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

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.
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.
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*

\[ \lambda \text{Thy}^{\text{op}} \xrightarrow{\mathcal{P}} \text{Topos} \xrightarrow{\mathcal{L}} \text{HDT}\Sigma \]

Hence, translations of languages induce translations of native type systems. If implemented well, this could provide a unified framework of reasoning for everyday programming.
The language of cartesian closed categories is *simply-typed* $\lambda$-*calculus*.

\[
\frac{\Gamma, x:S \vdash t : T}{\Gamma \vdash \lambda x.t : [S \rightarrow T]} \quad \text{abstraction} \\
\frac{\Gamma \vdash \lambda x.t : [S \rightarrow T], u : S}{\Gamma \vdash t[u/x] : T} \quad \text{application}
\]
The language of cartesian closed categories is *simply-typed λ-calculus*.

\[
\Gamma, x:S \vdash t : T \\
\Gamma \vdash \lambda x.t : [S \to T] \\
\Gamma \vdash t[u/x] : T
\]

**Definition**

A **λ-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 \(λ\text{Thy}_\equiv\).

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

\[
\Gamma \vdash x_1 : S_1, \ldots, x_n : S_n \\
\Gamma, \bar{x}_i : \bar{S}_i \vdash A(x_1, \ldots, x_n) \text{ sort}
\]
The \( \rho \)-calculus or reflective higher-order \( \pi \)-calculus is a concurrent language which refines the \( \pi \)-calculus. It is the language of the blockchain platform RChain.
The $\rho$-calculus or reflective higher-order $\pi$-calculus is a concurrent language which refines the $\pi$-calculus. It is the language of the blockchain platform RChain.

The language is represented by the free $\lambda$-theory with equality on the following presentation.

\begin{align*}
0 : & \ 1 \to P \\
\oplus : & \ P \to N \\
\otimes : & \ N \to P \\
\circ : & \ N \to [N \to P] \to P \\
\text{comm} : & \ N, P, [N \to P] \to E
\end{align*}

\begin{align*}
\text{comm}(n, q, \lambda x.p) : & \ \text{out}(n, q)\mid\text{in}(n, \lambda x.p) \leadsto p[@q/x] \\
\text{run}(p) : & \ \circ(@p) \leadsto p
\end{align*}
The Yoneda embedding $y : T \to [T^{op}, \text{Set}]$ sends $S$ to $T(-, S)$. This preserves limits and homs, and embeds $T$ into a \textit{presheaf topos}.

\textbf{Definition}

A \textbf{topos} is a $\lambda$-theory with equality $\mathcal{E}$ with $\mathcal{E}(-, \Omega) \simeq \text{Sub}(-)$.

For presheaves, the subobject classifier is defined $\Omega(S) = \{ \varphi \mapsto y(S) \}$. It is an internal complete Heyting algebra.
Language of a topos

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

**Definition**

\textbf{A topos} is a \( \lambda \)-theory with equality \( \mathcal{E} \) with \( \mathcal{E}(-, \Omega) \simeq \text{Sub}(-) \).

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

**Definition**

The \textbf{predicate functor} of a topos \( \mathcal{E} \) defined \([-, \Omega] : \mathcal{E}^{\text{op}} \to \text{CHA} \) 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 \textbf{direct image}, \textbf{preimage}, and \textbf{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

\[
single\text{.thread} := \neg[0] \wedge \neg[\neg[0] \mid \neg[0]]
\]
Language of a topos

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

Example

```
single.thread := \neg[0] \land \neg[\neg[0] \mid \neg[0]]
```

Example

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

```
\varphi[in] := [y(in), \Omega](\varphi) : y(N \times [N \to P]) \to \Omega
```

```
\varphi[in](S)(n, \lambda x.p) = \varphi(S)(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

$$\text{single.thread} := \neg[0] \land \neg[\neg[0] \lor \neg[0]]$$

Example

For a $\rho$-calculus predicate $\varphi : y(P) \to \Omega$, preimage by input is the query “inputting on what name-context pairs yield property $\varphi$?”

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

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

Example

direct-step $\exists_t \Omega^s$ and secure-step $\forall_t \Omega^s$
Predicates $\varphi : A \to \Omega$ correspond to subobjects $c(\varphi) \hookrightarrow A$. More generally, any $p : P \to A$ can be understood as a dependent type. The predicate fibration $\pi_\Omega$ embeds into the codomain fibration $\pi_\Delta$.

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 $c(\varphi) \hookrightarrow A$. More generally, any $p : P \to A$ can be understood as a dependent type. The predicate fibration $\pi_\Omega$ embeds into the codomain fibration $\pi_\Delta$.

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 $\mathcal{L}(\mathcal{E}) = \langle \pi_{\Omega \mathcal{E}}, \pi_{\Delta \mathcal{E}}, i_\mathcal{E}, c_\mathcal{E} \rangle$ defines a 2-functor $\mathcal{L} : \text{Topos} \to \text{HDT}\Sigma$.

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*.

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

The graph of rewrites is the space of all computations.

\[
g(S)(p_1, p_2) = \{ e \mid S \vdash e : p_1 \leadsto p_2 \}
\]
Applications: behavior

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

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

The graph of rewrites is the space of all computations.

\[
g(S)(p_1, p_2) = \{ e \mid S \vdash e : p_1 \rightsquigarrow p_2 \}
\]

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

\[
\Sigma e:\text{comm}(\alpha, \varphi, \chi.F).g
\]

We can then construct modalities relative to these subspaces, as well as behavioral equivalences.
Applications: refined binding

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

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

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

Then $\text{in}(n, \lambda x: \alpha.p)$ can be understood as a **query** for $\alpha$: 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 $a : \mathbb{N}$ and an untrusted $n : \mathbb{N}$, then the following program will not preserve safety on $a$.

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

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

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

where $\triangleright_{\text{act}}$ 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!