Structural types for algebraic theories

Christian Williams

May 10, 2020

Abstract

Structural types are made from the same operations as terms, plus those of predicate logic; they provide languages with a notion of type which is intrinsic. We give an algorithm which takes a second-order algebraic theory and produces a structural type theory with polymorphism, subtyping, and recursion.

A theory embeds into a topos of presheaves, which admits a subobject functor to posets. The image consists of sieves in the theory, understood as types, and constructors thereof. The functor is lax cartesian, and moreover colax closed: by these we can lift operations to the level of types.

The algorithm defines a 2-functor from second-order algebraic theories to colax models in Pos. Such a model gives a full image in Pos in which to derive the constructors of the theory, predicate logic, and limits and colimits; these data give a structural type theory. We demonstrate with the language of the ρ-calculus, a concurrent language with reflection, the logic for which gave original motivation.

1 Introduction

1.1 Motivation

RChain [24] is a distributed computing system based on the ρ-calculus, or **r**eflective **h**igher-**o**rder π-calculus.

The π-calculus [20] is a concurrent language: a program is not a sequence of instructions; it is a multiset of parallel processes, which evolves by *communication* over names. Computation is interactive, providing a unified language for both computers and networks. The basic rule is:

$$\mathtt{COMM}_\pi : \ \bar{\mathtt{n}}\mathtt{a} \mid \mathtt{n(x).p} \ \Rightarrow \ \mathtt{p[a/x]}$$

{[output on name n the name a] in parallel with [input on n, then p(x)]} evolves to {p(a)}.

The ρ-calculus [22] adds *reflection*: operators which turn processes (code) into names (data) and vice versa. Names are not atomic symbols; they are references to processes. Code can be sent as a message, then evaluated in another context – this "code mobility" allows for deep self-reference.

$$\mathtt{COMM}_\rho : \ \mathtt{out(n,q)} \mid \mathtt{in(n,x.p)} \Rightarrow \mathtt{p[@q/x]}$$

{[output on name n the process q] in parallel with [input on n, then p(x)]} evolves to {p(@q)}.

Names are both the data being communicated and the communication channels. Through reflection, names have form – this provides intrinsic structure to networks. We aim to use this structure, to think globally about the web. Namespace Logic [21] is a framework for this reasoning. The theory is:

$$\mathrm{NL} = \rho \text{ calculus} + \text{predicate logic} + \text{recursion}.$$

The signature of the language is augmented with that of predicate logic: a formula φ of NL is a predicate on the structure of terms in the ρ-calculus; this is used as a type, to condition programs. For example, in concurrent computation it is useful to determine whether a process is single-threaded:

$$\mathsf{single.thread} := \neg[\bar{0}] \wedge \neg[\neg[\bar{0}] \ \dot{|} \ \neg[\bar{0}]]$$

"not the null process, and not the parallel of two non-null processes".

The types are *structural* in that they are built from the same operations as terms. This provides an intrinsic type system for languages, which is especially useful for concurrent languages with reflection. The goal is to formalize the algorithm which equips a theory with structural types, predicate logic, and recursion.

1.2 Contribution

We give an algorithm which takes a second-order algebraic theory and produces a polymorphic type theory with subtyping and recursion [Theorem 11, Theorem 13, Definition 18].

A theory embeds by y into a topos of presheaves; this admits a subobject functor $\mathcal{P}$ to posets.

$$\mathcal{P} \circ y : \mathrm{T} \longrightarrow [\mathrm{T}^{\mathrm{op}}, \mathrm{Set}] \longrightarrow \mathrm{Pos}$$

Sieves, or right ideals, in a theory T provide a notion of predicate, or type. The Heyting algebra structure of subfunctors and the adjoints to substitution provide the constructors of intuitionistic predicate logic (§3).

Because $\mathcal{P}y$ is lax cartesian and colax closed, operations of T can be lifted to sieves and understood as type constructors (§4). The compatibility of operations and their liftings gives that the former are *polymorphic* with respect to the latter.

These posets are complete and cocomplete, providing fixpoint operators and hence *recursion* (§5.1). Putting the data together, we give the *structural type theory* of T (§6)

We denote the resulting theory by $\omega\mathrm{T}$, which has the general structure:

kinds	sorts	$\mathtt{s} \in \mathrm{T}$	**constructors**	lifted ops	$\bar{\mathtt{f}} : \mathcal{P}y(\mathtt{s}) \to \mathcal{P}y(\mathtt{t})$
types	sieves	$\varphi \rightarrowtail \mathrm{T}(-, \mathtt{s})$	$\cdot$	pred. logic	$\vee, \wedge, \Rightarrow, \exists, \forall$
terms	operations	$\mathtt{f} : \mathtt{s} \to \mathtt{t}$	$\cdot$	fixpoints	$\nu, \mu : \mathcal{P}y(\mathtt{s})^{\mathcal{P}y(\mathtt{s})} \to \mathcal{P}y(\mathtt{s})$.

We demonstrate the type theory with predicates useful to distributed computing (§5). This is a firewall:

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

The basic idea is that of generating a *logic over a type theory* [12]; we extend the case for second-order algebraic theories by the lifting of operations. The result is an algorithm that produces expressive type theories which enable high-level reasoning about the structure of terms in languages.

The present work is connected to several existing ideas. Matching μ-Logic [25] is a logic based on pattern-matching, which is used for program specification and verification in the K Framework [13]; we generalize this logic to include signatures with binding operations.

Parameterized algebraic theories [26] consider a theory "parameterized" by a Lawvere theory using presheaves; this is closely connected to the present work. There is much to be done in connecting the work to categorical logic, type theory, and topos theory; this is an application and an introductory thesis.

1.3 Notation

A cartesian category is a category with finite products; a cartesian functor is one which preserves them. For algebraic theories, an n-ary operation is equivalent to a term with n free variables.

We denote the category of sets by Set, a skeleton of the category of finite sets by $\mathbb{F}$, the category of partial orders by Pos, the category of Heyting algebras by Hey, and the 2-category of categories by Cat. The 2-category of 2-categories, colax 2-functors, and colax natural transformations is denoted by $2\mathrm{Cat}_{colax}$.

We denote the (small) Yoneda embedding by $\widehat{-} : \mathrm{Cat} \to \mathrm{Cat}$, and sometimes $y(-)$ to avoid large hats; we use the same notation on the level of objects, writing $\hat{\mathtt{t}} = y(\mathtt{t}) = \mathrm{T}(-, \mathtt{t})$. We take as given that y is cartesian closed, and omit the isomorphisms involved. We also omit considerations of size.

A presheaf on C is a functor $P : C^{\mathrm{op}} \to \mathrm{Set}$; we denote the action $P(f) : P(b) \to P(a)$ by $- \cdot f$. The sign $\int$ denotes a coend if there is a quantified variable on top; otherwise it denotes the Grothendieck construction.

An exponential object $\mathtt{t}^{\mathtt{s}}$ may alternately be written $[\mathtt{s}, \mathtt{t}]$ or $\mathtt{s.t}$, as in the case of binding operations. We denote currying $\mathtt{f} : \mathtt{a} \times \mathtt{b} \to \mathtt{c}$ by $\mathtt{f}^{\bullet} : \mathtt{a} \to [\mathtt{b}, \mathtt{c}]$, and uncurrying $u : \mathtt{a} \to [\mathtt{b}, \mathtt{c}]$ by $u^{\circ} : \mathtt{a} \times \mathtt{b} \to \mathtt{c}$.

1.4 Acknowledgements

Thank you to John Baez, Mike Shulman, and Nathanael Arkor for helpful conversations.

Thank you to Greg Meredith and Mike Stay for all of the discussion and support, and for the opportunity to build on your great vision. This work would not exist if we had not met and begun this inspiring dialogue. I look forward to seeing the dream of RChain realized.

