Category Theory and Mechanized Proof

CHRISTOPHER HENSON, Drexel University, USA

Category theory and programming languages have an intertwined history that continues to be exhibited in the latest surge of interest in proof assistants. We first motivate this connection by examining the categorical semantics of various lambda calculi, then turn to the existing uses of categories in proof assistants. A number of formalizations are discussed, spanning differing use cases from libraries of category theory to proof systems with category-theoretic elements. We conclude with some discussion of an ongoing effort to formalize categorical semantics in Lean, along with possible future work involving metaprogramming applications and an extensible framework for categorical semantics.

1 INTRODUCTION

While originally introduced in the context of nascent algebraic topology [22], category theory was quickly shown to extend to a possible foundation for mathematics [27]. Perhaps then, with a bit of hindsight, it is not so surprising that we would find a connection between category theory and the various lambda calculi, or in actuality with any system strong enough to serve as the metatheory for a general-use proof assistant. In some sense, the bar for forming a category is not so high, as it requires only associative composition and the existence of identities. What exactly then is gained in intermingling category theory with the lambda calculus, and more generally with proof assistants?

This question is partially answered by *categorical semantics* [17], which connects type theory and category theory in the same sense that the Curry-Howard isomorphism connects computation and intuitionistic logic. While the syntax of a lambda calculus itself forms a category, categorical semantics provide a more interesting construction of a mapping from a lambda calculus to some general collection of categories such that typing derivations are preserved. In parallel to some motivations for category theory itself, this provides a method for determining which pieces of mathematical machinery are essential components for some given lambda calculus. For instance, an essential component of a typed lambda calculus is some notion of a typing context, which must in turn be represented in its categorical semantics. While this is a relatively straightforward example, other systems have more complex categorical interpretations, for instance linear logic which can involve Gödel's Dialectica interpretation of intuitionistic logic [20]. In the presence of these more complex constructions, categorical semantics provides a method for understanding and relating different computational systems.

Another answer is simply that a significant contingent of contemporary mathematicians use and would like to formalize category theory. Given that category theory is general enough to be treated as a foundation for mathematics, an implementation embedded within a proof assistant raises some delicate issues [5]. Formalization of category theory requires a careful understanding of the host proof assistant's metatheory and several choices of how to lift notions such as equality and size issues that may not be as carefully considered in an informal setting. Beyond theoretic concerns, implementing a category theory library provides significant software engineering challenges [2]. Of special interest is the degree of automation that is possible through the use of metaprogramming and specialized tactics.

Lastly, the nature of categorical semantics suggests that, just as one can build proof terms as a program, it should also be possible to build proofs directly using categories. In some instances, this has taken the form of specialized proof assistants designed for working with certain classes of categories [1] [26], while others have taken the approach of an embedded system in an existing proof assistant [3] [9]. In recent years there has been close attention to implementing graphical interfaces for these proof systems. Again, specialized tactics and metaprogramming open up the possibility

even in the embedded case of not just providing static diagrams but an interactive interface that is able to incorporate proof state and mirror the experience of working with commutative diagrams.

This paper surveys these aspects of the integration of category theory and proof assistants, beginning with categorical semantics to motivate the connection between categories and programming languages. In the second half of the paper, we generally consider the current usage of categories in proof assistants and some possible directions for future work regarding an extensible framework for formalized categorical semantics.

2 CATEGORICAL SEMANTICS

We begin with an informal treatment of the categorical semantics of simply typed, polymorphic [6] [8], and dependently typed lambda calculi [7] [4], postponing most discussion of formalization concerns. An attempt has been made to introduce a minimal set of categorical constructions as their necessity arises in giving semantics to each lambda calculus.

2.1 Simply Typed Lambda Calculus

We consider the simply typed lambda calculus extended with products, a unit type, and types parameterized over some collection of ground types $\mathbb{G}$ as:

$$\mathrm{Ty}_{\mathbb{G}} := G \in \mathbb{G} \mid \top \mid \sigma \times \tau \mid \sigma \to \tau$$

with $\sigma, \tau \in \mathrm{Ty}_{\mathbb{G}}$. Terms are formed as

$$\mathrm{Term}_{\mathbb{G}} := x \in \mathbb{V} \mid \langle\rangle : \top \mid (t, t') \mid \mathtt{fst}\, t \mid \mathtt{snd}\, t \mid \lambda x : \sigma \,.\, t \mid t\, t'$$

where $t, t' \in \mathrm{Term}_{\mathbb{G}}$, $\sigma \in \mathrm{Ty}_{\mathbb{G}}$, and $\mathbb{V}$ is a countably infinite collection of variables. We also define *typing contexts*, which are intuitively lists of types

$$\mathrm{Ctx}_{\mathbb{G}} := [] \mid \Gamma, \sigma$$

for $\Gamma \in \mathrm{Ctx}_{\mathbb{G}}$ and $\sigma \in \mathrm{Ty}_{\mathbb{G}}$. Contexts come equipped with a notion of membership at the metatheory level. Our formalization takes the approach of *de Bruijn representation* and *intrinsic typing* [34]. As such, we do not give a careful treatment of the issues of binding and free variables. While we informally define $\mathrm{Term}_{\mathbb{G}}$ and use variable names for readability, in the formalization $\mathrm{Term}_{\mathbb{G}}$ is not directly defined. We instead work only with typing derivations that are correct by construction.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ (\textsc{Var}) \qquad \frac{}{\Gamma \vdash \langle\rangle : \top}\ (\textsc{Unit}) \qquad \frac{\Gamma, x : \sigma \vdash \tau}{\Gamma \vdash \sigma \to \tau}\ (\textsc{Lam}) \qquad \frac{\Gamma \vdash \sigma \to \tau \qquad \Gamma \vdash \sigma}{\Gamma \vdash \tau}\ (\textsc{App})$$

$$\frac{\Gamma \vdash \sigma \qquad \Gamma \vdash \tau}{\Gamma \vdash \sigma \times \tau}\ (\textsc{Pair}) \qquad \frac{\Gamma \vdash \sigma \times \tau}{\Gamma \vdash \sigma}\ (\textsc{Fst}) \qquad \frac{\Gamma \vdash \sigma \times \tau}{\Gamma \vdash \tau}\ (\textsc{Snd})$$

For two contexts Γ and Δ such that there exists a mapping $\rho : T \in \Gamma \to \Delta \vdash T$, *simultaneous substitution* is a lifting of type $\Gamma \vdash T \to \Delta \vdash T$ that intuitively represents the result of substituting each type of a context with some term. *Single substitution* is then just the special case admitted by the subset mapping $\Gamma \subseteq \Gamma, \sigma$. In both cases, we use the notation $t[x \mapsto x']$, which in a named form represents replacing the free occurrences of x with x'.

We then can define the equivalence relation of reduction as

$$\frac{\Gamma \vdash \sigma}{\Gamma \vdash \sigma = \sigma}\ (\textsc{Rfl}) \qquad \frac{\Gamma \vdash \sigma = \tau}{\Gamma \vdash \tau = \sigma}\ (\textsc{Sym}) \qquad \frac{\Gamma \vdash \sigma = \tau \qquad \Gamma \vdash \tau = \varphi}{\Gamma \vdash \tau = \varphi}\ (\textsc{Trans})$$

$$\frac{\Gamma, x : \sigma \vdash t : \tau \qquad \Gamma \vdash x' : \sigma}{\Gamma \vdash (\lambda x : \sigma \,.\, t)\, x' = t[x \mapsto x'] : \tau}\ (\beta\text{-Lam}) \qquad \frac{\Gamma \vdash t : \sigma \qquad \Gamma \vdash s : \tau}{\Gamma \vdash \mathtt{fst}\ (t, s) = t : \sigma}\ (\beta\text{-Fst}) \qquad \frac{\Gamma \vdash t : \sigma \qquad \Gamma \vdash s : \tau}{\Gamma \vdash \mathtt{snd}\ (t, s) = s : \tau}\ (\beta\text{-Snd})$$

$$\frac{\Gamma \vdash t : \sigma \times \tau}{\Gamma \vdash t = (\mathtt{fst}\, t, \mathtt{snd}\, t) : \sigma \times \tau}\ (\eta\text{-Pair}) \qquad \frac{\Gamma \vdash t : \top}{\Gamma \vdash t = \langle\rangle : \top}\ (\eta\text{-Unit}) \qquad \frac{\Gamma \vdash t : \sigma \to \tau}{\Gamma \vdash t = \lambda x : \sigma \,.\, (t\, x)}\ (\eta\text{-App})$$

$$\frac{\Gamma \vdash t = t' : \sigma \times \tau}{\Gamma \vdash \mathtt{fst}\, t = \mathtt{fst}\, t' : \sigma}\ (\text{Cong-Fst}) \qquad \frac{\Gamma \vdash t = t' : \sigma \times \tau}{\Gamma \vdash \mathtt{snd}\, t = \mathtt{snd}\, t' : \tau}\ (\text{Cong-Snd})$$

