Structural types for algebraic theories

Christian Williams
williams@math.ucr.edu

University of California, Riverside

MIT Categories
June 18, 2020

Introduction

Structural types are types built from term constructors and predicate logic.

Introduction

Structural types are types built from term constructors and predicate logic.

Example

The theory of monoids has the structural type

$$\mathrm{prime} := \neg[1] \wedge \neg[\neg[1] \cdot \neg[1]].$$

Introduction

Structural types are types built from term constructors and predicate logic.

Example

The theory of monoids has the structural type

$$\mathsf{prime} := \neg[1] \wedge \neg[\neg[1] \cdot \neg[1]].$$

We present a method of generating structural type theories, to provide languages with intrinsic type systems.

Introduction

The theoretical content is relatively simple. It's more about the motivation, and future applications and connections.

Introduction

The theoretical content is relatively simple. It's more about the motivation, and future applications and connections.

Related concepts

logic over a type theory, classifying toposes, logical relations, matching μ-logic, parameterized algebraic theories, hyperdoctrines...

Introduction

The theoretical content is relatively simple. It's more about the motivation, and future applications and connections.

Related concepts
logic over a type theory, classifying toposes, logical relations, matching μ-logic, parameterized algebraic theories, hyperdoctrines...

This is a fairly undeveloped introductory thesis, and it is to be implemented, so please let me know your thoughts.

Introduction

The theoretical content is relatively simple. It's more about the motivation, and future applications and connections.

Related concepts
logic over a type theory, classifying toposes, logical relations, matching μ-logic, parameterized algebraic theories, hyperdoctrines...

This is a fairly undeveloped introductory thesis, and it is to be implemented, so please let me know your thoughts.

The work was completely motivated by **Greg Meredith** and **Mike Stay**.

Contents

Motivation RChain

RChain is a distributed computing system based on a concurrent language called the **r**eflective **h**igher-**o**rder π-calculus, or ρ-calculus.

Motivation RChain

RChain is a distributed computing system based on a concurrent language called the **r**eflective **h**igher-**o**rder π-calculus, or ρ-calculus.

π-calculus

$$\lambda\text{-calculus} : \text{computer} :: \pi\text{-calculus} : \text{network}$$

Motivation RChain

RChain is a distributed computing system based on a concurrent language called the **r**eflective **h**igher-**o**rder π-calculus, or ρ-calculus.

π-calculus

$$\lambda\text{-calculus} : \text{computer} :: \pi\text{-calculus} : \text{network}$$

The π-calculus [3] is a *concurrent* language: a program is not a sequence of instructions, but a multiset of parallel processes. Computation is interactive, providing a unified language for both computers and networks.

Motivation RChain

RChain is a distributed computing system based on a concurrent language called the **r**eflective **h**igher-**o**rder π-calculus, or ρ-calculus.

π-calculus

$$\lambda\text{-calculus} : \text{computer} :: \pi\text{-calculus} : \text{network}$$

The π-calculus [3] is a *concurrent* language: a program is not a sequence of instructions, but a multiset of parallel processes. Computation is interactive, providing a unified language for both computers and networks. The basic rule is *communication*:

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

{[send on name $\mathtt{n}$ the name $\mathtt{a}$] in parallel with [recv on $\mathtt{n}$ then $\mathtt{p(x)}$]} evolves to {$\mathtt{p(a)}$}.

Motivation RChain

RChain is a distributed computing system based on a concurrent language called the **r**eflective **h**igher-**o**rder π-calculus, or ρ-calculus.

ρ-calculus

The ρ-calculus [5] adds *reflection*: operators which turn processes (code) into names (data) and vice versa.

$$\mathtt{@} : \mathtt{P} \rightleftharpoons \mathtt{N} : *$$

Motivation RChain

RChain is a distributed computing system based on a concurrent language called the **r**eflective **h**igher-**o**rder π-calculus, or ρ-calculus.

ρ-calculus

The ρ-calculus [5] adds *reflection*: operators which turn processes (code) into names (data) and vice versa.

$$\mathtt{@} : \mathrm{P} \rightleftharpoons \mathrm{N} : *$$

Communication is the transference of a program, rather than a pointer – this "code mobility" is of great practical utility and theoretical interest.