2 Algebraic theories

We can design and use languages, efficiently and effectively, by the method of *presentation*. Algebraic structures are presented by a set of sorts $\mathtt{s} \in \mathcal{T}$, operations $\mathtt{f} \in \mathcal{O}$, and equations $\mathtt{f}_1 = \mathtt{f}_2 \in \mathcal{E}$. For example, a monoid is a structure of the form:

$$\begin{aligned}
&\mathcal{T} = \{\mathtt{M}\} \\
&\mathcal{O} = \{\mathtt{m} : \mathtt{M}^2 \to \mathtt{M} \ \ , \mathtt{e} : 1 \to \mathtt{M}\} \\
&\mathcal{E} = \{\mathtt{m}(\mathtt{m}(x,y),z) = \mathtt{m}(x,\mathtt{m}(y,z)) \, [\mathtt{assoc}], \\
&\qquad \mathtt{m}(e,x) = x \ , \mathtt{m}(x,e) = x \qquad [\mathtt{lunit}] \ , [\mathtt{runit}]\}.
\end{aligned}$$

As in the case of presenting a particular monoid by generators and relations, such a presentation specifies the "theory" of such a structure: the *free cartesian category* on a presentation $(\mathcal{T}, \mathcal{O}, \mathcal{E})$.

Definition 1. An **algebraic theory** is a cartesian category T equipped with a presentation $(\mathcal{T}, \mathcal{O}, \mathcal{E})$. The 2-category of algebraic theories, cartesian functors, and natural transformations is denoted Thy. The category of **models** of T in **context** C is the functor category $\mathrm{Thy}(\mathrm{T}, \mathrm{C})$.

Theories which are single-sorted, T for which $\mathcal{T}$ is a singleton, are known as *Lawvere theories.* Lawvere initiated the categorical study of universal algebra by defining "finite product theories" in the 1963 thesis [14]. Soon after, Linton proved that Lawvere theories are equivalent to *finitary monads* on Set [17].

Given a Lawvere theory T, the *free model* $f_{\mathrm{T}} : \mathrm{Set} \to \mathrm{Thy}(\mathrm{T}, \mathrm{Set})$ is given by

$$f_{\mathrm{T}}(X) \simeq \int^{n \in \mathbb{F}} X^n \times \mathrm{T}(\mathtt{s}^n, -).$$

This f_{T} is left adjoint to the forgetful functor $u_{\mathrm{T}} : \mathrm{Thy}(\mathrm{T}, \mathrm{Set}) \to \mathrm{Set}$ which evaluates a model at $\mathtt{s}$. The induced monad $T : \mathrm{Set} \to \mathrm{Set}$ takes X as a set of generators, and makes the set of formal terms $\mathtt{f}(x_1, \ldots, x_n)$, subject to the equations of the theory. This gives a monadic adjunction.

Functional programming uses monads to encapsulate data structures and model computational effects [23]. By the equivalence above, these can be reformulated in terms of algebraic theories for more explicit presentation, as well as more direct connection to object-oriented programming [11].

A reference for algebraic theories is [1].

2.1 Theories with binding

Languages such as the λ-calculus or ρ-calculus cannot be described as algebraic theories. While cartesian categories are useful to both mathematics and computer science, there is an important syntactic construction which they cannot express:

how do we take an expression $e(x)$ and form the *map* or the *program* $x \mapsto e(x)$?

The key is to make explicit that every term has a *context* of free variables. This is necessary because this process of forming maps changes the context: we take a distinguished free variable and "bind" it in the term. The canonical example of variable binding is λ abstraction.

$$\frac{\Gamma, x \vdash \ x + 1}{\Gamma \vdash \ \lambda x.(x+1)} \ \lambda$$

Before the inference rule, the variable x is free in the term $x + 1$; after the rule it is bound – the bound variable is distinguished in such a way that we can define substitution:

$$\lambda x.(x+1) \bullet 2 \ \Rightarrow \ (x+1)[\mathbf{2}/\mathbf{x}] = 2 + 1 = 3.$$

The syntax of binding is well-known to type theorists, and its semantics is understood to be given by cartesian closed categories [16]. While the idea is expressive, this notion of "higher-order theory" does not share the direct connection of algebraic theories to monads and universal algebra ([2] 6.2).

Fiore, Turi, and Plotkin expanded the notion of algebraic theory to include binding [5]; the work has been expanded to demonstrate monadicity [3], and elaborate second-order logic [4]. The basic idea is to understand an operad as a set of terms indexed by context.

$$\overline{\mathrm{T}} : \mathbb{F} \to \mathrm{Set} \qquad \overline{\mathrm{T}}(n) = \{t \mid n \vdash t\}$$

A functor $\mathbb{F} \to \mathrm{Set}$ is a *cartesian operad* if it is monoidal with respect to substitution tensor product $(\mathrm{Set}^{\mathbb{F}}, \bullet, \mathrm{V})$. The functorial action reindexes variables, the monoidal unit $\nu_n : \mathrm{V}(n) = \mathbb{F}(1,n) \to \overline{\mathrm{T}}(n)$ gives variables-as-terms, and the multiplication $\mu : \overline{\mathrm{T}} \bullet \overline{\mathrm{T}} \Rightarrow \overline{\mathrm{T}}$ gives "substitution" or composition:

$$\mu_n : \int^m \overline{\mathrm{T}}^m(n) \times \overline{\mathrm{T}}(m) \to \overline{\mathrm{T}}(n)$$
$$[\langle \mathtt{o}_1, \ldots, \mathtt{o}_\mathtt{m}\rangle : \mathtt{s}^n \to \mathtt{s}^m, \mathtt{f} : \mathtt{s}^m \to \mathtt{s}] \mapsto \mathtt{f}(\mathtt{o}_1, \ldots, \mathtt{o}_\mathtt{m}) : \mathtt{s}^n \to \mathtt{s}.$$

Cartesian operads are equivalent to Lawvere theories, but theories as "context-indexed sets" admit constructions which theories as categories do not: exponentiating by representables gives *context extension.*

$$\overline{\mathrm{T}}^{\mathrm{V}}(n) \simeq \mathrm{Set}^{\mathbb{F}}(\mathbb{F}(n,-) \times \mathbb{F}(1,-), \overline{\mathrm{T}}) \simeq \mathrm{Set}^{\mathbb{F}}(\mathbb{F}(n+1,-), \overline{\mathrm{T}}) \simeq \overline{\mathrm{T}}(n+1).$$

We can then describe the inference rule for λ abstraction as a natural transformation:

$$\lambda : \overline{\mathrm{T}}^{\mathrm{V}} \Rightarrow \overline{\mathrm{T}} \qquad \lambda_n : \overline{\mathrm{T}}(n+1) \to \overline{\mathrm{T}}(n).$$

So in the same way that the structure of monoids is given by an endofunctor on Set, the pure λ-calculus is given by the signature for variables, application, and λ abstraction.

$$\Lambda : \mathrm{Set}^{\mathbb{F}} \to \mathrm{Set}^{\mathbb{F}} \qquad \Lambda(\overline{\mathrm{T}}) = \mathrm{V} + \overline{\mathrm{T}}^2 + \overline{\mathrm{T}}^{\mathrm{V}}$$

Hence, a class of monads on $\mathrm{Set}^{\mathbb{F}}$ correspond to "operads with binding". This notion of theory is equivalent to a category which is not closed, but equipped with a set of objects which serve as the binding types [3].

Definition 2. An object $\mathtt{s} \in \mathrm{T}$ is **exponentiable** if for all $\mathtt{t}$ there is an object $\mathtt{t}^{\mathtt{s}}$, equipped with a map $ev_{\mathtt{s},\mathtt{t}} : \mathtt{s} \times \mathtt{t}^{\mathtt{s}} \to \mathtt{t}$ which for every $u : \mathtt{a} \times \mathtt{s} \to \mathtt{t}$ there is a unique $u^{\bullet} : \mathtt{a} \to \mathtt{t}^{\mathtt{s}}$ so that $u = ev_{\mathtt{s},\mathtt{t}} \circ (u^{\bullet} \times id_{\mathtt{s}})$. A functor $F : \mathrm{T} \to \mathrm{C}$ **preserves** the exponentiable object $\mathtt{s}$ if $F(\mathtt{s})$ is exponentiable and for all $\mathtt{t}$, $ev^{\bullet}_{\mathtt{s},\mathtt{t}} : F(\mathtt{t}^{\mathtt{s}}) \simeq F(\mathtt{t})^{F(\mathtt{s})}$ and $F(ev_{\mathtt{s},\mathtt{t}}) = ev_{F(\mathtt{s}),F(\mathtt{t})}$. We call F *exponent* if it preserves all exponentiable objects.