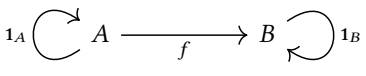
$$\frac{\Gamma \vdash s = s' : \sigma \qquad \Gamma \vdash t = t' : \tau}{\Gamma \vdash (s, t) = (s', t') : \sigma \times \tau}\ (\text{Cong-Pair}) \qquad \frac{\Gamma \vdash t = t' : \sigma \to \tau \qquad \Gamma \vdash s = s' : \sigma}{\Gamma \vdash t\, s = t'\, s' : \tau}\ (\text{Cong-App})$$

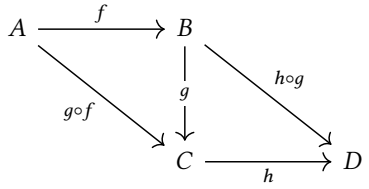
2.2 Categories

A category $\mathbf{C}$ consists of

- a collection Obj($\mathbf{C}$) of *objects*
- a collection of *morphisms* (also called *arrows*), denoted by $f : A \to B$, for *source* object A and *target* object B.
- an operation *composition*, denoted by $\circ$, that for any morphisms $f : A \to B$, $g : B \to C$, and $h : C \to D$ satisfies the associative rule
$$(h \circ g) \circ f = h \circ (g \circ f)$$
- for each object X, an *identity* arrow $\mathbf{1}_X : X \to X$ such that for any arrow $f : A \to B$
$$\mathbf{1}_B \circ f = f = f \circ \mathbf{1}_A$$

Given a morphism $f : A \to B$, we respectively denote the source and target as $\mathrm{dom}(f)$ and $\mathrm{cod}(f)$ and use the notation $\mathbf{C}(A, B)$ for the *hom-set*, the collection of morphisms with this typing. Categories are often drawn using commutative diagrams, where any paths with the same endpoints are considered equal. For example, the above composition rules above would be represented as





Intuitively, the aim of our categorical semantics will be to find a mapping from the lambda calculus to some general collection of categories that preserves the above typing and reduction judgements. As a first step, we need to add some restraining structure to our definition of a category that will ensure that we are able to model contexts, types, and typing derivations.

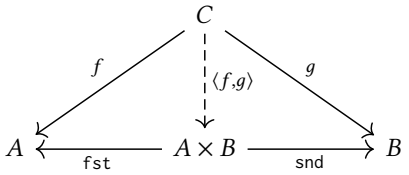
2.3 Cartesian Closed Categories

A *Cartesian closed category* has the additional requirements that

- there exists a *terminal object* $\top$, such that for every object A there is a unique arrow $\langle\rangle_A : A \to \top$
- the category has *binary products*, such that for all objects A, B, and C

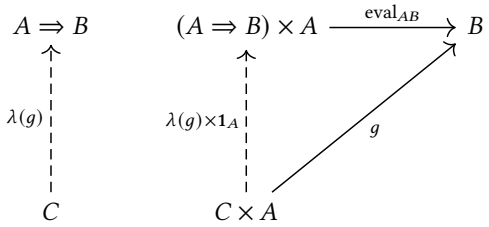
  - there exists a *product object* $A \times B$
  - *projections* $\texttt{fst} : A \times B \to A$ and $\texttt{snd} : A \times B \to B$
  - a unique *mediating arrow* $\langle f, g \rangle : C \to A \times B$

  that make the following diagram commute



- the category has *exponentiation*, such that for all objects A, B, and C
  - there exists an *exponential object* $A \Rightarrow B$
  - an arrow $\text{eval}_{AB} : ((A \Rightarrow B) \times A) \to B$
  - for each morphism $g : C \times A \to B$, a unique arrow $\lambda(g) : C \to (A \Rightarrow B)$

  that make the following diagram commute



2.4 Categorical Semantics of the Simply Typed Lambda Calculus

The key intuition is that the products of a Cartesian closed category provide a structure that can model typing contexts, while exponents in turn correspond to lambda abstraction and application. Types and contexts are both mapped to objects and typing derivations are mapped to morphisms.[1]

Let $\mathbf{C}$ be some Cartesian closed category such that there is a mapping $M : \mathbb{G} \to \text{Obj}(\mathbf{C})$ from ground types to objects. First we define the mapping from types to objects

$$\begin{aligned}
[\![\cdot]\!]_{\text{Ty}} &: \text{Ty}_{\mathbb{G}} \to \text{Obj}(\mathbf{C}) \\
[\![\top]\!]_{\text{Ty}} &= \top \\
[\![\sigma \times \tau]\!]_{\text{Ty}} &= [\![\sigma]\!]_{\text{Ty}} \times [\![\tau]\!]_{\text{Ty}} \\
[\![\sigma \to \tau]\!]_{\text{Ty}} &= [\![\sigma]\!]_{\text{Ty}} \Rightarrow [\![\tau]\!]_{\text{Ty}}
\end{aligned}$$

Next the mapping for contexts, which are also sent to objects

$$\begin{aligned}
[\![\cdot]\!]_{\text{Ctx}} &: \text{Ctx}_{\mathbb{G}} \to \text{Obj}(\mathbf{C}) \\
[\![[\,]]\!]_{\text{Ctx}} &= \top \\
[\![\Gamma, x : \sigma]\!]_{\text{Ctx}} &= [\![\sigma]\!]_{\text{Ty}} \times [\![\Gamma]\!]_{\text{Ctx}}
\end{aligned}$$

[1]While we have not yet mentioned functors, an intuitive viewpoint is thinking of categorical semantics (of simple types) as a collection of functors from the syntax of the lambda calculus to an arbitrary Cartesian closed category.

and finally typing derivations, which are sent to morphisms whose source and target are determined by the context and type of the derivation.

$$\llbracket \Gamma \vdash T \rrbracket_{\vdash} : \mathbf{C}(\llbracket \Gamma \rrbracket_{\mathrm{Ctx}}, \llbracket T \rrbracket_{\mathrm{Ty}})$$

$$\llbracket \Gamma \vdash \langle\rangle : \top \rrbracket_{\vdash} = \llbracket \Gamma \rrbracket_{\mathrm{Ctx}} \xrightarrow{\langle\rangle} \top$$

$$\llbracket \Gamma \vdash \mathtt{fst}\,(t : \sigma \times \tau) \rrbracket_{\vdash} = \llbracket \Gamma \rrbracket_{\mathrm{Ctx}} \xrightarrow{\llbracket \Gamma \vdash t : \sigma \times \tau \rrbracket_{\vdash}} \llbracket \sigma \times \tau \rrbracket_{\mathrm{Ty}} \xrightarrow{\mathtt{fst}} \llbracket \sigma \rrbracket_{\mathrm{Ty}}$$

$$\llbracket \Gamma \vdash \mathtt{snd}\,(t : \sigma \times \tau) \rrbracket_{\vdash} = \llbracket \Gamma \rrbracket_{\mathrm{Ctx}} \xrightarrow{\llbracket \Gamma \vdash t : \sigma \times \tau \rrbracket_{\vdash}} \llbracket \sigma \times \tau \rrbracket_{\mathrm{Ty}} \xrightarrow{\mathtt{snd}} \llbracket \tau \rrbracket_{\mathrm{Ty}}$$

