtt{x}.\mathtt{p}) : \mathtt{obin}(\mathtt{N}, \chi.\mathsf{sole.in}(\chi))$, the communication of the the continuing process $\mathtt{p}$ is controlled and understood.

C. Internal Hom and Semantic Search

Christian: compelling example for the hom.

Hoogle [9] is a search engine for the Haskell libraries on Stackage where one provides a type as a search query. A code search engine using native types could search by pre- or post-condition, by computational complexity, or even search by security vulnerability.

The RV-Match software, built on K Framework's operational semantics for C, essentially tries to unify programs with the type of "underspecified C program" [?]. A successful unification indicates a bug in the software like a buffer overflow or an out-of-lifetime access.

D. Sublanguages

Languages have many useful "sublanguages" consisting of terms that satisfy certain properties. For example, the linear λ-calculus consists of terms which use each variable exactly once, such as the pairing combinator $\lambda xyz.zxy$.

In any language T, we can specify the "linear sublanguage" $\mathsf{linear}(\mathrm{T})$ by restricting to terms which do not copy or delete, simply by excluding $\Delta_{\mathtt{S}} : \mathtt{S} \to \mathtt{S}^2$ and $!_{\mathtt{S}} : \mathtt{S} \to 1$.

$$\mathsf{linear}(\mathrm{T})(-, \mathtt{S}) := \bigwedge_{f:\mathtt{S}^2 \to \mathtt{S}} \neg[f(\Delta_{\mathtt{S}})] \wedge \bigwedge_{f:1 \to \mathtt{S}} \neg[f(!_{\mathtt{S}})]$$

In general, typing rules which restrict terms by certain "well-formed" conditions can define a predicate that determines the sublanguage of such terms.

For examples, the simply-typed λ-calculus consists of λ terms which terminate; Abramsky's "proofs as processes" does the same for a π-calculus [?]; and Baillot and Terui describe a λ-calculus whose terms halt in polynomial time [46].

Hence, native types can be used to reason about sublanguages which address practical issues such as resource usage and complexity.

E. Translations and Composite Systems

Christian: need to demonstrate functoriality of the construction. Might combine with Composite Systems, but not sure.

Even if a programming language has a strong type system, the type system is usually restricted to reasoning about the language in isolation. However, the reality of modern computing is that programs usually execute in an embedded, networked environment.

ECMAScript is a single-threaded language that executes purely synchronously, but often runs in an asynchronous environment. In a web browser, programmers have to be aware of HTML and CSS rendering, DOM events, network events, deferred scripts, async scripts, timers, and promises, and know the order in which each of these occur.

Given an operational semantics for a web browser that includes the operational semantics of ECMAScript, one can generate a type system that includes the semantics of the browser. Programmers can then express their intended order of execution in the type system, and the type checker can verify that the code satisfies the type.

VI. Conclusion

The conclusion goes here.

Acknowledgments

The authors would like to thank.

VII. Appendix

References

[1] B. Jacobs, *Categorical Logic and Type Theory*. Amsterdam: Elsevier, 1998.

[2] F. W. Lawvere, "Functorial semantics of algebraic theories," *Proceedings of the National Academy of Sciences*, vol. 50, no. 5, pp. 869–872, 1963. [Online]. Available: http://tac.mta.ca/tac/reprints/articles/5/tr5abs.html

[3] T. Leinster, *Basic Category Theory*. Cambridge University Press, 2009. [Online]. Available: https://doi.org/10.1017%2Fcbo9781107360068

[4] "Microsoft typescript." [Online]. Available: https://www.typescriptlang.org/

[5] "Flow: A static type checker for javascript." [Online]. Available: https://flow.org/

[6] "Google closure compiler." [Online]. Available: https://developers.google.com/closure/compiler

[7] "Typescript: A note on soundness." [Online]. Available: https://www.typescriptlang.org/docs/handbook/type-compatibility.html#a-note-on-soundness

[8] "K framework." [Online]. Available: http://www.kframework.org/index.php/Main_Page

[9] "Hoogle." [Online]. Available: https://hoogle.haskell.org/

[10] M. Hennessy and R. Milner, "On observing nondeterminism and concurrency," in *Automata, Languages and Programming*, J. de Bakker and J. van Leeuwen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1980, pp. 299–309.

[11] L. Caires, "Behavioral and spatial observations in a logic for the π-calculus," in *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004, pp. 72–89. [Online]. Available: https://doi.org/10.1007%2F978-3-540-24727-2_7

[12] "A spatial logic model checker." [Online]. Available: http://ctp.di.fct.unl.pt/SLMC/

[13] L. G. Meredith and M. Radestock, "Namespace logic: A logic for a reflective higher-order calculus," in *Trustworthy Global Computing*. Springer Berlin Heidelberg, 2005, pp. 353–369. [Online]. Available: https://doi.org/10.1007%2F11580850_19

[14] L. Meredith and M. Radestock, "A reflective higher-order calculus," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 5, pp. 49–67, dec 2005. [Online]. Available: https://doi.org/10.1016%2Fj.entcs.2005.05.016

[15] X. Chen and G. Rosu, "Matching µ-logic," in *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, jun 2019. [Online]. Available: https://doi.org/10.1109%2Flics.2019.8785675

[16] P.-A. Melliès and N. Zeilberger, "Functors are type refinement systems," *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 3–16, may 2015. [Online]. Available: https://doi.org/10.1145%2F2775051.2676970