Definition 3. A **second order algebraic theory** T is a cartesian category equipped with a set of exponentiable objects $\mathcal{X}$, given by a presentation $\langle(\mathcal{T}, \mathcal{O}, \mathcal{E}), \mathcal{X}\rangle$. The 2-category of second-order theories, cartesian exponent functors, and natural transformations is denoted SOAT. The category of **models** of T in **context** C is SOAT(T, C).

The object $\mathtt{t}^{\mathtt{s}}$ represents terms of type $\mathtt{t}$ with a free variable of type $\mathtt{s}$, and ev substitutes a term of type $\mathtt{s}$ for the free variable. Operations $\mathtt{f} : \mathtt{t}^{\mathtt{s}} \to \mathtt{a}$ are understood as binding a free variable $x.t : \mathtt{s}.\mathtt{t} \mapsto \mathtt{f}(t) : \mathtt{a}$. Though we do not need it here, second-order theories have an explicit syntax using metavariables [3].

Fiore et. al. continue the program of "algebraic type theory", which uses *generalized operads* [8] to encapsulate richer type theoretic notions such as polymorphism [6] and dependency [7].

2.2 The ρ-calculus

The ρ-calculus [22] has two sorts, processes `P` and names `N`. These act as code and data respectively, and the operations of reference @ and dereference $*$ transform one into the other. Terms are built up from the one constant, the null process `0`. The two basic actions of a process are output `out` and input `in`, and parallel | gives binary interaction: these earn their names in the communication rule.

Definition 4. ρ-calculus

$\mathtt{0} : 1 \to \mathtt{P}$	$\mid : \mathtt{P} \times \mathtt{P} \to \mathtt{P}$	$(\mathtt{P}, \mid, \mathtt{0})$	commutative monoid
$@ : \mathtt{P} \to \mathtt{N}$	$\mathtt{out} : \mathtt{N} \times \mathtt{P} \to \mathtt{P}$	reify	$*@(\mathtt{p}) = \mathtt{p}$
$* : \mathtt{N} \to \mathtt{P}$	$\mathtt{in} : \mathtt{N} \times \mathtt{P}^{\mathtt{N}} \to \mathtt{P}$	comm	$\mathtt{out}(\mathtt{n}, \mathtt{q}) \mid \mathtt{in}(\mathtt{n}, \mathtt{x}.\mathtt{p}) = \mathtt{p}[@\mathtt{q}/\mathtt{x}]$

Note The reader may see that we have equations rather than *reductions*; this of course oversimplifies the operational semantics of concurrent languages. The method of Plotkin and Turi [27] applies to languages without binding, and second-order logic with rewriting has been given in [4]. Incorporating this aspect is necessary to the project of structural types for algebraic theories.

Example 5. Though the ρ-calculus has a concise presentation, it is remarkably expressive. The λ-calculus embeds into the π-calculus [19], and the π-calculus embeds into the ρ-calculus [22]. For example, replication is used to make π-calculus processes persist over time, particularly in defining recursive processes. This can be expressed in the ρ-calculus without a designated operator, using reflection:

$$\begin{aligned}
&\text{pick a name} && \mathtt{n}, \\
&\text{define copier} && c(\mathtt{n}) := \mathtt{in}(\mathtt{n}, \mathtt{x}.\{\mathtt{out}(\mathtt{n}, *\mathtt{x}) \mid *\mathtt{x}\}) \\
&\text{and replicator} && !(-)_{\mathtt{n}} := \mathtt{out}(\mathtt{n}, \{c(\mathtt{n}) \mid -\}) \mid c(\mathtt{n}). \\
&\text{Then we have} && !(\mathtt{p})_{\mathtt{n}} = \mathtt{out}(\mathtt{n}, \{c(\mathtt{n}) \mid \mathtt{p}\}) \mid \mathtt{in}(\mathtt{n}, \mathtt{x}.\{\mathtt{out}(\mathtt{n}, *\mathtt{x}) \mid *\mathtt{x}\}) \\
& && \quad = \mathtt{out}(\mathtt{n}, \{c(\mathtt{n}) \mid \mathtt{p}\}) \mid *@\{c(\mathtt{n}) \mid \mathtt{p}\} \\
& && \quad = \mathtt{out}(\mathtt{n}, \{c(\mathtt{n}) \mid \mathtt{p}\}) \mid c(\mathtt{n}) \mid \mathtt{p} \quad = \ !(\mathtt{p})_{\mathtt{n}} \mid \mathtt{p}.
\end{aligned}$$

In demonstrating Namespace Logic §5, such recursive processes are terms of recursive types; these express properties of persistent behavior. We will give the expressive example of a predicate sole.in, which for a namespace α expresses "can always input on α, and can never input on $\neg\alpha$" – effectively a kind of firewall.

3 Topos of presheaves

We can embed a second-order theory such as the ρ-calculus into a category with rich internal logic, to utilize the structure of code by reasoning about subclasses of terms.

A category of presheaves $[\mathrm{T}^{\mathrm{op}}, \mathrm{Set}] =: \widehat{\mathrm{T}}$ is a *topos* [18]: it is cartesian closed, finite limits are given pointwise, and exponentials and subobject classifer are given by

$$[P, Q](\mathtt{s}) = \widehat{\mathrm{T}}(\mathrm{T}(-, \mathtt{s}) \times P, Q) \qquad\qquad \Omega(\mathtt{s}) = \{P \mid P \rightarrowtail \mathrm{T}(-, \mathtt{s})\}.$$

The subobject classifier allows us to reason with presheaves in the language of sets, using intuitionistic predicate logic. The definition above can be understood in light of the equivalence:

$$\{\text{subfunctors of } \mathrm{T}(-, \mathtt{s})\} \simeq \{\text{sieves on } \mathtt{s}\}.$$

A *sieve* on $\mathtt{s}$ is a class of morphisms in T with codomain $\mathtt{s}$, which is closed under precomposition. This generalizes the notion of right ideal of a monoid, which gives the equivalence: the representable $\mathrm{T}(-, \mathtt{s}) =: \hat{\mathtt{s}}$ is a right T-module, and sieves on $\mathtt{s}$ are submodules thereof.

The simplest example besides the maximal sieve $\mathrm{T}(-, \mathtt{s})$ is a *singleton sieve*:

$$\mathtt{f} : \mathtt{s} \to \mathtt{t} \quad \mapsto \quad S(\mathtt{f})(\mathtt{r}) := \{\mathtt{op} : \mathtt{r} \to \mathtt{t} \mid \exists u : \mathtt{r} \to \mathtt{s}.\ \mathtt{op} = \mathtt{f}(u)\}.$$

As suggested by notation, our interest is sieves in a theory T: these act as predicates or *types*. We recall that Ω is an internal Heyting algebra, providing the constructors of predicate logic; and in §4 we demonstrate how operations of T can be lifted to the level of sieves, providing algebraic type constructors.

For example, if T is the theory of monoids, we can construct the sieve of "prime operations"

$$\mathtt{prime} := \neg[S(\mathtt{e})] \wedge \neg[\bar{\mathtt{m}}(\neg[S(\mathtt{e})], \neg[S(\mathtt{e})])].$$

3.1 The subobject functor

The subobject classifier is so named because it represents the (contravariant) *subobject functor*

$$\widehat{\mathrm{T}}(-, \Omega) \simeq \mathrm{Sub}(-) : \widehat{\mathrm{T}}^{\mathrm{op}} \to \mathrm{Pos}.$$

Characteristic maps $\chi_\varphi : A \to \Omega$ correspond to subobjects $\varphi \rightarrowtail A$, and precomposing by $f : B \to A$ corresponds to pullback $f^* : \mathrm{Sub}(A) \to \mathrm{Sub}(B)$.