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

$\{$[out on name $\mathtt{n}$ the process q] in parallel with [in on $\mathtt{n}$ then $\mathtt{p(x)}$]$\}$ evolves to $\{\mathtt{p(@q)}\}$.

Motivation RChain

Names are both the data 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.

Motivation RChain

Names are both the data 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 [4] is a framework for this reasoning. The theory is

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

Motivation RChain

Names are both the data 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 [4] is a framework for this reasoning. The theory is

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

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.

Example

In concurrent computation it is useful to determine whether a process is single-threaded:

$$\text{single.thread} := \neg[0] \wedge \neg[\neg[0] \mid \neg[0]]$$

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

Motivation RChain

Example

In concurrent computation it is useful to determine whether a process is single-threaded:

$$\text{single.thread} := \neg[0] \wedge \neg[\neg[0] \mid \neg[0]]$$

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

Goal

How do we construct this logic *generally*, for any language?

Motivation RChain

Example

In concurrent computation it is useful to determine whether a process is single-threaded:

$$\text{single.thread} := \neg[0] \wedge \neg[\neg[0] \mid \neg[0]]$$

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

Goal

How do we construct this logic *generally*, for any language?

language	algebraic theory	T
logic	subobjects	$\mathcal{P} : \widehat{\mathrm{T}} \to \mathrm{Pos}.$

Algebraic Theories

Algebraic structures are presented by a set of sorts $\mathtt{s} \in \mathcal{T}$, operations $\mathtt{f} \in \mathcal{O}$, and equations $\mathtt{f}_1 = \mathtt{f}_2 \in \mathcal{E}$.

Algebraic Theories

Algebraic structures are presented by a set of sorts $\mathtt{s} \in \mathcal{T}$, operations $\mathtt{f} \in \mathcal{O}$, and equations $\mathtt{f}_1 = \mathtt{f}_2 \in \mathcal{E}$.

Example

A monoid is a structure of the form:

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

Algebraic Theories

Algebraic structures are presented by a set of sorts $\mathtt{s} \in \mathcal{T}$, operations $\mathtt{f} \in \mathcal{O}$, and equations $\mathtt{f}_1 = \mathtt{f}_2 \in \mathcal{E}$.

Example

A monoid is a structure of the form:

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

The *free cartesian category* on the presentation is the algebraic theory of monoids. Cartesian functors to a category C are “monoids in C”.

Algebraic Theories

Definition

Let X be the free cartesian category pseudomonad on Cat, and let Cart be the 2-category of cartesian categories. Define the 2-category of **multisorted algebraic theories** to be

$$\mathrm{AlgThy} := \int_{\mathrm{Set}} (\mathrm{X}(-)/\mathrm{Cart})_{\mathrm{idobj}}.$$

Algebraic Theories

Definition

Let X be the free cartesian category pseudomonad on Cat, and let Cart be the 2-category of cartesian categories. Define the 2-category of **multisorted algebraic theories** to be

$$\mathrm{AlgThy} := \int_{\mathrm{Set}} (\mathrm{X}(-)/\mathrm{Cart})_{\mathrm{idobj}}.$$

Hence a multisorted algebraic theory $(\mathcal{T}, \mathrm{T}, \tau)$ is a cartesian category T equipped with an identity-on-objects cartesian functor $\tau : \mathrm{X}(\mathcal{T}) \to \mathrm{T}$. The category of **models** of T in **context** C is $\mathrm{AlgThy}(\mathrm{T}, \mathrm{C})$.

Algebraic theories: Variable binding

While cartesian categories describe traditional algebraic structure, there is an important construction which they cannot express: *variable binding*.

$$\frac{\Gamma, x \vdash \; e(x)}{\Gamma \vdash \; \lambda x.e(x)} \, \lambda$$

Algebraic theories: Variable binding

While cartesian categories describe traditional algebraic structure, there is an important construction which they cannot express: *variable binding*.

$$\frac{\Gamma, x \vdash \; e(x)}{\Gamma \vdash \; \lambda x.e(x)} \, \lambda$$

To apply to programming languages, we need a theory which allows for binding operations and a definition of capture-avoiding substitution.

Algebraic theories: Variable binding

While cartesian categories describe traditional algebraic structure, there is an important construction which they cannot express: *variable binding*.