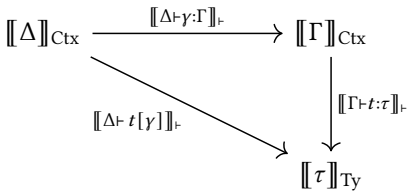
$$\llbracket \Gamma \vdash (s, t) : \sigma \times \tau \rrbracket_{\vdash} = \llbracket \Gamma \rrbracket_{\mathrm{Ctx}} \xrightarrow{\langle \llbracket \Gamma \vdash s : \sigma \rrbracket_{\vdash}, \llbracket \Gamma \vdash t : \tau \rrbracket_{\vdash} \rangle} \llbracket \sigma \times \tau \rrbracket_{\mathrm{Ty}}$$

$$\llbracket \Gamma \vdash (\lambda x\,.\,t) : \sigma \to \tau \rrbracket_{\vdash} = \lambda \left(\llbracket \sigma \rrbracket_{\mathrm{Ty}} \times \llbracket \Gamma \rrbracket_{\mathrm{Ctx}} \xrightarrow{\llbracket \Gamma, x : \sigma \vdash \tau \rrbracket_{\vdash}} \llbracket \tau \rrbracket_{\mathrm{Ty}} \right)$$

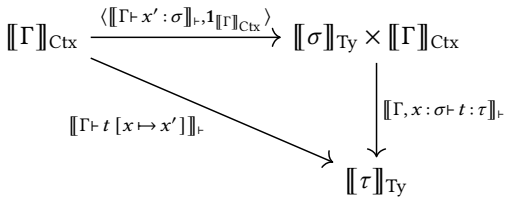
$$\llbracket \Gamma \vdash (t : \sigma \to \tau)\, t' \rrbracket_{\vdash} = \llbracket \Gamma \rrbracket_{\mathrm{Ctx}} \xrightarrow{\langle \llbracket \Gamma \vdash \sigma \to \tau \rrbracket_{\vdash}, \llbracket \Gamma \vdash \sigma \rrbracket_{\vdash} \rangle} (\llbracket \sigma \rrbracket_{\mathrm{Ty}} \Rightarrow \llbracket \tau \rrbracket_{\mathrm{Ty}}) \times \llbracket \sigma \rrbracket_{\mathrm{Ty}} \xrightarrow{\mathrm{eval}} \llbracket \tau \rrbracket_{\mathrm{Ty}}$$

Two theorems demonstrate that this mapping preserves the semantics of the simply typed lambda calculus. First, we have that composition corresponds to substitution:

Theorem 2.1 (Substitution). *Suppose there exists a mapping $\rho : T \in \Gamma \to \Delta \vdash T$ and typing derivations $\Delta \vdash \gamma : \Gamma$ and $\Gamma \vdash t : \tau$. Then the following diagram commutes*



and has as a special case the single substitution corresponding to the diagram



Importantly, we also have that the mapping preserves reductions as equality of morphisms:

Theorem 2.2 (Soundness). *Let $\mathbf{C}$ be a Cartesian closed category with some embedding of ground types $M : \mathbb{G} \to \mathrm{Obj}(\mathbf{C})$. Then we have the implication*

$$\Gamma \vdash t = t' : \tau \implies \llbracket \Gamma \vdash t \rrbracket_{\vdash} = \llbracket \Gamma \vdash t' \rrbracket_{\vdash}$$

For brevity, we will omit the straightforward statement of soundness for the polymorphic and dependently typed lambda calculi, but note that they are essential for a meaningful categorical semantics. The direction of the implication is also important. One could easily select a category with much additional structure such that not every equal morphism corresponds to a typing derivation.

2.5 Polymorphic Lambda Calculus

The polymorphic lambda calculus extends the simply typed lambda calculus by adding *type variables*, which allow type expressions with universally quantified types. Types now have the form

$$\mathrm{Ty}_{\mathbb{G}} := G \in \mathbb{G} \mid \top \mid \sigma \times \tau \mid \sigma \to \tau \mid \boxed{X : * \mid \forall X\,.\,(\Phi \in \mathrm{Ty}_{\mathbb{G}})}$$

Here we have denoted type variables by $X : *$, which have an ascription as a *kind*. Similarly to terms for the simply typed lambda calculus, we now have derivations for types that use *kinding contexts*. As we will not be considering higher-kinded types, a kinding context is simply a list of repeated elements

$$\Delta = \underbrace{[*, *, \ldots, *]}_{n}$$

corresponding to n available type variables. These kinding contexts then give the derivations

$$\frac{X : * \in \Delta}{\Delta \vdash_* X : *}\ (\textsc{Ty-Var}) \qquad \frac{\Delta,\, X \vdash_* Y}{\Delta \vdash_* \forall X\,.\,Y}\ (\textsc{Ty-}\forall)$$

where we assume appropriate binding of type variables. The syntax of our terms now adds type abstraction and application

$$\mathrm{Term}_{\mathbb{G}} := x \in \mathbb{V} \mid \langle\rangle : \top \mid (t, t') \mid \mathtt{fst}\, t \mid \mathtt{snd}\, t \mid \lambda x : \sigma\,.\,t \mid t\,t' \mid \boxed{\Lambda X : *\,.\,t}$$

Again, these named terms are just for convenience. In our formalization we work with an intrinsically typed implementation of the polymorphic lambda calculus [15] with the following additional term derivations indexed by both a kinding and typing context

$$\frac{\Delta, X : *;\, \Gamma \vdash \tau}{\Delta;\, \Gamma \vdash \Lambda X : *\,.\,\tau}\ (\textsc{Ty-Lam}) \qquad \frac{\Delta;\, \Gamma \vdash \Lambda X : *\,.\,t : \tau \qquad \Delta \vdash_* Y : *}{\Delta;\, \Gamma \vdash t[X \mapsto Y] : \tau}\ (\textsc{Ty-App})$$

where we allow for substitution to also occur within types. Finally, we extend reduction to additionally include

$$\frac{\Delta;\, \Gamma \vdash \Lambda X : *\,.\,t : \tau \qquad \Delta \vdash_* Y : *}{\Delta;\, \Gamma \vdash (\Lambda X : *\,.\,t : \tau)\; Y = t[X \mapsto Y : \tau]}\ (\beta\textsc{-Ty-App})$$

2.6 Functors

As discussed further in section 2.10, the introduction of polymorphism's kinding contexts forces us to consider maps between categories, partially motivating the following definition. Given

- two categories $\mathbf{C}$ and $\mathbf{D}$
- any objects X, Y, Z from $\mathbf{C}$
- any arrows $f : X \to Y$ and $g : Y \to Z$ (from $\mathbf{C}$)

a *functor* $F : \mathbf{C} \to \mathbf{D}$ is a mapping between the objects and morphisms of $\mathbf{C}$ and $\mathbf{D}$ that satisfies

$$F(\mathbf{1}_X) = \mathbf{1}_{F(X)}$$
$$F(g \circ f) = F(g) \circ F(f)$$

If these conditions hold only up to isomorphism rather than strict equality, we instead say that we have a *pseudofunctor*. A typical example given of a functor relevant to programming languages is lists [30], where we consider

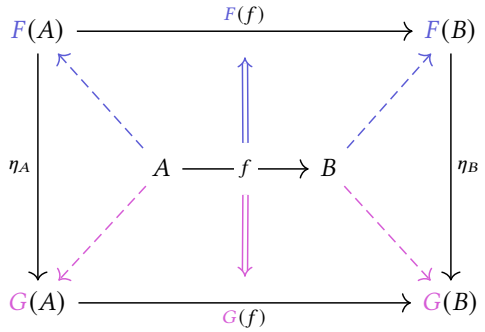
- **Set**, the category of sets
- **Mon**, the category of monoids (sets equipped with an associative binary operation and identity element)
- the functor $List : \mathbf{Set} \to \mathbf{Mon}$ which
  - takes each set to the monoid of lists using these elements, where the operation is list concatenation
  - takes each arrow (function) $f : A \to B$ to the function $\text{map}_f : List(A) \to List(B)$

2.7 Natural Transformations

Taking one further step of abstraction, we can consider mappings between functors that preserve some structure. This notion is expressed through the idea of a *natural transformation*. Given

- two categories $\mathbf{C}$ and $\mathbf{D}$
- functors $F, G : \mathbf{C} \to \mathbf{D}$
- for every object X in $\mathbf{C}$, a morphism $\eta_X : F(X) \to G(X)$

that make the following diagram[2] commute for any arrow $f : A \to B$ in $\mathbf{C}$



we say that η is a natural transformation from F to G. The intuition is that for any morphism in category $\mathbf{C}$ that these pair of functors have analogous structure preserved by the natural transformation.