For any presheaf A, subfunctors of A form a complete Heyting algebra. They are ordered by inclusion, with meet and join defined by pointwise intersection and union. Implication is defined:

$$(\psi \Rightarrow \varphi)(\mathtt{t}) := \{a \in A(\mathtt{t}) \mid \forall u : \mathtt{s} \to \mathtt{t}.\ a \cdot u \in \psi(\mathtt{s}) \Rightarrow a \cdot u \in \varphi(\mathtt{s})\}$$

and negation is $\neg[\varphi] := \varphi \Rightarrow \emptyset$.

Lemma 6. The subobject functor is a *hyperdoctrine* [15]: for each $f : B \to A$ there is a triple adjunction $\exists_f \dashv f^* \dashv \forall_f$ which satisfies that quantification commutes with substitution and $\forall_f$ preserves implication. While f^* is a morphism of Heyting algebras, $\exists_f$ and $\forall_f$ in general only preserve join and meet, respectively.

$$\begin{tikzcd} \mathrm{Sub}(B) \arrow[r, shift left=2, "\exists_f"] \arrow[r, shift right=2, "\forall_f"'] & \mathrm{Sub}(A) \arrow[l, "f^*" description] \end{tikzcd}$$

These adjoints are intuited in the case of Set. For a function $f : A \to B$, the left adjoint $\exists_f$ is *direct image*, and the right adjoint $\forall_f$ is *saturated image*, where $\forall_f(U)$ is the maximal subset of $\exists_f(U)$ whose preimage is contained in U. In a presheaf topos $\widehat{\mathrm{T}}$ these morphisms are defined for a subobject $\varphi \rightarrowtail A$:

$$\exists_f(\varphi)(\mathtt{t}) = \{b \in B(\mathtt{t}) \mid \exists u : \mathtt{s} \to \mathtt{t}.\ \exists a \in A.\ b \cdot u = f_{\mathtt{s}}(a) \wedge a \in \varphi(\mathtt{s})\}$$
$$\forall_f(\varphi)(\mathtt{t}) = \{b \in B(\mathtt{t}) \mid \forall u : \mathtt{s} \to \mathtt{t}.\ \forall a \in A.\ b \cdot u = f_{\mathtt{s}}(a) \Rightarrow a \in \varphi(\mathtt{s})\}.$$

Note There are three possible actions of Sub, one contravariant and two covariant. When lifting operations to sieves, we focus on the left adjoint, but each is useful: the middle adjoint splits terms apart by "undoing" operations, while the right adjoint applies operations "securely", giving only the terms which could have come from that application. These are very expressive.

The composite $\mathrm{T} \to \widehat{\mathrm{T}} \to \mathrm{Pos}$ induces a preorder fibration $\int \mathrm{Sub} \circ y \to \widehat{\mathrm{T}}$, providing the canonical example of a "logic over a type theory" [12]. We expand on this idea, using the fact that Sub is both lax cartesian and colax closed to lift operations of a second-order algebraic theory.

3.2 Lax structure

We henceforth use $\mathcal{P} : \widehat{\mathrm{T}} \to \mathrm{Pos}$ to denote the subobject functor with $\mathcal{P}(f) := \exists_f$. Let $A, B : \mathrm{T}^{\mathrm{op}} \to \mathrm{Set}$.

Every functor has *colax* preservation of products: there is a canonical natural transformation given by the pairing of the projections. For $R \rightarrowtail A \times B$ and denoting $R_A(\mathtt{t}) = \{a \in A(\mathtt{t}) \mid \exists b \in B(\mathtt{t})\ (a, b) \in R(\mathtt{t})\}$,

$$\lambda_{AB} : \mathcal{P}(A \times B) \to \mathcal{P}(A) \times \mathcal{P}(B) \quad \text{by} \quad \lambda_{AB}(R) = (R_A, R_B).$$

We are interested in *lax* preservation of products, denoted $\sqcap$, so that we can lift operations

$$\mathtt{f} : \mathtt{a} \times \mathtt{b} \to \mathtt{c} \quad \mapsto \quad \bar{\mathtt{f}} := \mathcal{P}(\mathtt{f}) \circ \sqcap : \mathcal{P}(y(\mathtt{a})) \times \mathcal{P}(y(\mathtt{b})) \to \mathcal{P}(y(\mathtt{a} \times \mathtt{b})) \to \mathcal{P}(y(\mathtt{c})).$$

Lemma 7. Let $(\mathcal{P}, \lambda) : \widehat{\mathrm{T}} \to \mathrm{Pos}$ be the colax subobject functor. Define

$$\sqcap_{AB} : \mathcal{P}(A) \times \mathcal{P}(B) \to \mathcal{P}(A \times B) \quad \text{by} \quad \sqcap_{AB}(U, V) = U \times V.$$

Then $(\mathcal{P}, \sqcap)$ is a lax functor, and for all $A, B \in \widehat{\mathrm{T}}$ the component $\sqcap_{AB}$ is right adjoint to λ_{AB}.

By the terminology of [10], $(\mathcal{P}, \sqcap)$ is *adjoint-lax*: every component of the laxor $\sqcap_{AB}$ has a left adjoint. Moreover, we can lift binding operations: by Lemma 9, we can define

$$\mathtt{f} : [\mathtt{s}, \mathtt{t}] \to \mathtt{a} \quad \mapsto \quad \bar{\mathtt{f}} := \mathcal{P}(\mathtt{f}) \circ \mathrm{R}_{y(\mathtt{s})y(\mathtt{t})} : [\mathcal{P}(y(\mathtt{s})), \mathcal{P}(y(\mathtt{t}))] \to \mathcal{P}(y([\mathtt{s}, \mathtt{t}])) \to \mathcal{P}(\mathtt{a}).$$

The product laxor $\sqcap$ gives an "exponent laxor" Λ which is the currying of evaluation.

For sets, we have definition by pointwise evaluation:

$$\Lambda_{AB} : \mathcal{P}([A, B]) \to [\mathcal{P}(A), \mathcal{P}(B)] \quad \text{by} \quad \Lambda_{AB}(X)(U) = \{ev(f, a) \mid f \in X, a \in U\}.$$

For presheaves, define

$$\Lambda_{AB} : \mathcal{P}([A, B]) \to [\mathcal{P}(A), \mathcal{P}(B)] \quad \text{by} \quad \Lambda_{AB}(X)(S)(\mathtt{c}) = \{ev(f, a) \mid f \in X(\mathtt{c}), a \in S(\mathtt{c})\}.$$

For sets, Λ_{AB} has a right adjoint sending a power set map F to the subset of functions that "respect" F:

$$\mathrm{R}_{AB}(F) = \{f : A \to B \mid \forall U \in \mathcal{P}(A).\ \mathcal{P}(f)(U) \subseteq F(U)\}.$$

This generalizes to presheaves. Let $A, B : \mathrm{T}^{\mathrm{op}} \to \mathrm{Set}$, and recall that $[A, B](\mathtt{c}) = \widehat{\mathrm{T}}(y(\mathtt{c}) \times A, B)$.