[17] C. A. R. Hoare, "An axiomatic basis for computer programming," in *Programming Methodology*. Springer New York, 1978, pp. 89–100. [Online]. Available: https://doi.org/10.1007%2F978-1-4612-6315-9_9

[18] "Compile-time primes." [Online]. Available: http://stoppels.blog/posts/compile-time-primes

[19] "Type guards and differentiating types." [Online]. Available: https://www.typescriptlang.org/docs/handbook/advanced-types.html#type-guards-and-differentiating-types

[20] J. M. E. Hyland and A. M. Pitts, "The theory of constructions: categorical semantics and topos-theoretic models," pp. 137–199, 1989. [Online]. Available: https://doi.org/10.1090%2Fconm%2F092%2F1003199

[21] S. M. Lane and I. Moerdijk, *Sheaves in Geometry and Logic*. Springer New York, 1994. [Online]. Available: https://doi.org/10.1007%2F978-1-4612-0927-0

[22] "Higher-order algebraic theories." [Online]. Available: https://www.cl.cam.ac.uk/~na412/Higher-order%20algebraic%20theories.pdf

[23] T. Coquand and G. Huet, "The calculus of constructions," *Information and Computation*, vol. 76, no. 2-3, pp. 95–120, feb 1988. [Online]. Available: https://doi.org/10.1016%2F0890-5401%2888%2990005-3

[24] J. Adamek, J. Rosicky, E. M. Vitale, and F. W. Lawvere, *Algebraic Theories*. Cambridge University Press, 1994. [Online]. Available: https://doi.org/10.1017%2Fcbo9780511760754.005

[25] M. Fiore, G. Plotkin, and D. Turi, "Abstract syntax and variable binding," in *Proceedings, 14th Symposium on Logic in Computer Science*, 1999, pp. 193–202.

[26] M. Fiore and O. Mahmoud, "Second-order algebraic theories," in *Mathematical Foundations of Computer Science 2010*. Springer Berlin Heidelberg, 2010, pp. 368–380. [Online]. Available: https://doi.org/10.1007%2F978-3-642-15155-2_33

[27] M. ESCARDÓ and P. OLIVA, "Selection functions, bar recursion and backward induction," *Mathematical Structures in Computer Science*, vol. 20, no. 2, pp. 127–168, mar 2010. [Online]. Available: https://doi.org/10.1017%2Fs0960129509990351

[28] P. Wadler, "Comprehending monads," in *Proceedings of the 1990 ACM conference on LISP and functional programming - LFP '90*. ACM Press, 1990. [Online]. Available: https://doi.org/10.1145%2F91556.91592

[29] "Haskell: monad do-notation." [Online]. Available: https://wiki.haskell.org/Monad#do-notation

[30] "Scala for-comprehensions and for-loops." [Online]. Available: https://scala-lang.org/files/archive/spec/2.11/06-expressions.html#for-comprehensions-and-for-loops

[31] "Data structures — python 3.9.1 documentation." [Online]. Available: https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

[32] "Overloading python list comprehension." [Online]. Available: http://blog.sigfpe.com/2012/03/overloading-python-list-comprehension.html

[33] P. Selinger, "Lecture notes on the lambda calculus," 2013.

[34] S. Schanuel and R. Street, "The free adjunction," *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, vol. 27, no. 1, pp. 81–83, 1986. [Online]. Available: www.numdam.org/item/CTGDC_1986__27_1_81_0/

[35] J. Moeller and C. Vasilakopoulou, "Monoidal grothendieck construction," 2020.

[36] R. Street, "Categorical and combinatorial aspects of descent theory: Categorical descent theory (guest editors: George janelidze, bodo pareigis and walter tholen)," *Applied Categorical Structures*, vol. 12, 10 2004.

[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. [Online]. Available: https://doi.org/10.1109%2Flics.1997.614955

[38] "Rchain." [Online]. Available: https://www.rchain.coop/

[39] M. A. Shulman, "Framed bicategories and monoidal fibrations," 2009.

[40] F. W. Lawvere, "Adjointness in foundations," *dialectica*, vol. 23, no. 3-4, pp. 281–296, dec 1969. [Online]. Available: https://doi.org/10.1111%2Fj.1746-8361.1969.tb01194.x

[41] A. Tarlecki, R. M. Burstall, and J. A. Goguen, "Some fundamental algebraic tools for the semantics of computation: Part 3. indexed categories," *Theoretical Computer Science*, vol. 91, no. 2, pp. 239 – 264, 1991. [Online]. Available: http://www.sciencedirect.com/science/article/pii/030439759190085G

[42] C. Vasilakopoulou, "On enriched fibrations," 2018.

[43] ——, "Generalization of algebraic operations via enrichment," 2014.

[44] R. Street, "Elementary cosmoi i," in *Category Seminar*, G. M. Kelly, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1974, pp. 134–180.

[45] R. A. G. Seely, "Locally cartesian closed categories and type theory," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 95, no. 1, pp. 33–48, jan 1984. [Online]. Available: https://doi.org/10.1017%2Fs0305004100061284

[46] P. Baillot and K. Terui, "Light types for polynomial time computation in lambda-calculus," in *In Proc. of LICS 2004*. IEEE Computer Society, 2004, pp. 266–275.