Continuing with the example of lists, consider any polymorphic function with type

$$r : \forall X.\ List(X) \to List(X)$$

which could for instance be a function that reverses lists. In this context r is a natural transformation, giving exactly the same parametricity [33] result that for any $g : A \to B$

$$r_B \circ \text{map}_g = \text{map}_g \circ r_A$$

2.8 Indexed Categories

The category of categories **Cat** then has categories as objects, functors as morphisms, and natural transformations as functors. Given a category $\mathbf{C}$, the pseudofunctor $\mathbb{C} : \mathbf{C}^{\text{op}} \to \mathbf{Cat}$ is called an *indexed category*. Intuitively, the name describes how each object of $\mathbf{C}$ specifies a category. For any object $A \in \text{Obj}(\mathbf{C})$, this category $\mathbb{C}(A)$ is sometimes referred to as a *fiber* of the indexed category.

[2]This diagram is a touch nonstandard, but provides for nice intuition. The colored arrows represent both the object and morphism parts of the functors.

2.9 Categorical Semantics of the Polymorphic Lambda Calculus

For the categorical semantics of the polymorphic lambda calculus [13], first assume that we have some Cartesian closed category $\mathbf{C}$, and consider only repeated products of some distinguished object Ω. These indices then are the semantics for kinding contexts, such that

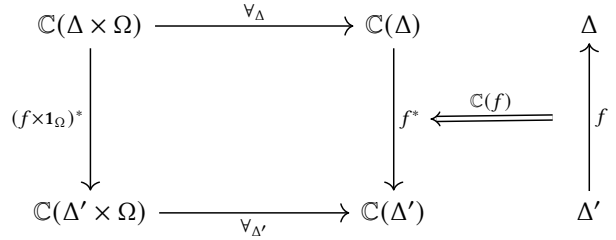
$$[\![\Omega^n]\!]_* = \underbrace{[*, *, \ldots, *]}_{n}$$

We further interpret the morphisms $\mathbf{C}(\Omega^n, \Omega)$ as a type with n free type variables, but must consider this alongside a functor $\mathbb{C} : \mathbf{C}^{\mathrm{op}} \to \mathbf{Cat}$ to establish a categorical semantics. For each kinding context Ω^n, the functor gives a category $\mathbb{C}(\Omega^n)$ that is a model of the simply typed lambda calculus, with the same categorical semantics as outlined above.

The additional structure that we require for polymorphism is a way to model the universal quantifiers that now appear in types and terms. This takes the form of a functor that relates the fibers of our indexed category, specifically

$$\forall_\Delta : \mathbf{C}(\Delta \times \Omega) \to \mathbf{C}(\Delta)$$

that represents the addition of a free variable under universal quantification, parameterized by some kinding context Δ. This universal quantification functor must also satisfy the natural transformation given by the commutative diagram



2.10 Why Not Sets?

One point of interest brought up by polymorphic categorical semantics is the relationship to set theory. For the simply typed lambda calculus, our semantics required only a Cartesian closed category, and thus **Set** is a perfectly valid interpretation. Even with the above semantics of polymorphism, you could still choose to interpret each fiber as a one of a countably infinite number of "copies" of the category of sets corresponding to each parameterization of the simply typed lambda calculus with particular type variables. A natural question to then ask is if it would be possible to encode these same semantics with a single copy of **Set**?

The answer is that given some reasonable assumptions about compatibility with the semantics of the simply typed lambda calculus and semantics depending on contexts in an expected way, that a relatively simple cardinality contradiction is found in a set-theoretic model [32]. In fact, this argument was phrased categorically as the non-existence of a functor into the category of sets that fully preserves the Cartesian structure and indexed products. This was later refined into showing that a restriction to *constructive sets* does admit a polymorphic categorical semantics [6], again using an argument that heavily involved category theory by embedding each categorical semantics in another categorical (topos) model known to relate to intuitionistic logic.

In some sense, both these positive and negative results show the ability of categorical semantics to aid in understanding the inherent constraints provided by a particular lambda calculus. Very informally, one could say that trying to find a categorical semantics involving some familiar category

has the potential to identify the compatibility or incompatibility of how much "information" is respectively carried by each.

2.11 Dependently Typed Lambda Calculus

The dependently typed lambda calculus extends the simply typed lambda calculus by allowing types to depend on terms. Types now have the form [31]

$$\mathrm{Ty}_{\mathbb{G}} := G \in \mathbb{G} \mid \top \mid \sigma \times \tau \mid \sigma \to \tau \mid \boxed{X : * \mid \Pi x : T \,.\, \Phi \in \mathrm{Ty}_{\mathbb{G}}}$$

where the Π type binds a term that can then be used in a type, and we again have kind ascriptions that identify properly formed types. Our contexts may now contain both type and term bindings, which interact via the following judgements

$$\frac{\Gamma \vdash \sigma : * \qquad \Gamma, x : \sigma \vdash \tau}{\Gamma \vdash \Pi x : \sigma \,.\, \tau} \text{ (WF-PI)} \qquad \frac{\Gamma \vdash \sigma : * \qquad x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \text{ (WF-VAR)}$$

With the addition of Π types, we allow substitution within types and add judgements that mirror lambda abstraction at the type level

$$\frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash (\lambda x : \sigma \,.\, t) : (\Pi x : \sigma \,.\, \tau)} \text{ (PI-LAM)} \qquad \frac{\Gamma \vdash t : (\Pi x : \sigma \,.\, U) \qquad \Gamma \vdash s : \sigma}{\Gamma \vdash t\, s : U[x \mapsto s]} \text{ (PI-APP)}$$

along with notions of equivalence at the type and term levels

$$\frac{\Gamma \vdash T = T' : * \qquad \Gamma \vdash t : T}{\Gamma \vdash t : T'} \text{ (TY-CONV)} \qquad \frac{\Gamma, x : \sigma \vdash t : \tau \qquad \Gamma \vdash s : \sigma}{\Gamma \vdash (\lambda x \,.\, t)\, s = t[s \mapsto x] : \tau[s \mapsto x]} \text{ (TY-SUB)}$$

$$\frac{\Gamma \vdash \sigma = \tau : * \qquad \Gamma, x : \tau \vdash \sigma' = \tau' : *}{\Gamma \vdash (\Pi x : \sigma \,.\, \sigma') = (\Pi x : \tau \,.\, \tau') : *} \text{ (PI-CAST)} \qquad \frac{\Gamma \vdash \sigma = \sigma' : * \qquad \Gamma, x : \sigma \vdash t = t' : \tau}{\Gamma \vdash (\lambda x : \sigma \,.\, t) = (\lambda x : \sigma' \,.\, t') : \Pi x : \sigma \,.\, \tau} \text{ (LAM-CAST)}$$

which have appropriate type-level reflexive, symmetric, and transitive judgements that we omit for the sake of brevity.

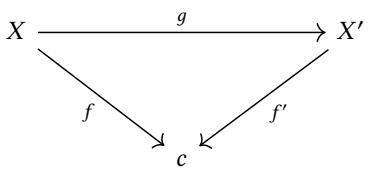
Dependent types in particular have inspired a variety of categorical semantics, with at least half a dozen different constructions appearing in the last fifty years [23]. Here we mention two that are notable for their intuitive extension of the categorical semantics we have already seen or are particularly amenable to formalization.

