Native Type Theory

Christian Williams cwill041@ucr.edu
Mike Stay stay@pyrofex.net

ACT 2021

Background

I came to grad school wanting to apply category theory to blockchain, or the movement toward a distributed internet.

Background

I came to grad school wanting to apply category theory to blockchain, or the movement toward a distributed internet.

John Baez connected me with Statebox, which is developing languages and software based on category theory.

Background

I came to grad school wanting to apply category theory to blockchain, or the movement toward a distributed internet.

John Baez connected me with Statebox, which is developing languages and software based on category theory.

This led to a collaboration with Mike Stay, and Greg Meredith at RChain. They had been looking for ways to *generate logics for languages*.

Background

I came to grad school wanting to apply category theory to blockchain, or the movement toward a distributed internet.

John Baez connected me with Statebox, which is developing languages and software based on category theory.

This led to a collaboration with Mike Stay, and Greg Meredith at RChain. They had been looking for ways to *generate logics for languages*.

I talked with them and struggled with this question for a long time. In retrospect, the solution was much simpler than we thought.

Background

I came to grad school wanting to apply category theory to blockchain, or the movement toward a distributed internet.

John Baez connected me with Statebox, which is developing languages and software based on category theory.

This led to a collaboration with Mike Stay, and Greg Meredith at RChain. They had been looking for ways to *generate logics for languages*.

I talked with them and struggled with this question for a long time. In retrospect, the solution was much simpler than we thought.

Now this topic is my thesis direction. I am happy that the idea is simple, because I think its application can have a real impact.

Native Type Theory

The whole idea: two basic facts of category theory **compose**.

The whole idea: two basic facts of category theory **compose**.

> Every category embeds into a topos.

Native Type Theory

The whole idea: two basic facts of category theory **compose**.

Every category embeds into a topos.

Every topos has a rich internal language.

Native Type Theory

The whole idea: two basic facts of category theory **compose**.

Every category embeds into a topos.

Every topos has a rich internal language.

Native Type Theory simply gives a name to *the language of presheaves*, and advocates for *real-world application of internal logic*.

Native Type Theory

The whole idea: two basic facts of category theory **compose**.

Every category embeds into a topos.

Every topos has a rich internal language.

Native Type Theory simply gives a name to *the language of presheaves*, and advocates for *real-world application of internal logic*.

These facts are well-known, but some aspects have less public awareness.

The embedding is **continuous** and **monoidal closed**.

Native Type Theory

The whole idea: two basic facts of category theory **compose**.

Every category embeds into a topos.

Every topos has a rich internal language.

Native Type Theory simply gives a name to *the language of presheaves*, and advocates for *real-world application of internal logic*.

These facts are well-known, but some aspects have less public awareness.

The embedding is **continuous** and **monoidal closed**.

The language of a topos is more than just a syntax; it is a structured **fibration**, and this construction is **2-functorial**.

Motivation: Programming Languages

Type theory is growing as a guiding philosophy in the design of programming languages. But in practice, many popular languages do not have well-structured type systems.

Motivation: Programming Languages

Type theory is growing as a guiding philosophy in the design of programming languages. But in practice, many popular languages do not have well-structured type systems.

Ideally, there ought to be a way for a language to *generate* a type system. Categorical logic provides a method to generate a **native type system** for reasoning about the structure and behavior of programs.

Motivation: Programming Languages

Type theory is growing as a guiding philosophy in the design of programming languages. But in practice, many popular languages do not have well-structured type systems.

Ideally, there ought to be a way for a language to *generate* a type system. Categorical logic provides a method to generate a **native type system** for reasoning about the structure and behavior of programs.

Theorem (W., Stay)

There is a 2-functor

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

Hence, translations of languages induce translations of native type systems. If implemented well, this could provide a unified framework of reasoning for everyday programming.

λ-theories

The language of cartesian closed categories is *simply-typed λ-calculus*.

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

λ-theories

The language of cartesian closed categories is *simply-typed λ-calculus*.

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

Definition

A **$\boldsymbol{\lambda}$-theory with equality** is a cartesian closed category with pullbacks. The 2-category of λ-theories with equality, finitely continuous closed functors, and cartesian natural transformations is $\lambda\mathrm{Thy}_=$.