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

To apply to programming languages, we need a theory which allows for binding operations and a definition of capture-avoiding substitution.

In *Abstract Syntax and Variable Binding* [1], Fiore, Turi, and Plotkin describe operads with this structure, using the insight that exponentiating by representables is context extension.

$$\Lambda^{y(1)}(n) \simeq \mathrm{Set}^{\mathbb{F}}(\mathbb{F}(n,-) \times \mathbb{F}(1,-), \Lambda) \simeq \mathrm{Set}^{\mathbb{F}}(\mathbb{F}(n+1,-), \Lambda) \simeq \Lambda(n+1).$$

$$\lambda : \Lambda^{y(1)} \Rightarrow \Lambda$$

Algebraic theories: Variable binding

Definition

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 $u = ev_{\mathtt{s},\mathtt{t}} \circ (u^{\bullet} \times id_{\mathtt{s}})$.

Definition

A functor $F : \mathrm{T} \to \mathrm{C}$ **preserves** the exponentiable object $\mathtt{s}$ if $F(\mathtt{s})$ is exponentiable and for all $\mathtt{t}$,

$$ev^{\bullet}_{\mathtt{s},\mathtt{t}} : F(\mathtt{t}^{\mathtt{s}}) \simeq F(\mathtt{t})^{F(\mathtt{s})} \text{ and } F(ev_{\mathtt{s},\mathtt{t}}) = ev_{F(\mathtt{s}),F(\mathtt{t})}.$$

We call F **exponent** if it preserves all exponentiable objects.

Algebraic theories: Variable binding

There is a pseudomonad $\mathrm{E} : \mathrm{Cat} \to \mathrm{Cat}$ for the "free cartesian category on a set of exponentiable objects" construction.

Definition

Let Cart be the 2-category of cartesian categories. Define the 2-category of **second-order algebraic theories** to be

$$\mathrm{SOAT} := \int_{\mathrm{Set}} (\mathrm{E}(-)/\mathrm{Cart})_{\mathrm{idobj,exp}}.$$

Algebraic theories: Variable binding

There is a pseudomonad $\mathrm{E} : \mathrm{Cat} \to \mathrm{Cat}$ for the "free cartesian category on a set of exponentiable objects" construction.

Definition

Let Cart be the 2-category of cartesian categories. Define the 2-category of **second-order algebraic theories** to be

$$\mathrm{SOAT} := \int_{\mathrm{Set}} (\mathrm{E}(-)/\mathrm{Cart})_{\mathrm{idobj,exp}}.$$

A second-order algebraic theory $(\mathcal{T}, \mathrm{T}, \tau)$ is a cartesian category T with an identity-on-objects cartesian exponent functor $\tau : \mathrm{E}(\mathcal{T}) \to \mathrm{T}$. The category of **models** of T in **context** C is $\mathrm{SOAT}(\mathrm{T}, \mathrm{C})$.

Algebraic theories: ρ-calculus

ρ-calculus

$$
\begin{array}{lll}
\mathtt{0} : 1 \to \mathtt{P} & \quad \mid \; : \mathtt{P} \times \mathtt{P} & \to \mathtt{P} \\
\mathtt{@} : \mathtt{P} \to \mathtt{N} & \quad \mathtt{out} : \mathtt{N} \times \mathtt{P} & \to \mathtt{P} \\
* : \mathtt{N} \to \mathtt{P} & \quad \mathtt{in} : \mathtt{N} \times [\mathtt{N}, \mathtt{P}] & \to \mathtt{P}
\end{array}
$$

Algebraic theories: ρ-calculus

ρ-calculus

$$\begin{array}{lll} \mathtt{0} : 1 \to \mathtt{P} & \mid \; : \mathtt{P} \times \mathtt{P} & \to \mathtt{P} \\ \mathtt{@} : \mathtt{P} \to \mathtt{N} & \mathtt{out} : \mathtt{N} \times \mathtt{P} & \to \mathtt{P} \\ * : \mathtt{N} \to \mathtt{P} & \mathtt{in} : \mathtt{N} \times [\mathtt{N}, \mathtt{P}] & \to \mathtt{P} \end{array}$$