2.12 Slice Categories and Locally Cartesian Categories

Given some category $\mathbf{C}$ and object $c \in \mathrm{Obj}(\mathbf{C})$, the *slice category* $\mathbf{C}/c$ has

- as objects, a collection of morphisms such that $f \in \mathbf{C}(-, c)$
- a collection of morphisms such that for
  - $f, f' \in \mathbf{C}(-, c)$,
  - $g : \mathrm{dom}(f) \to \mathrm{dom}(f')$

the following diagram commutes



If all slices of a category are Cartesian closed, we call this a *locally Cartesian closed category* (LCCC). Historically, one of the first approaches to providing a categorical semantics for dependent types was to use LCCCs. Informally, this semantics maps

- contexts to objects and substitution to morphisms (as with simple types)
- dependent types as objects in a slice category
- terms as sections

Despite the clarity of this approach in extending the categorical semantics of simple types, there is a significant coherence issue found by using the composition of pullbacks as a model of substitution [18, 19, 24]. While we mention this approach for its historical and intuitive relevance, these issues push us to consider other approaches for formalization.

2.13 Dependently Typed Categorical Semantics via Categories with Families

In part driven by these issues, *categories with families* (CwFs) [21] emerged as another option for categorical semantics in some sense closer to the syntax of dependent types. First we consider *indexed sets*, which in a type-theoretic setting we treat as the pair

$$B = (B^0 : \text{Type},\ B^1 : B^0 \to \text{Type})$$

which forms the category **Fam** when equipped with morphisms

$$B \to C := (f^0 : B^0 \to C^0, f^1_b : B^1(b) \to C^1(f^0(b)))$$

A CwF then contains [4]

- a category $\mathbf{C}$, representing contexts
- a functor $F := (Ty, Tm) : \mathbf{C}^{\text{op}} \to \mathbf{Fam}$, representing types and terms

such that given contexts Γ and Δ, a morphism $f : \Gamma \to \Delta$, and a type $\sigma \in Ty(\Delta)$, there exists

- a type-level substitution operation $-\{f\} : Ty(\Delta) \to Ty(\Gamma)$
- a term-level substitution $-\{f\} : Tm(\Delta, \sigma) \to Tm(\Gamma, \sigma\{f\})$

that both respect identity and composition, along with projections

- $p : \Gamma, \sigma \to \Gamma$, representing a Cartesian projection
- $v_\sigma : Tm(\Gamma.\sigma,\ \sigma\{p(\sigma)\})$, representing the dependent projection

such that there is a unique morphism $\langle f, M\rangle_\sigma : \Gamma \to \Delta, \sigma$ that satisfies

$$p(\sigma) \circ \langle f, M\rangle_\sigma = f$$
$$v_\sigma\{\langle f, M\rangle_\sigma\} = M$$

Essentially the role of categories with families is to equationally specify something that is very close to the existing syntax of the dependently typed lambda calculus, thereby avoiding coherence issues. As discussed further in section 5.1, this also translates into an easier implementation within a proof assistant.

3 FORMALIZATION OF CATEGORY THEORY

While many attempts at formalization of mathematics present issues that do not occur in an informal setting, category theory presents particularly pointed challenges. Some are matters of foundations, which is perhaps to be expected when attempting to formalize a branch of mathematics that can itself independently serve as a foundation for mathematics. Other issues are of a more practical nature, concerning matters of performance and developing an interface that aims to capture the experience and benefits of working informally with categories. Here we survey the most prominent of these challenges.

3.1 Collections and Equality

Implicit in the above definition of categories is what is exactly meant by a "collection" of objects and morphisms [5]. Even considered informally, it is easy to quickly run into foundational issues. As an example consider **Set**, the category of sets. Even in defining this, we are forced to address some thorny questions. Putting aside the question of exactly what foundations we are using to define a set, we can conclude that **Set**'s collection of object cannot itself be a set, as this would run into some variation of the size issues of Russell's paradox. In a proof assistant, this is commonly handled by specifying *universe levels* for the types of objects and morphisms and definitions that utilize *universe polymorphism*.

In embedding our definition of a category into the setting of a proof assistant, which we assume is some variant of the calculus of inductive constructions with universes, one choice that is fairly fixed is that objects should be modeled as types. There is however a choice between defining a *strict category*, which has some notion of equality of objects, versus a *weak category*, which has no such requirement. Typically formalizations of category theory have tended towards the latter.

A bit more thought is required for morphisms. First, consider if we were to define a single type for the collection of morphisms. This immediately presents a practical issue for the definition of composition, which must then restrict its arguments to morphisms with appropriately typed source and target objects. With a single type for all hom-sets, this means this restriction would be forced to be implemented via an argument checking for type equality, creating significant practical issues of usability and readability. More commonly, the definition of morphisms is parameterized by the types of the source and target objects, splitting the full hom-set into a family of dependent types.

This still leaves the question of how to define equality of morphisms, which in the informal setting is assumed to be part of some underlying metatheory. Several options have emerged:

- *Propositional Equality*, which lifts the proof assistant's notion of definitional (judgemental) equality into a propositional type
- *Setoids*, with which a user can define their own notion of equality to be used for each category
- *Higher Inductive Types*, a extension of inductive types that natively includes equality proofs between constructors

Each of these comes with their own advantages and disadvantages. While on the surface appearing simple, using the proof assistant's native notion of equality directly or via propositional equality is often not a practical choice unless augmented by additional axioms. Categories often have functions and propositions as morphisms, which then prompts the axiomatic inclusion of the *Uniqueness of Identity Proofs* (UIP) or *functional extensionality* respectively for the sake of usability. While this resolves practical issues, it can be undesirable from the perspective of maintaining compatibility with constructive mathematics or other libraries with different axioms.

In comparison, setoids work with equality in a *proof relevant* way that does not require any additional axioms. The trade-off here is overhead that affects both performance and user interface. Each category now requires its own definition of an equivalence relation, along with a proof that this relation is compatible with composition. In some cases, for instance if we need a higher-level equivalence relation on setoids themselves, this can be a significant burden [2].

Higher inductive types, a more recent development originating from homotopy type theory, essentially offloads the proof burdens of custom equivalence to the typechecker itself. The primary trade-offs here are type system complexity and that this feature is not currently supported by most proof assistants.

While there remains a variety of definitions of categories across proof assistants, their differences are primarily attributable to the above choices regarding the treatment of equality, typically agreeing

on the usage of dependently typed morphisms and universes. We can thus think of the generally accepted embedding into proof assistants as the following typing derivations for some category $\mathbf{C}$

$$\frac{}{\mathrm{Obj}(\mathbf{C}) : \mathrm{Type}_u}\ (\textsc{Obj}) \qquad \frac{A,\ B : \mathrm{Obj}(\mathbf{C})}{\mathbf{C}(A,B) : \mathrm{Type}_v}\ (\textsc{Hom}) \qquad \frac{A : \mathrm{Obj}(\mathbf{C})}{\mathbf{1}_A : \mathbf{C}(A,A)}\ (\textsc{Id}) \qquad \frac{f : \mathbf{C}(A,B) \qquad g : \mathbf{C}(B,C)}{g \circ f : \mathbf{C}(A,C)}\ (\textsc{Comp})$$

$$\frac{f : \mathbf{C}(A,B)}{\mathbf{1}_B \circ f = f}\ (\textsc{Left-Id}) \qquad \frac{f : \mathbf{C}(A,B)}{f \circ \mathbf{1}_A = f}\ (\textsc{Right-Id}) \qquad \frac{f : \mathbf{C}(A,B) \qquad g : \mathbf{C}(B,C) \qquad h : \mathbf{C}(C,D)}{(h \circ g) \circ f = h \circ (g \circ f)}\ (\textsc{Assoc})$$

which are essentially the same as the initial definition of a category, but enforce the dependent typing of morphisms and universe levels u and v for objects and morphisms respectively, while still leaving the nature of equality implicit.

Figure 2 shows the definition of a category in Lean's Mathlib [28] library, which has emerged as a monolithic development of a significant amount of mathematics that includes category theory. Here the situation is exactly as described above, though again with a subtlety regarding equality, as this definition inherits proof irrelevance through UIP being built into Lean's implementation.

```
class Quiver (V : Type u) where
  Hom : V → V → Sort v

infixr:10 " ⟹ " => Quiver.Hom

class CategoryStruct (obj : Type u) extends Quiver.{v + 1} obj : Type max u (v + 1)
    where
  id : ∀ X : obj, Hom X X
  comp : ∀ {X Y Z : obj}, (X ⟹ Y) → (Y ⟹ Z) → (X ⟹ Z)

scoped notation "𝟙" => CategoryStruct.id
scoped infixr:80 " ≫ " => CategoryStruct.comp

class Category (obj : Type u) extends CategoryStruct.{v} obj : Type max u (v + 1)
    where
  id_comp : (f : X ⟹ Y), 𝟙 X ≫ f = f := ...
  comp_id : (f : X ⟹ Y), f ≫ 𝟙 Y = f := ...
  assoc : (f : W ⟹ X) (g : X ⟹ Y) (h : Y ⟹ Z), (f ≫ g) ≫ h = f ≫ g ≫ h := ...
```

Fig. 2. Mathlib's definition of a category (with some small edits for readability). Note that the direction of the composition notation is reversed from the usual mathematical convention.