Definition 8. We say that $f \in [A, B](\mathtt{c})$ **respects** a functor $F : \mathcal{P}(A) \to \mathcal{P}(B)$ if every sieve S on A paired with $y(\mathtt{c})$ has direct image contained in $F(S)(\mathtt{c})$:

$$\{f \text{ respects } F\} \ :=\ \forall S \in \mathcal{P}(A).\ \mathcal{P}(f)(y(\mathtt{c}) \times S) \subseteq F(S)(\mathtt{c}).$$

Lemma 9. Let $(\mathcal{P}, \Lambda) : \widehat{\mathrm{T}} \to \mathrm{Pos}$ be the lax closed subobject functor. Define

$$\mathrm{R}_{AB} : [\mathcal{P}(A), \mathcal{P}(B)] \to \mathcal{P}([A, B]) \quad \text{by} \quad \mathrm{R}_{AB}(F)(\mathtt{c}) = \{f \in [A, B](\mathtt{c}) \mid f \text{ respects } F\}.$$

Then for all $A, B \in \widehat{\mathrm{T}}$, R_{AB} is right adjoint to Λ_{AB}.

Proof. Suppose $X \in \mathcal{P}([A, B])$ is a subfunctor of an exponential of presheaves, and that

$$\Lambda(X) \leq F : \mathcal{P}(A) \to \mathcal{P}(B).$$

This means for all subfunctors $S \in \mathcal{P}(A)$, $\Lambda(X)(S) \leq F(S) \in \mathcal{P}(B)$; and this in turn means for all objects $\mathtt{c} \in \mathrm{T}$ we have $\Lambda(X)(S)(\mathtt{c}) \subseteq F(S)(\mathtt{c})$. Expanding the definition,

$$\Lambda(X)(S)(\mathtt{c}) = \{ev(f, u) \mid f \in X(\mathtt{c}), u \in S(\mathtt{c})\} \subseteq F(S)(\mathtt{c}).$$

This pointwise evaluation can be reformulated as a union over all $f \in X(\mathtt{c})$. For all S and $\mathtt{c}$, we have

$$\Lambda(X)(S)(\mathtt{c}) = (\bigcup_{f \in X(\mathtt{c})} \mathcal{P}(f)(y(\mathtt{c}) \times S)) \subseteq F(S)(\mathtt{c}).$$

Then by the universal property of coproduct, this is equivalent to every f having $\mathcal{P}(f)(y(\mathtt{c}) \times S) \subseteq F(S)(\mathtt{c})$. This is precisely the condition for $f \in \mathrm{R}(F)(\mathtt{c})$. Hence we have the desired result,

$$X \leq R(F) \in \mathcal{P}([A, B]).$$

□

Hence $\mathcal{P}$ is not only adjoint-lax with respect to products, but exponents as well. This is expressive: considering subfunctors as predicates, the family of adjunctions

$$\mathrm{R}_{AB} \vdash \Lambda_{AB} : \qquad \begin{tikzcd} {[\mathcal{P}(A), \mathcal{P}(B)]} \arrow[r, shift left] & \mathcal{P}([A, B]) \arrow[l, shift left] \end{tikzcd}$$

transforms maps of predicates to predicates on contexts, and vice versa. This allows for the evaluation of formulae on the level of predicates, providing *type-level binding.*

4 Structural types

We can now lift the operations of a theory to the level of sieves, to create type constructors.

Definition 10. Let $\mathtt{f} : \prod_{i=1}^{n}[\prod_{j=1}^{n_i} \mathtt{s}_{ij}, \mathtt{t}_i] \to \mathtt{t}$ be an operation in a second-order theory T. Define the **lifting** of $\mathtt{f}$ to be the composite given below.

$$\begin{tikzcd} \prod_{i=1}^{n}[\prod_{j=1}^{n_i} \mathcal{P}(y(\mathtt{s}_{ij})), \mathcal{P}(y(\mathtt{t}_i))] \arrow[rr, dotted, "\bar{\mathtt{f}}"] \arrow[d, "\prod[\lambda, id]"'] & & \mathcal{P}(y(\mathtt{t})) \\ \prod_{i=1}^{n}[\mathcal{P}(\prod_{j=1}^{n_i} y(\mathtt{s}_{ij})), \mathcal{P}(y(\mathtt{t}_i))] \arrow[r, "\prod \mathrm{R}"'] & \prod_{i=1}^{n} \mathcal{P}(y([\prod_{j=1}^{n_i} \mathtt{s}_{ij}, \mathtt{t}_i])) \arrow[r, "\sqcap"'] & \mathcal{P}(y(\prod_{i=1}^{n}[\prod_{j=1}^{n_i} \mathtt{s}_{ij}, \mathtt{t}_i])) \arrow[u, "\mathcal{P}(\mathtt{f})"'] \end{tikzcd}$$

Hence for $(F_i : \prod_{j=1}^{n_i} \mathcal{P}(y(\mathtt{s}_{ij})) \to \mathcal{P}(y(\mathtt{t}_i)))_{i=1}^{n}$,

$$\bar{\mathtt{f}}(F_1, \ldots, F_n)(\mathtt{r}) = \{\mathtt{f}(u_1, \ldots, u_n) : \mathtt{r} \to \mathtt{t} \mid \forall i.\ u_i \text{ respects } F_i \circ \lambda\}.$$

We summarize the data of the lifting by giving a model of the theory in posets. We denote currying $\mathtt{f} : \mathtt{a} \times \mathtt{b} \to \mathtt{c}$ by $\mathtt{f}^\bullet : \mathtt{a} \to [\mathtt{b}, \mathtt{c}]$, and uncurrying $u : \mathtt{a} \to [\mathtt{b}, \mathtt{c}]$ by $u^\circ : \mathtt{a} \times \mathtt{b} \to \mathtt{c}$.

Theorem 11. Let $\mathrm{T} = \langle (\mathcal{T}, \mathcal{O}, \mathcal{E}), \mathcal{X} \rangle$ be a second-order theory. The lifting (10) defines a colax functor

$$\omega_{\mathrm{T}} : \mathrm{T} \to \mathrm{Pos}.$$

Moreover, ω_{T} preserves products and exponentials by construction, giving a "colax model" of T in Pos.

Proof. Define

$$\omega_{\mathrm{T}}(\prod_{i=1}^{n}[\prod_{j=1}^{n_i} \mathtt{s}_{ij}, \mathtt{t}_i]) = \prod_{i=1}^{n}[\prod_{j=1}^{n_i} \mathcal{P}(y(\mathtt{s}_{ij})), \mathcal{P}(y(\mathtt{t}_i))]$$

and for $\mathtt{f} : \prod_{i=1}^{n}[\prod_{j=1}^{n_i} \mathtt{s}_{ij}, \mathtt{t}_i] \to \mathtt{t}$ in $\mathcal{O}$ define

$$\omega_{\mathrm{T}}(\mathtt{f}) = \bar{\mathtt{f}} : \prod_{i=1}^{n}[\prod_{j=1}^{n_i} \mathcal{P}(y(\mathtt{s}_{ij})), \mathcal{P}(y(\mathtt{t}_i))] \to \mathcal{P}(y(\mathtt{t})).$$

A general operation $\mathtt{g} : \prod_{k=1}^{m}[\prod_{l=1}^{m_k} \mathtt{p}_{kl}, \mathtt{q}_k] \to \prod_{i=1}^{n}[\prod_{j=1}^{n_i} \mathtt{s}_{ij}, \mathtt{t}_i]$ is equivalent to an n-tuple of operations $\langle \mathtt{g}_i^\circ : \prod_{k=1}^{m}[\prod_{l=1}^{m_k} \mathtt{p}_{kl}, \mathtt{q}_k] \times \prod_{j=1}^{n_i} \mathtt{s}_{ij} \to \mathtt{t}_i \rangle_n$; we thereby define

$$\omega_{\mathrm{T}}(\mathtt{g}) = \langle \overline{\mathtt{g}_1^\circ}^\bullet, \dots, \overline{\mathtt{g}_n^\circ}^\bullet \rangle =: \bar{\mathtt{g}} : \prod_{k=1}^{m}[\prod_{l=1}^{m_k} \mathcal{P}(y(\mathtt{p}_{kl})), \mathcal{P}(y(\mathtt{q}_k))] \to \prod_{i=1}^{n}[\prod_{j=1}^{n_i} \mathcal{P}(y(\mathtt{s}_{ij})), \mathcal{P}(y(\mathtt{t}_i))].$$