$$\mathtt{COMM} : \mathtt{out}(\mathtt{a}, \mathtt{q}) \mid \mathtt{in}(\mathtt{a}, \mathtt{x}.\mathtt{p}) = \mathtt{p}[\mathtt{@q}/\mathtt{x}]$$

$$\mathtt{EVAL} : *\mathtt{@}(\mathtt{p}) = \mathtt{p}$$

$$(\mathtt{P}, \mid, \mathtt{0}) \quad \mathtt{commutative\ monoid}$$

Algebraic theories: ρ-calculus

ρ-calculus

$$\begin{aligned} \mathtt{0} &: 1 \to \mathtt{P} & \mid \; &: \mathtt{P} \times \mathtt{P} && \to \mathtt{P} \\ \mathtt{@} &: \mathtt{P} \to \mathtt{N} & \mathtt{out} &: \mathtt{N} \times \mathtt{P} && \to \mathtt{P} \\ * &: \mathtt{N} \to \mathtt{P} & \mathtt{in} &: \mathtt{N} \times [\mathtt{N}, \mathtt{P}] && \to \mathtt{P} \end{aligned}$$

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

$$\mathtt{EVAL} : *\mathtt{@}(\mathrm{p}) \Rightarrow \mathrm{p}$$

$$(\mathtt{P}, \mid, \mathtt{0}) \quad \texttt{commutative monoid}$$

Presheaves and subobjects

A **presheaf** on a category T is a functor $A : \mathrm{T}^{\mathrm{op}} \to \mathrm{Set}$. We can understand A as data on the objects of T.

Presheaves and subobjects

A **presheaf** on a category T is a functor $A : \mathrm{T}^{\mathrm{op}} \to \mathrm{Set}$. We can understand A as data on the objects of T.

Definition

The **Yoneda embedding** $y : \mathrm{T} \to [\mathrm{T}^{\mathrm{op}}, \mathrm{Set}]$ sends $\mathtt{s} \mapsto \mathrm{T}(-, \mathtt{s})$ and $(\mathtt{f} : \mathtt{s} \to \mathtt{t}) \mapsto (\mathtt{f} \circ - : \mathrm{T}(-, \mathtt{s}) \Rightarrow \mathrm{T}(-, \mathtt{t}))$.

Presheaves and subobjects

A **presheaf** on a category T is a functor $A : \mathrm{T}^{\mathrm{op}} \to \mathrm{Set}$. We can understand A as data on the objects of T.

Definition

The **Yoneda embedding** $y : \mathrm{T} \to [\mathrm{T}^{\mathrm{op}}, \mathrm{Set}]$ sends $\mathtt{s} \mapsto \mathrm{T}(-, \mathtt{s})$ and $(\mathtt{f} : \mathtt{s} \to \mathtt{t}) \mapsto (\mathtt{f} \circ - : \mathrm{T}(-, \mathtt{s}) \Rightarrow \mathrm{T}(-, \mathtt{t}))$.

If T is a theory, then $\mathrm{T}(-, \mathtt{s})$ assigns the data

$$\mathtt{r} \mapsto \{\text{terms } \mathtt{r} \to \mathtt{s}\}.$$

Presheaves and subobjects

The category of presheaves $\widehat{\mathrm{T}} := [\mathrm{T}^{\mathrm{op}}, \mathrm{Set}]$ is a *topos*, a category with rich internal logic: finite limits are given pointwise, exponentials are given by

$$[P, Q](\mathtt{s}) = \widehat{\mathrm{T}}(\mathrm{T}(-, \mathtt{s}) \times P, Q)$$

and the subobject classifer is given by

$$\Omega(\mathtt{s}) = \{P \mid P \rightarrowtail \mathrm{T}(-, \mathtt{s})\}.$$

Presheaves and subobjects

The category of presheaves $\widehat{\mathrm{T}} := [\mathrm{T}^{\mathrm{op}}, \mathrm{Set}]$ is a *topos*, a category with rich internal logic: finite limits are given pointwise, exponentials are given by

$$[P, Q](\mathtt{s}) = \widehat{\mathrm{T}}(\mathrm{T}(-, \mathtt{s}) \times P, Q)$$

and the subobject classifer is given by

$$\Omega(\mathtt{s}) = \{P \mid P \rightarrowtail \mathrm{T}(-, \mathtt{s})\}.$$

We can interpret subobjects as predicates, and use logical constructors.