Implementation in other libraries and proof assistants is similar, with the differences beyond syntax primarily being additional fields if the setoid approach to equality is taken or differences in which arguments are bundled into the record or typeclass definition (as discussed further in section 3.3).

3.2 User Proof Interface - Duality

Another area of particular interest in the formalization of category theory is how the practical experience of developing proofs and foundational definitions themselves mutually affect each other, which we refer to as an aspect of the *user proof interface*. While some of this is more prominent in proof systems developed for working with categories as discussed in section 4, duality presents

an interesting case at the object level. Several category theory libraries have made changes in definitions, even the central typeclass or records for categories, to better accommodate the formation of dual constructions and proofs.

In the informal setting we often consider the *dual category*, which reverses the direction of a category's morphisms. In turn, there are many categorical constructions and theorems that are found in dual pairs, such as initial and terminal objects. An example of a dual theorem is the uniqueness of initial objects, which is dual to the uniqueness of terminal objects in the opposite category. Informally, we expect that given a proof of some dual categorical theorem, we can replace all categorical constructions with its corresponding dual construction and be given another valid proof, and would ideally like for this to be replicated in our user proof interface. The key tension in embedding this idea in a proof assistant then becomes ensuring the correctness of these derived theorems by being able to make replacements that work with definitionally equal terms.

As an example, in some sense an axiom for the existence of duality theorems at the metatheory level, is that taking the opposite category is an involution, that is that for any category $(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}} = \mathbf{C}$. From our original category, we have a proof that for any morphisms $f : A \to B$, $g : B \to C$, and $h : C \to D$ we satisfy the associative rule

$$(h \circ g) \circ f = h \circ (g \circ f)$$

For the opposite category we then need to prove that

$$h \circ (g \circ f) = (h \circ g) \circ f$$

after reversing the direction of arrows and appropriately renaming. Regardless of which notion of equality that we have adopted for our definition of a category, there should be some proof $\equiv_{\mathrm{SYM}}$ of the symmetric property of equivalence relations that exactly proves this from the associative property in the original category. The issue arises when we apply the opposite operation twice, which results in an application of $(\equiv_{\mathrm{SYM}} \circ \equiv_{\mathrm{SYM}})$ to our original proof of associative composition. This is not definitionally equal to the identity type, removing much of the benefit of working with dual constructions. One approach to this is to amend our definition of categories to include this symmetric version of association. Then when taking the opposite category, we simply swap the two proofs of associativity, so that constructing $(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}$ results in a definitionally equal category.

This pattern of explicitly defining dual constructions or adding additional typeclass fields exists for many cases of duality. Some other examples include adding a proof that $\mathbf{1}_A \circ \mathbf{1}_A = \mathbf{1}_A$, which proves helpful when working with duals of certain functors. Additionally, as most proof assistants provide some mechanism for providing default values for typeclass/record fields, this causes no additional overhead in terms of the user proof interface when the added field is derivable as in the case above.

3.3 Definitional Shape - Record/Typeclass Bundling

While some category theory libraries have previously used Σ types as essentially an ad hoc alternative to typeclasses or records, this approach has primarily been abandoned for reasons of both performance and the fact that this sidesteps advances in proof assistants' support for namespacing, modules, and editor integration. This leaves a question of the *definitional shape* of both categories themselves and additional categorical constructions, in the sense that deciding what terms are included as internal fields versus external paramaters of a typeclass/record has effects concerning both the user proof interface and performance.

A large number of unbundled arguments can increase the term size and hinder performance, while at the same time is sometimes a necessity for usability in being able to easily work with universal quantification and computationally relevant terms. The issue here is that even a few external

parameters across deeply nested typeclass hierarchies can quickly snowball into a significant performance issue.

Correctly deciding on the correct level of bundling is also important to the user proof interface because converting or refactoring between different typeclass representations can be arduous. Some existing work has considered the possibility of moving automatically between different levels of bundling [11, 12, 16] both in the general case and in applications specific to category theory. While there has been some success, for instance using metaprogramming to convert between different levels of bundling in typeclasses for isomorphism, the general case remains an open research area.

4 CATEGORICAL PROOF SYSTEMS

Another interesting interaction beyond just formalizing category theory in a proof assistant is *proof systems* that incorporate category theory as a core piece of their metatheory. These can range from extensions to the proof interface [29], deeply-embedded domain-specific languages [3], or proof assistants designed specifically for working with some particular sort of categories [26] [1].

Even in this wide variety of use cases, some common themes emerge. One of these is an increasing focus on using commutative diagrams and other visual representations as a structured proof interface. These disparate developments seem to share some design principles:

- diagrams should be first-class objects, or at least abstracted from object-level category theory as much as possible
- specialized tactics for common categorical proofs such as commutativity and associativity
- in embedded proof systems, metaprogramming can be used to enrich typeclass inference, derive certain theorems, and provide visual integration with proof state

In particular, Lean 4 is especially amenable to extending an existing proof assistant with visual interfaces, as the core language includes modules for creating "widgets" that build user interface components in the infoview using HTML and JavaScript, while also providing the ability to interact with Lean in much the same way as a standard tactic. These core features of the language have been extended into more user-facing libraries for creating interactive UI components, including ProofWidgets [29], which has notably been adopted into Mathlib and also has included functionality for generating interactive diagrams from the Penrose [35] library. This includes functionality for working with commutative diagrams, as seen in figure 3. These diagrams are generated by using Lean's metaprogramming to identify patterns such as commutative triangles or squares, then assembling the corresponding interactive widget.

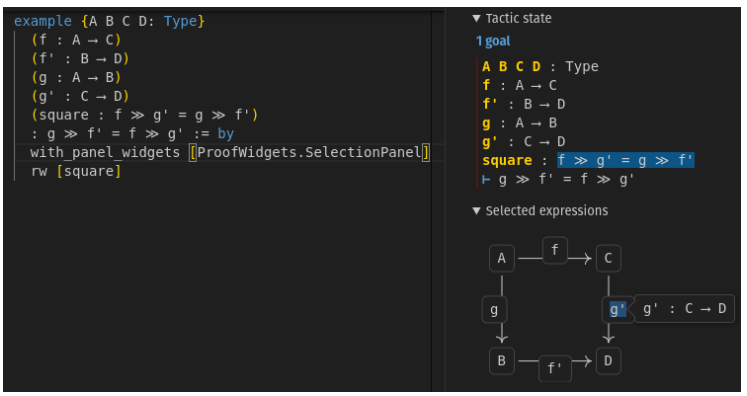


Fig. 3. An example of Mathlib rendering a commutative square in the infoview. Note the hover text that contains type information from Lean.

Another popular visual proof interface is *string diagrams*, which in the context of category theory were originally introduced to represent monoidal categories [25]. This has expanded in scope to inspire a variety of proof systems, both embedded in existing proof assistants and standalone implementations, that can handle more general constructions such as those found in higher category theory. Apart from pure category theory proofs, string diagrams have found applications in several areas such as quantum computing [9]. Here string diagrams are used in visualizing the linear transformations between quantum bits (*qubits*), which can be thought of complex vectors.

The appeal of these systems from the perspective of the user proof interface is that rules for rewriting are expressed visually as graph manipulations. There are some difficulties in translating informal string diagrams, which for instance may not treat associativity with the level of formality required for definitional equality in a proof assistant, similar to our previous discussion of duality arguments. Again, metaprogramming and tactics can approach some of these issues, such as tactics that can automatically identify when the proof goal can be reassociated to allow rewriting by some given lemma. In conjunction with other domain-specific simplification tactics such as syntactically eliminating identity morphisms or isomorphisms adjacent to their inverse, there is a possibility of a high degree of automation and coercing flexibility from a graphical representation.