We interpret the language as simply-typed λ-calculus combined with the syntax of *generalized algebraic theories*, which provide *indexed sorts*.

$$\frac{\Gamma \vdash x_1 : \mathtt{S}_1, \ldots, x_n : \mathtt{S}_n}{\Gamma, \vec{x_i} : \vec{\mathtt{S}_i} \vdash \mathtt{A}(x_1, \ldots, x_n)\ \text{ sort}}$$

ρ-calculus

The ρ-calculus or **r**eflective **h**igher-**o**rder π-calculus is a concurrent language which refines the π-calculus. It is the language of the blockchain platform RChain.

ρ-calculus

The ρ-calculus or **r**eflective **h**igher-**o**rder π-calculus is a concurrent language which refines the π-calculus. It is the language of the blockchain platform RChain.

The language is represented by the free λ-theory with equality on the following presentation.

ρ-calculus

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

$$
\begin{array}{rl}
\mathtt{comm}(n, q, \lambda x.p): & \mathtt{out}(n,q)|\mathtt{in}(n,\lambda x.p) \rightsquigarrow p[\mathtt{@}q/x] \\
\mathtt{run}(p): & *(\mathtt{@}p) \rightsquigarrow p
\end{array}
$$

Language of a topos

The Yoneda embedding $y : \mathsf{T} \to [\mathsf{T}^{\mathsf{op}}, \mathrm{Set}]$ sends $\mathtt{S}$ to $\mathsf{T}(-, \mathtt{S})$. This preserves limits and homs, and embeds T into a *presheaf topos*.

Definition

A **topos** is a λ-theory with equality $\mathcal{E}$ with $\mathcal{E}(-, \Omega) \simeq \mathrm{Sub}(-)$.

For presheaves, the subobject classifier is defined $\Omega(\mathtt{S}) = \{\varphi \rightarrowtail y(\mathtt{S})\}$. It is an internal complete Heyting algebra.

Language of a topos

The Yoneda embedding $y : \mathsf{T} \to [\mathsf{T}^{\mathsf{op}}, \mathrm{Set}]$ sends $\mathtt{S}$ to $\mathsf{T}(-, \mathtt{S})$. This preserves limits and homs, and embeds T into a *presheaf topos*.

Definition

A **topos** is a λ-theory with equality $\mathcal{E}$ with $\mathcal{E}(-, \Omega) \simeq \mathrm{Sub}(-)$.

For presheaves, the subobject classifier is defined $\Omega(\mathtt{S}) = \{\varphi \rightarrowtail y(\mathtt{S})\}$. It is an internal complete Heyting algebra.

Definition

The **predicate functor** of a topos $\mathcal{E}$ defined $[-, \Omega] : \mathcal{E}^{\mathsf{op}} \to \mathrm{CHA}$ gives a higher-order fibration $\pi_\Omega : \Omega\mathcal{E} \to \mathcal{E}$. This means for each $f : A \to B$, the functor $\Omega^f : \Omega^B \to \Omega^A$ has adjoints $\exists_f \dashv \Omega^f \dashv \forall_f$ (satisfying BC).

These can be understood as **direct image**, **preimage**, and **secure image**.

Language of a topos

Using these operations, we can construct highly expressive predicates on the structure of terms in a language T.

Example

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

Language of a topos

Using these operations, we can construct highly expressive predicates on the structure of terms in a language T.

Example

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

Example

For a ρ-calculus predicate $\varphi : y(\mathtt{P}) \to \Omega$, preimage by input is the query "inputting on what name-context pairs yield property φ?"

$$\varphi[\mathtt{in}] := [y(\mathtt{in}), \Omega](\varphi) : y(\mathtt{N} \times [\mathtt{N} \to \mathtt{P}]) \to \Omega$$

$$\varphi[\mathtt{in}](\mathtt{S})(n, \lambda x.p) = \varphi(\mathtt{S})(\mathtt{in}(n, \lambda x.p))$$

Language of a topos

Using these operations, we can construct highly expressive predicates on the structure of terms in a language T.