Lemma

For any presheaf $A : \mathrm{T}^{\mathrm{op}} \to \mathrm{Set}$, subfunctors form a Heyting algebra $\mathcal{P}(A)$. They are ordered by inclusion, with meet and join defined by pointwise intersection and union. Implication is defined:

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

Presheaves and subobjects: Quantifiers

The subobject functor is a *hyperdoctrine*:

Lemma

Let $\widehat{\mathrm{T}}$ be a topos, and $\mathcal{P} : \mathrm{T}^{\mathrm{op}} \to \mathrm{Pos}$ be the subobject functor. For each $\mathtt{f} : A \to B$ there is a triple adjunction $\exists_{\mathtt{f}} \dashv \mathcal{P}(\mathtt{f}) \dashv \forall_{\mathtt{f}}$ which satisfies that quantification commutes with substitution and $\forall_{\mathtt{f}}$ preserves implication.

Presheaves and subobjects: Quantifiers

The subobject functor is a *hyperdoctrine*:

Lemma

Let $\widehat{\mathrm{T}}$ be a topos, and $\mathcal{P} : \mathrm{T}^{\mathrm{op}} \to \mathrm{Pos}$ be the subobject functor. For each $\mathtt{f} : A \to B$ there is a triple adjunction $\exists_{\mathtt{f}} \dashv \mathcal{P}(\mathtt{f}) \dashv \forall_{\mathtt{f}}$ which satisfies that quantification commutes with substitution and $\forall_{\mathtt{f}}$ preserves implication.

In Set, $\mathcal{P}(\mathtt{f})$ is preimage, $\exists_{\mathtt{f}}$ is *direct image*, and $\forall_{\mathtt{f}}$ is *saturated image*.

In a presheaf topos $\widehat{\mathrm{T}}$ these are defined for $\varphi \rightarrowtail A$:

$$\exists_{\mathtt{f}}(\varphi)(\mathtt{t}) = \{b \in B(\mathtt{t}) \mid \exists u : \mathtt{s} \to \mathtt{t}.\ \exists a \in A.\ b \cdot u = \mathtt{f}_{\mathtt{s}}(a) \wedge a \in \varphi(\mathtt{s})\}$$
$$\forall_{\mathtt{f}}(\varphi)(\mathtt{t}) = \{b \in B(\mathtt{t}) \mid \forall u : \mathtt{s} \to \mathtt{t}.\ \forall a \in A.\ b \cdot u = \mathtt{f}_{\mathtt{s}}(a) \Rightarrow a \in \varphi(\mathtt{s})\}.$$

Presheaves and subobjects: Sieves

Let T be a category and $\mathtt{s}$ be an object. A **sieve** on $\mathtt{s}$ is a class of morphisms into $\mathtt{s}$ which is closed under precomposition.

Presheaves and subobjects: Sieves

Let T be a category and s be an object. A **sieve** on s is a class of morphisms into s which is closed under precomposition.

Lemma

There is a natural bijection of sieves on s *and subfunctors of* $\mathrm{T}(-,\mathrm{s})$.

Presheaves and subobjects: Sieves

Let T be a category and $\mathtt{s}$ be an object. A **sieve** on $\mathtt{s}$ is a class of morphisms into $\mathtt{s}$ which is closed under precomposition.

Lemma

There is a natural bijection of sieves on $\mathtt{s}$ and subfunctors of $\mathrm{T}(-,\mathtt{s})$.

Example

Any morphism in T generates a *singleton sieve*:

$$\mathtt{f} : \mathtt{r} \to \mathtt{s} \quad \mapsto \quad S(\mathtt{f}) \rightarrowtail \mathrm{T}(-,\mathtt{s})$$

$$S(\mathtt{f})(\mathrm{a}) := \{\mathtt{op} : \mathtt{a} \to \mathtt{s} \mid \exists u : \mathtt{a} \to \mathtt{s}.\ \mathtt{op} = \mathtt{f} \circ u\}.$$

Presheaves and subobjects: Lax structure

We henceforth use $\mathcal{P} : \widehat{\mathrm{T}} \to \mathrm{Pos}$ to denote $\mathcal{P}(\mathtt{f}) := \exists_{\mathtt{f}}$.