5 FUTURE WORK

As part of this candidacy exam, the beginnings of a formalization of categorical semantics, with the scope of simple, polymorphic, and dependent types as described in section 2 was begun. While not complete, it provided some insight into challenges of formalizing categorical semantics and some potential directions for future work.

5.1 Challenges of Formalizing Categorical Semantics

As is common with formalization efforts, particularly those involving category theory, formalizing even the categorical semantics for simple types provides some challenges in aligning formal and informal arguments. The first of these is found in the informal interchangeability of contexts and types. For instance, in the substitution theorem (theorem 2.1), pay special attention to Γ. Is it a type or a context? In the informal usage above, it appears as both a type derived from Δ, then a context from which τ is derived. The intuition here is that in the presence of products that there is not much difference between a type and a context. In a proof assistant however, we need to resolve the incongruity between the typing derivations

$$(\cdot \vdash \cdot) : Ctx \to Ty \to \text{Type}$$

and the type of morphisms that requires two arguments with definitionally equal types for objects.

One way to proceed is to simply change the typing of $(\cdot \vdash \cdot)$ to either take two contexts or two types. The idea is that we either translate a context into a product type or consider types as singleton contexts. This works, but is a bit unsatisfying in requiring a fundamental change to the definition of typing derivations, along with a bit more convoluted requirements of restricting contexts to valid types or restricting types to valid contexts. Another approach is to define a function that transforms context into types (or vice versa), then prove that this translation maintains the semantics given by $[\![\cdot]\!]_{\text{Ctx}}$ and $[\![\cdot]\!]_{\text{Ty}}$. This raises difficulties with composition, as we must now carry around a proof of type equality.

A more subtle issue is exactly how to represent the generality of categorical semantics, which should encode not just the syntactic category, but the full correspondence between the simply typed lambda calculus and any Cartesian closed category. While there is an interpretation as a functor from the syntactic category, this both presents typing issues and is not a single functor,

but a family of functors. The approach that we took was to directly encode the three semantic functions, as seen in the excerpt of type signatures in figure 4.

```
-- specify a target Cartesian closed category
variable {X: Type} [Category X] [Limits.HasFiniteProducts X] [CartesianClosed X]
-- ground types, and their embedding into the category's objects
variable {G : Type} (f : G → X)

-- semantic functions
def Ty.sem                                : Ty G      → X                  := ...
def Context.sem                           : Context G → X                  := ...
def Term.sem  {Γ : Context G} {T : Ty G} : Γ ⊢ T     → (Γ.sem ⟹  T.sem) := ...

-- type equality of contexts and types
def Context.to_ty                 : Context G → Ty G    := ...
theorem Ctx_Ty_Eq (Γ : Context G) : Γ.to_ty.sem = Γ.sem  := ...

-- substitution, requiring type equality for composition
theorem Cat_subst (t1 : Γ ⊢ τ) (t2 : Δ ⊢ Γ.to_ty) (ρ : Δ ⇒ Γ)
  : t2.sem ≫  eqToHom (Ctx_Ty_Eq Γ) ≫  t1.sem = (Term.subst ρ t1).sem := ...

-- soundness
theorem Term.soundness {t t' : Γ ⊢ T} (eq : Γ ⊢ t =βη t' ~~ T) : t.sem = t'.sem := ...
```

Fig. 4. Type signatures for our encoding of the categorical semantics of the simply typed lambda calculus, with some syntactical liberties for readability.

The type equality difficulties with this approach can be seen in the substitution semantics, where we are forced to use `eqToHom` (a Mathlib provided utility) to convert a type equality into an additional morphism composition. While potentially unpleasant, it is perhaps inevitable that some type equalities must be utilized in formalizing categorical semantics. If not the case for simple types, this is certainly the case for semantics of dependent types themselves. While our formalization has not fully developed a categorical semantics as we did above for simple types, we did develop typeclasses representing semantic substitution and categories with families used as (one possible) categorical semantics for dependent types. As Hoffman even explicitly notes, these definitions inherently require type-level rewriting [4], as seen in figure 5.

Here `PFunctor` is a Mathlib definition corresponding to indexed sets, for which we have also provided an appropriate category typeclass instance. In order for the term-level identity and composition fields to typecheck, we are required to typecast using the corresponding type-level fields appearing earlier in the typeclass. Of note is that this is a prime example of Lean's flexible metaprogramming and tactic capabilities, both in that we are able to use a tactic to succinctly construct this typecast, as well as the fact that these rewrites were found using proof search tactics.

5.2 Metaprogramming and Extensible Categorical Semantics

Another possible direction for future work is investigating the ability to create an extensible framework for formalized categorical semantics. Existing languages such as Abella [14] and Beluga [10] have explored the idea of treating essential components of mechanized metatheory such as abstract syntax, contexts, and relational reasoning as first-class objects. A reasonable question then is if there is a similar potential for identifying shared components among different categorical semantics that could be used in a general framework for producing and reasoning about categorical semantics. As an example, typing contexts are usually given semantics via Cartesian closed categories, but

```
abbrev Ty [Category C] (F : C^op ⇉ PFunctor) (Γ : C) := (F.obj (Opposite.op Γ)).A
abbrev Tm [Category C] (F : C^op ⇉ PFunctor) (Γ : C) := (F.obj (Opposite.op Γ)).B

class SemSub [Category C] (F : C^op ⇉ PFunctor) where
  /-- substitution at the type level -/
  sty : (Γ ⟹ Δ) → Ty F Δ  → Ty F Γ
  /-- type level identity -/
  sty_id (σ : Ty F Θ) : sty (𝟙 Θ) σ = σ
  /-- type level composition -/
  sty_comp (σ : Ty F Θ) (f : Γ ⟹ Δ) (g : Δ ⟹ Θ) : sty (f ≫g) σ = sty f (sty g σ)
  /-- substitution at the term level -/
  stm (σ : Ty F Δ) (f : Γ ⟹ Δ) : Tm F Δ σ → Tm F Γ (sty f σ)
  /-- term level identity, depending on a typecast given by sty_id -/
  stm_id (σ : Ty F Θ) (M : Tm F Θ σ) : stm σ (𝟙 Θ) M = cast (by rw [sty_id]) M
  /-- term level composition, depending on a typecast given by sty_comp -/
  stm_comp (σ : Ty F Θ) (M : Tm F Θ σ) (f : Γ ⟹ Δ) (g : Δ ⟹ Θ)
    : stm σ (f ≫g) M = cast (by rw [sty_comp]) (stm (sty g σ) f (stm σ g M))
```

Fig. 5. A typeclass representing the semantic substitution used in categories with families.

even this essential construction raises subtle technical issues that are not central to the semantics. It seems both reasonable and valuable then, that a general framework for producing categorical semantics could relieve the tedium of addressing these issues for each individual semantics.

Additionally, there remains the question of the purpose of formalized categorical semantics outside of their inherent mathematical value. Categorical semantics, and more generally category theory, has as a recurring theme the building of intuition by understanding analogous constructions. A reasonable question then is, given some formalized categorical semantics, is it possible to make use of the categories it specifies in some meaningful way? It seems that it should be possible to work interchangeably with some particular category and its corresponding lambda calculus. Especially given the flexibility of tactics and metaprogramming in modern proof assistants, this presents the possibility of even mid-proof switching between building terms in a lambda calculus to building terms in a category and the ability to search for categorically dual proofs.

ACKNOWLEDGMENTS

Thank you to Ismail Kuru for reviewing a draft of this paper. The Lean Zulip was also an invaluable resource. In particular, I would like to thank Henrik Böving, Yaël Dillies, Kyle Miller, Kim Morrison, and Eric Wieser.

CORE MANUSCRIPTS

[1] Nathan Corbyn, Lukas Heidemann, Nick Hu, Chiara Sarti, Calin Tataru, and Jamie Vicary. 2024. homotopy.io: A Proof Assistant for Finitely-Presented Globular n-Categories. In *9th International Conference on Formal Structures for Computation and Deduction (FSCD 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 299)*, Jakob Rehof (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 30:1–30:26. https://doi.org/10.4230/LIPIcs.FSCD.2024.30 ISSN: 1868-8969.