We now give colax functoriality, by giving for the right diagram a canonical inclusion $\overline{\mathtt{f} \circ \mathtt{g}} \le \bar{\mathtt{f}} \circ \bar{\mathtt{g}}$.

$$\begin{tikzcd}
\prod_{k=1}^{m}[\prod_{l=1}^{m_k} \mathtt{p}_{kl}, \mathtt{q}_k] \arrow[d, "\mathtt{g}"'] \arrow[dr, "\mathtt{f}\circ\mathtt{g}"] & \\
\prod_{i=1}^{n}[\prod_{j=1}^{n_i} \mathtt{s}_{ij}, \mathtt{t}_i] \arrow[r, "\mathtt{f}"'] & \mathtt{t}
\end{tikzcd}
\qquad
\begin{tikzcd}
\prod_{k=1}^{m}[\prod_{l=1}^{m_k} \mathcal{P}(y(\mathtt{p}_{kl})), \mathcal{P}(y(\mathtt{q}_k))] \arrow[d, "\bar{\mathtt{g}}"'] \arrow[dr, "\overline{\mathtt{f}\circ\mathtt{g}}"] & \\
\prod_{i=1}^{n}[\prod_{j=1}^{n_i} \mathcal{P}(y(\mathtt{s}_{ij})), \mathcal{P}(y(\mathtt{t}_i))] \arrow[r, "\bar{\mathtt{f}}"'] & \mathcal{P}(y(\mathtt{t}))
\end{tikzcd}$$

Let $(F_k : \prod_{l=1}^{m_k} \mathcal{P}(y(\mathtt{p}_{kl})) \to \mathcal{P}(y(\mathtt{q}_k)))_{k=1}^{m}$ be maps of subfunctor posets. Suppose $u \in \overline{\mathtt{f} \circ \mathtt{g}}(F_1, \dots, F_k)(\mathtt{r})$.

$$u = \mathtt{f}(\mathtt{g}(v_1, \dots, v_m)) \text{ for } (v_k : \mathtt{r} \to \prod_{k=1}^{m}[\prod_{l=1}^{m_k} \mathtt{p}_{kl}, \mathtt{q}_k])_{k=1}^{m} \text{ which each respect } F_k.$$

As $\mathtt{g} = \langle \mathtt{g}_1, \dots, \mathtt{g}_n \rangle$, we have that $\mathtt{f}(\mathtt{g}(\vec{v_k})) = \mathtt{f}(\mathtt{g}_1(\vec{v_k}), \dots, \mathtt{g}_n(\vec{v_k}))$. Because we want to show that

$$\overline{\mathtt{f} \circ \mathtt{g}}(F_1, \dots, F_m)(\mathtt{r}) \subseteq \bar{\mathtt{f}}(\bar{\mathtt{g}}(F_1, \dots, F_m))(\mathtt{r}) = \bar{\mathtt{f}}(\overline{\mathtt{g}_1^\circ}^\bullet(\vec{F_k}), \dots, \overline{\mathtt{g}_n^\circ}^\bullet(\vec{F_k}))(\mathtt{r}),$$

we only need to show that each $\mathtt{g}_i(\vec{v_k})$ respects $\overline{\mathtt{g}_i^\circ}^\bullet(\vec{F_k})$. Expanding the definition, we check

$$\forall U \in \mathcal{P}(\prod_{j=1}^{n_i} y(\mathtt{s}_{ij})).\ \mathcal{P}(g_i(\vec{v_k}))(\hat{\mathtt{r}} \times U) \subseteq \overline{\mathtt{g}_i^\circ}^\bullet(\vec{F_k})(U);$$

where the latter sieve is given by

$$\overline{\mathtt{g}_i^\circ}^\bullet(\vec{F_k})(U)(\mathtt{r}) = \{\mathtt{g}_i^\circ(\vec{v_k}, (x_1, \dots, x_{n_i})) : \mathtt{r} \to \mathtt{t}_i \mid \forall k.\ (v_k \text{ respects } F_k) \wedge \forall j.\ (x_j \in U(\mathtt{r}))\}.$$

Any element $u_i \in \mathcal{P}(g_i(\vec{v_k}))(\hat{\mathtt{r}} \times U)$ is automatically of this form. Hence we have the containment:

$$[u = \mathtt{f}(\mathtt{g}(v_1, \dots, v_m)) \in \overline{\mathtt{f} \circ \mathtt{g}}(F_1, \dots, F_k)(\mathtt{r})] \Rightarrow [u \in \bar{\mathtt{f}}(\bar{\mathtt{g}}(F_1, \dots, F_m))(\mathtt{r})].$$

Because Pos is locally preordered, coherence of the colax structure is trivial. Hence, $\omega_{\mathrm{T}} : \mathrm{T} \to \mathrm{Pos}$ is a colax functor which preserves products and exponents. □

Definition 12. The **structural theory** of T is ω_T. The **category of constructors** of T is the full image of ω_T, the full subcategory $\omega_\mathrm{T}(\mathrm{T}) \subset \mathrm{Pos}$ containing all $\omega_\mathrm{T}(\mathtt{s})$. We abbreviate $\omega_\mathrm{T}(\mathtt{s}) =: \mathtt{s}_\omega$

Theorem 13. The map of Thm. 11 defines a 2-functor $\omega : \mathrm{SOAT} \to (\iota \downarrow \mathrm{Pos})$, where the latter is the comma 2-category of the inclusion $\iota : \mathrm{SOAT} \rightarrowtail 2\mathrm{Cat}_{colax}$ and the constant $\mathrm{Pos} : 1 \to 2\mathrm{Cat}$.

Note The colaxity of ω_T is inevitable when lifting binding operations. For example, consider the ρ-calculus term $\mathtt{in}(-, x.\mathtt{out}(x, -)) : \mathtt{N} \times \mathtt{P} \to \mathtt{P}$. Then

$$\overline{\mathtt{in}}(\alpha, \overline{\mathtt{out}^\bullet}(\varphi)) = \overline{\mathtt{in}}(\alpha, \beta.\overline{\mathtt{out}}(\beta, \varphi)) \text{ , while } \overline{\mathtt{in} \circ (\mathtt{N} \times \mathtt{out}^\bullet)}(\alpha, \varphi) = \overline{\mathtt{in}}(\alpha, x.\overline{\mathtt{out}}(x, \varphi)).$$

The lifting of the composite doesn't "see" the type-level binding for constructors; nevertheless, the information is retained by the colax structure.

Let T be the theory of the ρ-calculus [§2.2]. The lifted signature provides the algebraic type constructors of namespace logic [§5].

Definition 14. The ωρ**-calculus** has algebraic type constructors:

$$\begin{array}{llll} \bar{\mathtt{0}} : 1 & \to \mathtt{P}_\omega & \bar{|} : \mathtt{P}_\omega \times \mathtt{P}_\omega & \to \mathtt{P}_\omega \\ \bar{@} : \mathtt{P}_\omega \to \mathtt{N}_\omega & & \overline{\mathtt{out}} : \mathtt{N}_\omega \times \mathtt{P}_\omega & \to \mathtt{P}_\omega \\ \bar{*} : \mathtt{N}_\omega \to \mathtt{P}_\omega & & \overline{\mathtt{in}} : \mathtt{N}_\omega \times [\mathtt{N}_\omega, \mathtt{P}_\omega] & \to \mathtt{P}_\omega. \end{array}$$

We have products and exponent types given by the theory, and $y(\mathtt{P}) : 1 \to \mathtt{P}_\omega$ and $y(\mathtt{N}) : 1 \to \mathtt{N}_\omega$, the maximal sieves which we abbreviate to $\mathtt{P}$ and $\mathtt{N}$. From these we generate the singleton sieves, such as $\bar{\mathtt{0}}(\mathtt{P})(\mathtt{t}) = \{\mathtt{0} \circ !_\mathtt{t} : \mathtt{t} \to \mathtt{P}\}$, which we overload and denote by $\bar{\mathtt{0}}$.