Presheaves and subobjects: Lax structure

We henceforth use $\mathcal{P} : \widehat{\mathrm{T}} \to \mathrm{Pos}$ to denote $\mathcal{P}(\mathtt{f}) := \exists_{\mathtt{f}}$.

We are interested in the lax cartesian structure of $\mathcal{P}$, denoted $\sqcap$, so that we can lift operations

$$\mathtt{f} : \mathtt{a} \times \mathtt{b} \to \mathtt{c}$$

$$\bar{\mathtt{f}} : \mathcal{P}(y(\mathtt{a})) \times \mathcal{P}(y(\mathtt{b})) \to \mathcal{P}(y(\mathtt{a} \times \mathtt{b})) \to \mathcal{P}(y(\mathtt{c})).$$

Presheaves and subobjects: Lax structure

We henceforth use $\mathcal{P} : \widehat{\mathrm{T}} \to \mathrm{Pos}$ to denote $\mathcal{P}(\mathtt{f}) := \exists_{\mathtt{f}}$.

We are interested in the lax cartesian structure of $\mathcal{P}$, denoted $\sqcap$, so that we can lift operations

$$\mathtt{f} : \mathtt{a} \times \mathtt{b} \to \mathtt{c}$$

$$\bar{\mathtt{f}} : \mathcal{P}(y(\mathtt{a})) \times \mathcal{P}(y(\mathtt{b})) \to \mathcal{P}(y(\mathtt{a} \times \mathtt{b})) \to \mathcal{P}(y(\mathtt{c})).$$

Lemma

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

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

Then $(\mathcal{P}, \sqcap)$ *is lax cartesian, and* $\lambda \dashv \sqcap$.

Presheaves and subobjects: Lax structure

While it is well-known that $\mathcal{P}$ is "adjoint-lax" with respect to products, it is also adjoint-lax with respect to closed structure.

Definition

The lax structure $(\mathcal{P}, \sqcap)$ induces a lax *closed* structure $(\mathcal{P}, \Lambda)$, given by the currying of evaluation:

$$\Lambda_{AB} : \mathcal{P}([A, B]) \to [\mathcal{P}(A), \mathcal{P}(B)]$$

$$\Lambda_{AB}(X)(S)(\mathtt{c}) = \{ev(\mathtt{f}, \mathtt{a}) \mid \mathtt{f} \in X(\mathtt{c}), \mathtt{a} \in S(\mathtt{c})\}.$$

Presheaves and subobjects: Lax structure

While it is well-known that $\mathcal{P}$ is "adjoint-lax" with respect to products, it is also adjoint-lax with respect to closed structure.

Definition

The lax structure $(\mathcal{P}, \sqcap)$ induces a lax *closed* structure $(\mathcal{P}, \Lambda)$, given by the currying of evaluation:

$$\Lambda_{AB} : \mathcal{P}([A, B]) \to [\mathcal{P}(A), \mathcal{P}(B)]$$

$$\Lambda_{AB}(X)(S)(\mathtt{c}) = \{ev(\mathtt{f}, \mathtt{a}) \mid \mathtt{f} \in X(\mathtt{c}), \mathtt{a} \in S(\mathtt{c})\}.$$

Puzzle

Given a function $\mathtt{f} : [A, B] \to C$, how can we define a lifting

$$\bar{\mathtt{f}} : [\mathcal{P}(A), \mathcal{P}(B)] \to \mathcal{P}(C)?$$

Presheaves and subobjects: Lax structure

Definition

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

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

Presheaves and subobjects: Lax structure

Definition

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

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

Lemma

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

$$\mathrm{R}_{AB} : [\mathcal{P}(A), \mathcal{P}(B)] \to \mathcal{P}([A, B])$$

$$\mathrm{R}_{AB}(F)(\mathtt{c}) = \{f \in [A, B](\mathtt{c}) \mid f \textit{ respects } F\}.$$

Then $(\mathcal{P}, \mathrm{R})$ *is colax closed, and* $\Lambda \dashv \mathrm{R}$.

Structural types: Lifting

Definition

Let T be a second-order algebraic theory, and let $\mathtt{f} : \prod_{i=1}^{n}[\prod_{j=1}^{n_i} \mathtt{s}_{ij}, \mathtt{t}_i] \to \mathtt{t}$ be an operation.