Example

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

Example

For a ρ-calculus predicate $\varphi : y(\mathtt{P}) \to \Omega$, preimage by input is the query "inputting on what name-context pairs yield property φ?"

$$\varphi[\mathtt{in}] := [y(\mathtt{in}), \Omega](\varphi) : y(\mathtt{N} \times [\mathtt{N} \to \mathtt{P}]) \to \Omega$$

$$\varphi[\mathtt{in}](\mathtt{S})(n, \lambda x.p) = \varphi(\mathtt{S})(\mathtt{in}(n, \lambda x.p))$$

Example

direct-step $\exists_t \Omega^s$ and secure-step $\forall_t \Omega^s$

Functoriality

Predicates $\varphi : A \to \Omega$ correspond to subobjects $\mathrm{c}(\varphi) \rightarrowtail A$. More generally, any $p : P \to A$ can be understood as a *dependent type*. The predicate fibration π_Ω embeds into the *codomain fibration* π_Δ.

The two fibrations are connected by the image-comprehension adjunction. All together, this forms a *higher-order dependent type theory*.

Functoriality

Predicates $\varphi : A \to \Omega$ correspond to subobjects $\mathrm{c}(\varphi) \rightarrowtail A$. More generally, any $p : P \to A$ can be understood as a *dependent type*. The predicate fibration π_Ω embeds into the *codomain fibration* π_Δ.

The two fibrations are connected by the image-comprehension adjunction. All together, this forms a *higher-order dependent type theory*.

Theorem (W., Stay)

The construction which sends a topos to its **internal language** $\mathcal{L}(\mathcal{E}) = \langle \pi_{\Omega\mathcal{E}}, \pi_{\Delta\mathcal{E}}, \mathrm{i}_\mathcal{E}, \mathrm{c}_\mathcal{E} \rangle$ *defines a 2-functor* $\mathcal{L} : \mathrm{Topos} \to \mathrm{HDT\Sigma}$.

There are many questions about this functoriality of both theoretical and practical importance.

Applications: behavior

In a concurrent language like the ρ-calculus, the basic rule is *communication*.

$$\mathtt{comm}(n, q, \lambda x.p) : \mathtt{out}(n, q)|\mathtt{in}(n, \lambda x.p) \rightsquigarrow p[\mathtt{@}q/x]$$

The graph of rewrites is the space of all computations.

$$\mathrm{g}(\mathtt{S})(p_1, p_2) = \{e \mid \mathtt{S} \vdash e : p_1 \rightsquigarrow p_2\}$$

Applications: behavior

In a concurrent language like the ρ-calculus, the basic rule is *communication*.

$$\texttt{comm}(n, q, \lambda x.p) : \texttt{out}(n, q)|\texttt{in}(n, \lambda x.p) \rightsquigarrow p[@q/x]$$

The graph of rewrites is the space of all computations.

$$\mathrm{g}(\mathtt{S})(p_1, p_2) = \{e \mid \mathtt{S} \vdash e : p_1 \rightsquigarrow p_2\}$$

We can filter to subspaces: the type of communications on channels in α, sending data in ψ, and continuing in contexts $\lambda x.c : [\mathtt{N}, \mathtt{P}]$ such that $\chi(n) \Rightarrow F(\chi)(c[n/x])$ can be constructed as a native type.

$$\Sigma e{:}\texttt{comm}(\alpha, \varphi, \chi.F).\mathrm{g}$$

We can then construct modalities relative to these subspaces, as well as behavioral equivalences.

Applications: refined binding

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

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

The **refinement** of the ρ-calculus is the subtheory in which the only rewrite constructors are $\mathtt{comm}_\alpha$ for each namespace.

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

Applications: predicate hom

Given $\varphi : A \to \text{Prop}$ and $\psi : B \to \text{Prop}$, the **predicate hom** is defined

$$[\varphi, \psi] : [A, B] \to \text{Prop}$$

$$[\varphi, \psi](f) = \forall a{:}A \;\; \varphi(a) \Rightarrow \psi(f(a))$$

Example

We can detect security leaks: given a trusted channel $a : \mathtt{N}$ and an untrusted $n : \mathtt{N}$, then the following program will not preserve safety on a.

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

We can also detect if a program may not remain single-threaded:

$$\mathtt{out}(a, (- \mid q)) : \text{single.thread} \rhd_{\text{act}} \neg[\text{s.thread}]$$

where $\rhd_{\text{act}}$ is the arrow relative to the observational transition system.

Going forward: join us!

Two main kinds of application:

- Debug, condition, and query existing codebases.
- Expand software capability with native types.

The tools necessary for implementation already exist.
Contact us: cwill041@ucr.edu, stay@pyrofex.net.

Thank you!

C. Williams and M. Stay, *Native Type Theory*. arXiv:2102.04672