We can now form predicates on names, processes, or contexts thereof. For example, there is a predicate expressing that a process is *single-threaded*: it is not null, and not the parallel of two non-null processes.

$$\mathsf{single.thread} := \neg[\bar{\mathtt{0}}] \wedge \neg[\neg[\bar{\mathtt{0}}] \;\bar{|}\; \neg[\bar{\mathtt{0}}]]$$

By lifting the binding operation of input, $\mathtt{in} : \mathtt{N} \times [\mathtt{N}, \mathtt{P}] \to \mathtt{P}$, we gain significant expressivity. In the same way that input binds a free name variable, the type constructor $\overline{\mathtt{in}} : \mathtt{N}_\omega \times [\mathtt{N}_\omega, \mathtt{P}_\omega] \to \mathtt{P}_\omega$ binds a free *namespace* variableinto a constructor type constructor $\Phi : \mathtt{N}_\omega \to \mathtt{P}_\omega$.

Terms of this type are constructed by the following inference rule (§6). We use the notation of categorical logic [12]: Ξ is a type context, and Γ is a term context.

$$\frac{\Xi, \alpha : \mathtt{N}_\omega \mid \Gamma \vdash \mathtt{a} : \alpha \qquad \Xi, \chi : \mathtt{N}_\omega \mid \Gamma, \mathtt{x} : \chi \vdash \mathtt{p} : \Phi}{\Xi \mid \Gamma \vdash \mathtt{in}(\mathtt{a}, \mathtt{x}.\mathtt{p}) : \overline{\mathtt{in}}(\alpha, \chi.\Phi).}$$

In the untyped language, the process $\mathtt{in}(\mathtt{n}, \mathtt{x}.\mathtt{p})$ receives data over $\mathtt{n}$ and substitutes into $\mathtt{p}[\mathtt{x}]$. In the ωρ-calculus, a term of type $\overline{\mathtt{in}}(\alpha, \chi.\Phi)$ is a process which inputs on a name in α, and receiving a name of type χ gives a process of type $\Phi(\chi)$. One can use typed input to design *structural queries*: programs which search not by external attributes, but by the actual structure of code.

5 Namespace Logic

The motivation of namespace logic is to develop a theory of the structure of names as used in distributed computing. It demonstrates the utility of reflection, thereby drawing a sharp contrast between the atomic names of the π-calculus and the structured names of the ρ-calculus.

The term "namespace" comes from thinking of a subset $\varphi(1) \subseteq \mathrm{T}(1, \mathtt{N})$ as a space of locations in a network, or as a space of data which may be communicated. The namespace φ might be a collection of trusted addresses for an organization or it could be a datatype, such as the XML schema.

Of course, since names are the references of processes, we expect the same reasoning for "codespaces". As we have seen, the framework of structural types does this and more: we have predicates for all $\mathrm{T}(-, \mathtt{t})$, which in general give tuples of process and name contexts.

We have built all of the tools used in namespace logic, except for one. For types which involve infinitary operations, such as those which allow for an indeterminate number of operands, or those which predicate over all possible evolutions of a process, we need to add *fixed point type constructors.*

5.1 Recursion

The final aspect of the type theory is limits and colimits.

Definition 15. Let T be a second-order algebraic theory, and let $\omega_T : \mathrm{T} \to \mathrm{Pos}$ be the structural theory of T. Let $\Phi : \mathtt{s}_\omega \to \mathtt{t}_\omega$ be a morphism in the category of constructors of T.

Because each $\mathtt{s}_\omega$ is complete and cocomplete, the limit of Φ and the colimit of Φ exist, denoted

$$\mathrm{lim}\Phi = \bigwedge_{\varphi \in \mathtt{s}_\omega} \Phi(\varphi) \qquad \text{and} \qquad \bigvee_{\varphi \in \mathtt{s}_\omega} \Phi(\varphi).$$

In the case that $\mathtt{s} = \mathtt{t}$, denote the *greatest fixed point* and *least fixed point* as special limits and colimits:

$$\nu\mathtt{X}.\Phi(\mathtt{X}) := \bigwedge_{\varphi \leq \Phi(\varphi)} \Phi(\varphi) \qquad \text{and} \qquad \mu\mathtt{X}.\Phi(\mathtt{X}) := \bigvee_{\Phi(\varphi) \leq \varphi} \Phi(\varphi).$$

Define the **predicate completion** of T to be a choice of limit and colimit for every $\Phi : \mathtt{s}_\omega \to \mathtt{t}_\omega$ in the category of constructors of T:

$$\mathrm{T}^{\mathrm{lim}}_{\mathrm{colim}} := \sum_{\mathtt{s} \in \mathrm{T}} \sum_{\mathtt{t} \in \mathrm{T}} \sum_{\Phi : \mathtt{s}_\omega \to \mathtt{t}_\omega} (\mathrm{lim}\Phi, \mathrm{colim}\Phi).$$

These provide the structural type theory with recursive types, which can be used to construct data types, to ensure certain persistent behavior, and for many other purposes. We demonstrate one simple example, and one which begins to illustrate the expressivity of namespace logic.

Example 16. A general process in the ρ calculus is a parallel of a single-threaded "parands":

$$\mathtt{p} = \mathtt{p}_1 \mid \ldots \mid \mathtt{p}_n.$$

When considering a predicate such as $\varphi = \overline{\mathtt{in}}(\alpha, \mathtt{N.P})$, one may require only that some parand inputs on α; this is given by the predicate $\overline{\mathtt{in}}(\alpha, \mathtt{N.P}) \mid \mathtt{P}$. But often, one may require a predicate to hold for every parand. This is given by the recursive type which is analogous to the free monoid on a set:

$$\varphi^* := \nu\mathtt{X}.\ \varphi \mid \mathtt{X}.$$

Example 17. Two important properties of a distributed system are *liveness* and *safety*. Suppose we have a namespace α of all names trusted by a system of processes S.

Liveness : S can always communicate on α.
Safety : S can never communicate on $\neg\alpha$.

We can express these conditions as a recursive structural type.

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

In effect, we are making a *firewall*: a process satisfies this predicate if and only if

$$\begin{aligned} \mathtt{p} : \mathsf{sole.in}(\alpha) \equiv \exists\ \mathtt{a} : \alpha. \qquad & \mathtt{p} = \mathtt{in}(\mathtt{a}, \mathtt{x.q}) \mid \mathtt{p}' \\ \wedge \forall\ \mathtt{n} : \mathtt{N}, \mathtt{q}[\mathtt{x}] : [\mathtt{N}, \mathtt{P}].\ & [\mathtt{p} = \mathtt{in}(\mathtt{n}, \mathtt{x.q}) \mid \mathtt{p}'] \Rightarrow [\mathtt{n} : \alpha \wedge \mathtt{q} : \mathsf{sole.in}(\alpha)]. \end{aligned}$$

One can see that the type-checking procedure is recursive; hence a term of type $\mathsf{sole.in}(\alpha)$ must be recursively defined, which can be done through reflection (§2.2).

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 (§3) 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.

6 The type theory

We give a brief summary of the structural type theory generated by Theorem 11 and Definition 15. The semantics and formal presentation corresponding to Jacobs' Categorical Logic [12] is forthcoming.

Definition 18. Let $\mathrm{T} = \langle (\mathcal{T}, \mathcal{O}, \mathcal{E}), \mathcal{X} \rangle$ be a second-order algebraic theory. Let $\omega_{\mathrm{T}} : \mathrm{T} \to \mathrm{Pos}$ be the structural theory of T, and let $\mathrm{T}^{\mathrm{lim}}_{\mathrm{colim}}$ be a predicate completion of T. The data $\omega\mathrm{T} := \langle \omega_{\mathrm{T}}, \mathrm{T}^{\mathrm{lim}}_{\mathrm{colim}} \rangle$ constitute the **structural type theory** of T.