Structural types: Lifting

Definition

Let T be a second-order algebraic theory, and let $\mathtt{f} : \prod_{i=1}^{n}[\prod_{j=1}^{n_i} \mathtt{s}_{ij}, \mathtt{t}_i] \to \mathtt{t}$ be an operation.

Define the **lifting** of $\mathtt{f}$ to be

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

Structural types: Lifting

Definition

Let T be a second-order algebraic theory, and let $\mathtt{f} : \prod_{i=1}^{n}[\prod_{j=1}^{n_i} \mathtt{s}_{ij}, \mathtt{t}_i] \to \mathtt{t}$ be an operation.

Define the **lifting** of $\mathtt{f}$ to be

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

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

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

Structural types: Lifting

Theorem

Let $(\mathcal{T}, \mathrm{T}, \tau)$ *be a second-order algebraic theory.*
The lifting 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.

Structural types: Structural theory

Definition

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

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

Structural types: Structural theory

Definition

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

$$\omega_{\mathrm{T}}(\mathtt{g}) = \langle \overline{\mathtt{g}_1^{\circ}}^{\bullet}, \ldots, \overline{\mathtt{g}_n^{\circ}}^{\bullet} \rangle.$$

Structural types: Structural theory

Definition

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 $\omega_{\mathrm{T}}(\mathtt{s}) =: \mathtt{s}_{\omega}$.

Structural types: Structural theory

Definition

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

The map $(\mathrm{T} \mapsto \omega_{\mathrm{T}})$ *defines a 2-functor*

$$\omega : \mathrm{SOAT} \to (\iota \downarrow \mathrm{Pos})$$

where the latter is the comma 2-category of the inclusion $\iota : \mathrm{SOAT} \rightarrowtail 2\mathrm{Cat}_{\mathrm{colax}}$ *and the constant* $\mathrm{Pos} : 1 \to 2\mathrm{Cat}_{\mathrm{colax}}$.

Structural types: ρ-calculus

Let T be the theory of the ρ-calculus. The lifted signature provides the algebraic type constructors of namespace logic.

Definition

The ωρ-**calculus** has algebraic type constructors:

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

Structural types: ρ-calculus

Let T be the theory of the ρ-calculus. The lifted signature provides the algebraic type constructors of namespace logic.

Definition

The ωρ-**calculus** has algebraic type constructors:

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

We can now construct the type of single-threaded processes:

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

Structural types: ρ-calculus

By lifting the binding operation $\mathtt{in} : \mathtt{N} \times [\mathtt{N}, \mathtt{P}] \to \mathtt{P}$, we gain significant expressiveness. Just as input binds a free name variable,

$$\overline{\mathtt{in}} : \mathtt{N}_\omega \times [\mathtt{N}_\omega, \mathtt{P}_\omega] \to \mathtt{P}_\omega$$

binds a free "namespace" variable $\Phi : \mathtt{N}_\omega \to \mathtt{P}_\omega$.

Structural types: ρ-calculus

By lifting the binding operation $\mathtt{in} : \mathtt{N} \times [\mathtt{N}, \mathtt{P}] \to \mathtt{P}$, we gain significant expressiveness. Just as input binds a free name variable,

$$\overline{\mathtt{in}} : \mathtt{N}_\omega \times [\mathtt{N}_\omega, \mathtt{P}_\omega] \to \mathtt{P}_\omega$$

binds a free "namespace" variable $\Phi : \mathtt{N}_\omega \to \mathtt{P}_\omega$.

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 $\Phi(\chi)$.

Structural types: ρ-calculus

By lifting the binding operation $\mathtt{in} : \mathtt{N} \times [\mathtt{N}, \mathtt{P}] \to \mathtt{P}$, we gain significant expressiveness. Just as input binds a free name variable,

$$\overline{\mathtt{in}} : \mathtt{N}_\omega \times [\mathtt{N}_\omega, \mathtt{P}_\omega] \to \mathtt{P}_\omega$$

binds a free "namespace" variable $\Phi : \mathtt{N}_\omega \to \mathtt{P}_\omega$.

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.

Structural types: Colaxity

Note The colaxity of ω_{T} is inevitable when lifting binding operations.