[2] Jason Gross, Adam Chlipala, and David I. Spivak. 2014. Experience Implementing a Performant Category-Theory Library in Coq. In *Interactive Theorem Proving*, Gerwin Klein and Ruben Gamboa (Eds.). Springer International Publishing, Cham, 275–291. https://doi.org/10.1007/978-3-319-08970-6_18

[3] Benoît Guillemet, Assia Mahboubi, and Matthieu Piquerez. 2024. Machine-Checked Categorical Diagrammatic Reasoning. In *9th International Conference on Formal Structures for Computation and Deduction (FSCD 2024) (Leibniz International*

Proceedings in Informatics (LIPIcs), Vol. 299), Jakob Rehof (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 7:1–7:19. https://doi.org/10.4230/LIPIcs.FSCD.2024.7 ISSN: 1868-8969.

[4] Martin Hofmann. 1997. Syntax and Semantics of Dependent Types. In *Semantics and Logics of Computation* (1 ed.), Andrew M. Pitts and P. Dybjer (Eds.). Cambridge University Press, 79–130. https://doi.org/10.1017/CBO9780511526619.004

[5] Jason Z. S. Hu and Jacques Carette. 2021. Formalizing category theory in Agda. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, Virtual Denmark, 327–342. https://doi.org/10.1145/3437992.3439922

[6] A. M. Pitts. 1987. Polymorphism is set theoretic, constructively. In *Category Theory and Computer Science*, David H. Pitt, Axel Poigné, and David E. Rydeheard (Eds.). Springer, Berlin, Heidelberg, 12–39. https://doi.org/10.1007/3-540-18508-9_18

[7] R. A. G. Seely. 1984. Locally cartesian closed categories and type theory. *Mathematical Proceedings of the Cambridge Philosophical Society* 95, 1 (Jan. 1984), 33–48. https://doi.org/10.1017/S0305004100061284

[8] R. A. G. Seely. 1987. Categorical semantics for higher order polymorphic lambda calculus. *Journal of Symbolic Logic* 52, 4 (Dec. 1987), 969–989. https://doi.org/10.2307/2273831

[9] Bhakti Shah, William Spencer, Laura Zielinski, Ben Caldwell, Adrian Lehmann, and Robert Rand. 2024. ViCAR: Visualizing Categories with Automated Rewriting in Coq. https://doi.org/10.48550/arXiv.2404.08163 arXiv:2404.08163 [cs, math].

ADDITIONAL REFERENCES

[10] 2024. Beluga-lang/Beluga. https://github.com/Beluga-lang/Beluga original-date: 2015-05-14T14:00:29Z.

[11] Musa Al-hassy, Jacques Carette, and Wolfram Kahl. 2019. A language feature to unbundle data at will (short paper). In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. ACM, Athens Greece, 14–19. https://doi.org/10.1145/3357765.3359523

[12] Samuel Arsac, Russell Harmer, and Damien Pous. 2025. Formalizing adhesive category theory in Rocq. (2025).

[13] Andrea Asperti and Simone Martini. 1992. Categorical models of polymorphism. *Information and Computation* 99, 1 (July 1992), 1–79. https://doi.org/10.1016/0890-5401(92)90024-A

[14] David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. 2014. Abella: A System for Reasoning about Relational Specifications. *Journal of Formalized Reasoning* 7, 2 (Dec. 2014), 1–89. https://doi.org/10.6092/issn.1972-5787/4650 Number: 2.

[15] James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. 2019. System F in Agda, for Fun and Profit. In *Mathematics of Program Construction*, Graham Hutton (Ed.). Vol. 11825. Springer International Publishing, Cham, 255–297. https://doi.org/10.1007/978-3-030-33636-3_10 Series Title: Lecture Notes in Computer Science.

[16] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. 2020. Hierarchy Builder: Algebraic hierarchies Made Easy in Coq with Elpi (System Description). *LIPIcs, Volume 167, FSCD 2020* 167 (2020), 34:1–34:21. https://doi.org/10.4230/LIPICS.FSCD.2020.34 Artwork Size: 21 pages, 674766 bytes ISBN: 9783959771559 Medium: application/pdf Publisher: Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[17] Roy L. Crole. 1994. *Categories for Types*. Cambridge University Press, Cambridge. https://doi.org/10.1017/CBO9781139172707

[18] P.-L. Curien. 1993. Substitution up to Isomorphism. *Fundamenta Informaticae* 19, 1-2 (July 1993), 51–85. https://doi.org/10.3233/FI-1993-191-204

[19] Pierre-Louis Curien, Richard Garner, and Martin Hofmann. 2014. Revisiting the categorical interpretation of dependent type theory. *Theoretical Computer Science* 546 (Aug. 2014), 99–119. https://doi.org/10.1016/j.tcs.2014.03.003

[20] V. C. V. De Paiva. 1989. The Dialectica categories. In *Contemporary Mathematics*, John W. Gray and Andre Scedrov (Eds.). Vol. 92. American Mathematical Society, Providence, Rhode Island, 47–62. https://doi.org/10.1090/conm/092/1003194

[21] Peter Dybjer. 1996. Internal type theory. In *Types for Proofs and Programs*, Stefano Berardi and Mario Coppo (Eds.). Springer, Berlin, Heidelberg, 120–134. https://doi.org/10.1007/3-540-61780-9_66

[22] Samuel Eilenberg and Saunders MacLane. 1945. General theory of natural equivalences. *Trans. Amer. Math. Soc.* 58 (1945), 231–294. https://doi.org/10.1090/S0002-9947-1945-0013131-6

[23] Daniel Gratzer and Jonathan Sterling. 2021. Syntactic categories for dependent type theory: sketching and adequacy. https://doi.org/10.48550/arXiv.2012.10783 arXiv:2012.10783 [cs].

[24] Martin Hofmann. 1995. On the interpretation of type theory in locally cartesian closed categories. In *Computer Science Logic*, Leszek Pacholski and Jerzy Tiuryn (Eds.). Springer, Berlin, Heidelberg, 427–441. https://doi.org/10.1007/BFb0022273

[25] André Joyal and Ross Street. 1991. The geometry of tensor calculus, I. *Advances in Mathematics* 88, 1 (July 1991), 55–112. https://doi.org/10.1016/0001-8708(91)90003-P

[26] Nikolai Kudasov, Abdelrahman Abounegm, and Danila Danko. 2024. Rzk: a proof assistant for synthetic \infty\-categories. https://github.com/rzk-lang/rzk original-date: 2020-11-26T10:28:20Z.

[27] Jean-Pierre Marquis and Gonzalo Reyes. 2004. The history of categorical logic: 1963-1977. (2004). https://philarchive.org/archive/MARTHO-8

[28] The mathlib Community. 2020. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 367–381. https://doi.org/10.1145/3372885.3373824

[29] Wojciech Nawrocki, Edward W. Ayers, and Gabriel Ebner. 2023. An Extensible User Interface for Lean 4. In *DROPS-IDN/v2/document/10.4230/LIPIcs.ITP.2023.24*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ITP.2023.24

[30] Benjamin C. Pierce. 1991. *Basic category theory for computer scientists*. MIT Press, Cambridge, MA, USA.

[31] Benjamin C. Pierce. 2004. *Advanced Topics in Types and Programming Languages*. The MIT Press.

[32] John C. Reynolds. 1984. *Polymorphism is not set-theoretic*. report. INRIA. https://inria.hal.science/inria-00076261

[33] Philip Wadler. 1989. Theorems for free!. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture (FPCA '89)*. Association for Computing Machinery, New York, NY, USA, 347–359. https://doi.org/10.1145/99370.99404

[34] Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2022. *Programming Language Foundations in Agda*. https://plfa.inf.ed.ac.uk/22.08/

[35] Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine, and Keenan Crane. 2020. Penrose: from mathematical notation to beautiful diagrams. *ACM Trans. Graph.* 39, 4 (Aug. 2020), 144:144:1–144:144:16. https://doi.org/10.1145/3386569.3392375