$\omega\mathrm{T}$

Kinds The sorts $\mathtt{s} \in \mathrm{T}$. Constructors: products in T and exponents by $\mathtt{t} \in \mathcal{X}$.
Types The objects of $\int \omega_{\mathrm{T}}$ which can be derived by the following constructors.

Lifted operations	$\sum_{\mathtt{s}\in\mathrm{T}} \sum_{\mathtt{t}\in\mathrm{T}}$	$\{\bar{\mathtt{f}} : \mathtt{s}_\omega \to \mathtt{t}_\omega\}$
Predicate logic	$\sum_{\mathtt{s}\in\mathrm{T}}$	$\{\wedge_{\mathtt{s}}, \vee_{\mathtt{s}} : \mathtt{s}_\omega^2 \to \mathtt{s}_\omega\}$
	$\sum_{\mathtt{s}\in\mathrm{T}}$	$\{\Rightarrow_{\mathtt{s}} : \mathtt{s}_\omega^{\mathrm{op}} \times \mathtt{s}_\omega \to \mathtt{s}_\omega\}$
Fixed points	$\sum_{\mathtt{s}\in\mathrm{T}}$	$\{\nu_{\mathtt{s}}, \mu_{\mathtt{s}} : [\mathtt{s}_\omega, \mathtt{s}_\omega] \to \mathtt{s}_\omega\}$
Limits and Colimits	$\sum_{\mathtt{s}\in\mathrm{T}} \sum_{\mathtt{t}\in\mathrm{T}}$	$\{\lim_{\mathtt{st}}, \mathrm{colim}_{\mathtt{st}} : [\mathtt{s}_\omega, \mathtt{t}_\omega] \to \mathtt{t}_\omega\}$

Terms The rules for predicate logic are known; those for fixed points are limits and colimits, which are analogous to Σ and Π types of dependent type theory. The following inference is the introduction rule for terms in $\omega\mathrm{T}$ constructed by operations of T and their liftings; the equations of T lift to equations in ω_{T}, and the polymorphism of operations with respect to their liftings is automatic from the construction of ω_{T}.

$$\{\mathtt{f} : \vec{\mathtt{s}_1}.\mathtt{t}_1, \ldots, \vec{\mathtt{s}_k}.\mathtt{t}_k \to \mathtt{t}\} \; \frac{\Xi, \vec{\chi_i} : \vec{\mathtt{s}_\omega} \vdash \Phi : \mathtt{t}^i_\omega \qquad \Xi \mid \Gamma, \vec{\mathtt{x}_i} : \vec{\chi_i} \vdash u_i : \Phi_i \qquad (1 \leq i \leq n)}{\Xi \mid \Gamma \vdash \mathtt{f}(u_1, \ldots, u_n) : \bar{\mathtt{f}}(\vec{\chi_1}.\Phi_1, \ldots, \vec{\chi_n}.\Phi_n) :: \mathtt{t}_\omega}$$

References

[1] J. Adámek and J. Rosický and E.M. Vitale, Algebraic Theories, *Cambridge University Press*, Cambridge 2010. (p.3)

[2] R. Blackwell, G. M. Kelly, A. J. Power, Two-dimensional monad theory, *Journal of Pure and Applied Algebra*, Elsevier 1989. Available at https://www.sciencedirect.com/science/article/pii/0022404989901606. (p.3)

[3] M. Fiore and O. Mahmoud, Second-Order Algebraic Theories, *Mathematical Foundations of Computer Science*, Heidelberg 2010. (p.4)

[4] M. Fiore and C. K. Hur, Second-Order Equational Logic, *Computer Science Logic*, Springer 2010, pp.320-335. (p.4, 5)

[5] M. Fiore, D. Turi, and G. Plotkin, Abstract Syntax with Variable Binding, *Logic in Computer Science*, IEEE Computer Science Press, 1999. Available at https://ieeexplore.ieee.org/document/782615/. (p.4)

[6] M. Fiore and M. Hamana, Multiversal Polymorphic Algebraic Theories, *Logic in Computer Science*, 2013. Available at https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.590.5728. (p.4)

[7] M. Fiore, Second-Order and Dependently-Sorted Abstract Syntax, *Logic in Computer Science*, 200of8. Available at https://ieeexplore.ieee.org/document/4557900. (p.4)

[8] M. Fiore, N. Gambino, M. Hyland, and G. Winskel, Relative Pseudomonads, Kleisli Bicategories, and Substitution Monoidal Structures, *Selecta Mathematica* 24, 2017. Available at https://link.springer.com/article/10.1007/s00029-017-0361-3. (p.4)

[9] M. Fiore and S. Staton, Comparing Operational Models of Name-Passing Process Calculi, *Electronic Notes in Theoretical Computer Science* 106, Elsevier, 2004. (p.)

[10] B. Fong and D. Spivak, Graphical Regular Logic, 2018. Available at https://arxiv.org/abs/1812.05765. (p.6)

[11] M. Hyland and J. Power, The category theoretic understanding of universal algebra: Lawvere theories and monads, in *Electron. Notes Theor. Comput. Sci.* **172** (2007), 437–458. (p.3)

[12] B. Jacobs, Categorical Logic and Type Theory, *Studies in Logic and the Foundations of Mathematics* 141, Elsevier, 1999. Available at https://www.sciencedirect.com/bookseries/studies-in-logic-and-the-foundations-of-mathematics/vol/141. (p.2, 6, 9, 11)

[13] K Framework, http://www.kframework.org/index.php/Main_Page. (p.2)

[14] F. W. Lawvere, Functorial semantics of algebraic theories, reprinted in *Repr. Theory Appl. Categ.* **5** (2004), 1–121. Available at http://tac.mta.ca/tac/reprints/articles/5/tr5abs.html. (p.3)

[15] F. W. Lawvere, Adjointness in foundations, *Dialectica* 23, 1969. Available at https://www.emis.de//journals/TAC/reprints/articles/16/tr16abs.html. (p.6)

[16] J. Lambek and P. Scott, *Introduction to higher order categorical logic*, Cambridge Studies in Advanced Mathematics, 1986. (p.3)

[17] F. E. J. Linton, Some aspects of equational theories, *Proceedings of the Conference on Categorical Algebra*, eds. S. Eilenberg et al., Springer, Berlin, 1965. (p.3)

[18] S. Mac Lane and I. Moerdijk, *Sheaves in Geometry and Logic*, Springer-Verlag, New York 1992. (p.5)

[19] R. Milner, The polyadic π-calculus: a tutorial, in *Logic and Algebra of Specification*, Springer, Berlin, 1993, 203–246. (p.5)

[20] R. Milner, Communicating and Mobile Systems: The Pi Calculus, in *Cambridge University Press*, Cambridge, UK, 1999. (p.1)

[21] L.G. Meredith and M. Radestock, Namespace Logic: A logic for a reflective higher-order calculus, *Trustworthy Global Computing*, Edinburgh 2005. (p.1)

[22] L. G. Meredith and M. Radestock, A reflective higher-order calculus, *Electronic Notes in Theoretical Computer Science* **141** (2005), 49–67. (p.1, 4, 5)

[23] E. Moggi, Notions of computation and monads, *Logic in Computer Science*, IEEE, 1989, 55-92. Available at https://www.sciencedirect.com/science/article/pii/0890540191900524. (p.3)

[24] RChain Cooperative, https://www.rchain.coop/. (p.1)

[25] G. Roşu, Matching μ-Logic, *Logic in Computer Science* IEEE, 2019. (p.2)

[26] S. Staton, An algebraic presentation of predicate logic, *Foundations of Software Science and Computation Structures*, 2013, 401-417. Available at https://link.springer.com/chapter/10.1007/978-3-642-37075-5_26. (p.2)

[27] D. Turi and G. Plotkin, Towards a Mathematical Operational Semantics, *Logic in Computer Science*, IEEE, 1999. Available at https://ieeexplore.ieee.org/document/614955. (p.5)