Structural types: Colaxity

Note The colaxity of ω_{T} is inevitable when lifting binding operations. For example, consider the ρ-calculus term

$$\mathtt{in}(-, \mathtt{x}.\mathtt{out}(\mathtt{x}, -)) : \mathtt{N} \times \mathtt{P} \to \mathtt{P}.$$

Structural types: Colaxity

Note The colaxity of ω_{T} is inevitable when lifting binding operations. For example, consider the ρ-calculus term

$$\mathtt{in}(-, \mathtt{x}.\mathtt{out}(\mathtt{x}, -)) : \mathtt{N} \times \mathtt{P} \to \mathtt{P}.$$

Then

$$\overline{\mathtt{in}}(\alpha, \overline{\mathtt{out}^{\bullet}}(\varphi)) = \overline{\mathtt{in}}(\alpha, \beta.\overline{\mathtt{out}}(\beta, \varphi)),$$

Structural types: Colaxity

Note The colaxity of ω_{T} is inevitable when lifting binding operations. For example, consider the ρ-calculus term

$$\mathtt{in}(-, \mathtt{x}.\mathtt{out}(\mathtt{x}, -)) : \mathtt{N} \times \mathtt{P} \to \mathtt{P}.$$

Then

$$\overline{\mathtt{in}}(\alpha, \overline{\mathtt{out}^{\bullet}}(\varphi)) = \overline{\mathtt{in}}(\alpha, \beta.\overline{\mathtt{out}}(\beta, \varphi)),$$

while

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

Structural types: Colaxity

Note The colaxity of ω_{T} is inevitable when lifting binding operations. For example, consider the ρ-calculus term

$$\mathtt{in}(-, \mathtt{x}.\mathtt{out}(\mathtt{x}, -)) : \mathtt{N} \times \mathtt{P} \to \mathtt{P}.$$

Then

$$\overline{\mathtt{in}}(\alpha, \overline{\mathtt{out}^{\bullet}}(\varphi)) = \overline{\mathtt{in}}(\alpha, \beta.\overline{\mathtt{out}}(\beta, \varphi)),$$

while

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

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

Application Namespace Logic

Starting from the practical end of things, whether we consider MAC addresses, IP addresses, domain names or URL's it is clear that distributed computing is practiced, today, using names. Moreover, it is essential to the programs that administer as well as to the ones that compute over this distributed computing infrastructure that these names have structure.

Thus, when we look to theory, especially a theory like the π-calculus, of computing based on interaction over named channels, to help us with this practice some story must be told about how the struture of these names contributes to interaction and computation over (channels named by) them. [4]

Application Namespace Logic

We have built all of the tools used in namespace logic, except for one: *fixed point type constructors*.

Definition

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 and colimit of Φ exist, denoted

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

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

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

Application Namespace Logic

Example

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

$$\text{Liveness : } S \text{ can always communicate on } \alpha.$$
$$\text{Safety : } S \text{ can never communicate on } \neg\alpha.$$

Application Namespace Logic

Example

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

$$\text{Liveness}: \ S \text{ can always communicate on } \alpha.$$
$$\text{Safety}: \ S \text{ can never communicate on } \neg\alpha.$$

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, this is a *compile-time firewall*: a process satisfies this predicate if and only if it can always input on a name in α, and it can never input on a name in $\neg\alpha$.

The type theory

Kinds The sorts $\mathtt{s} \in \mathrm{T}$, products in T and exponents by $\mathtt{t} \in \mathcal{T}$.
Types The objects of $\int \omega_{\mathrm{T}}$ 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 following inference is the introduction rule for terms in $\omega\mathrm{T}$; the polymorphism of operations with respect to their liftings is automatic from the construction of ω_{T}.

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

Thanks

Thanks for listening.

Structural types for algebraic theories
C. Williams, May 2020
github:cbw124/stat.pdf

References

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/.

Marcelo Fiore and Ola Mahmoud, *Second-Order Algebraic Theories*, Mathematical Foundations of Computer Science, Heidelberg 2010.

R. Milner, Communicating and Mobile Systems: The Pi Calculus, in *Cambridge University Press*, Cambridge, UK, 1999.

L.G. Meredith and Matthias Radestock, *Namespace Logic: A logic for a reflective higher-order calculus*, Trustworthy Global Computing, Edinburgh 2005.

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