Categorical Semantics for Type Theories

JASON Z.S. HU, McGill University, Canada

As the growing number of varieties and features of type theories, type theoretical study has become more and more complex. From the mere syntax, it is very difficult for researchers to connect two existing systems or work on new type theories. Category theory is a library of mathematical concepts and has been popular in the community to study the semantics for programming languages. In this report, we survey category theory and various categorical models of programming languages with a range of typical features.

CCS Concepts: • **Theory of computation → Type theory**; *Higher order logic*; *Constructive mathematics*.

Additional Key Words and Phrases: Category Theory, Categorical Semantics, Type Theory, Logic, Dependent Types

ACM Reference Format:
Jason Z.S. Hu. 2020. Categorical Semantics for Type Theories. 1, 1 (August 2020), 39 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

As software grows larger and larger, software communities are looking for more disciplined ways to software development. The research community has built many useful tools to ensure the correctness of programs, e.g. model checking [Clarke et al. 2001], static analysis [Distefano et al. 2019], deductive verification based on axiomatic semantics of the programs [Leino 2010; Signoles 2009], and so on. Despite that these techniques have achieved large degrees of successes in various applications like aerospace, embedded systems and cloud computing, the majority of programmers do not benefit from them. The fundamental reasons here are that most programmers are not aware of these tools, and that the programmers are not forced to use them even if they are aware.

In contrast, type systems provide a lightweight method to reason about programs and are supported widely in many languages, such as Java, C#, Scala, Rust, etc. There are a number advantages in using types. First, types simply reflect people's intuition on data structures. Second, types are a builtin feature in many programming languages which programmers have to live with. In a programming language with types, a type checker checks for the consistency of types based on the information collected from a program. In case of an inconsistency, the compiler stops the compilation at that point and prints error messages to the programmers. In order to obtain an executable, the programmers must comply with the requirement of the type system and fix the program until it passes the type checking phase. Last, types are not only disciplined, but also serves as a medium to assist programmers. The research community has built many other useful features around types: type inference which allows programmers to avoid elaborate type ascriptions, type-based synthesis which synthesizes programs based on type information, and so on. As a consequence, type-based disciplined software development is closer to the heart of problems and has a stronger impact to programmers than all other techniques mentioned above.

Over decades of research and development, types have been seen great successes in many domains. In the industry, Twitter uses Scala to rewrite their core functionalities which was originally written in Ruby. In the research community, Leroy [2009] and collaborators use a very powerful type system to develop a bug-free optimizing C compiler. Jung et al. [2018] join both paths and target to prove

Author's address: Jason Z.S. Hu, School of Computer Science, McGill University, 3480 University St., Montréal, Quebec, H3A 0E9, Canada, zhong.s.hu@mail.mcgill.ca.

2020. XXXX-XXXX/2020/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the correctness of the standard library of the Rust programming language. Some type systems are so strong that they can serve as mathematical foundations and are used by the mathematical community for proof-reading their mathematical proofs [Cohen and Mahboubi 2012; Krebbers and Spitters 2011; Univalent Foundations Program 2013]. These successes have shown that types are a practical, scalable and flexible solution to a wide spectrum of problems.

Under the hood, however, research on the underlying foundations of type systems has become very complex. Over the last few decades, many type systems are designed for different purposes: general program safety, security, privacy, staged programming, etc. Some systems target strong expressive power by introducing dependent types and various modalities. This large variety makes it difficult to look into these features one by one and understand the relations between these type systems. To tackle this complexity, we need a unified way to discuss different type theories and reveal their mathematical structures and internal connections.

In this report, we survey one popular and effective way, categorical semantics. Category semantics is a collection of methods which define meanings of types and programs in category theory. Compared to other mathematical theories, category theory has a relatively short history, but has been actively developed since its appearance. Intuitively, category theory is a "library" of mathematics. Each category is a "module" generally characterizing some mathematical properties. In categorical semantics, we "instantiate" some category with a type system. As a result, many properties of the type system become immediately available as ones of the category. Moreover, its connections to other type systems are obtained by observing the structures of the categories which they live in. In this way, category theory serves as a common platform for type theoretical study and hence categorical semantics is a more systematic and more modular method for theoretical study than looking into each feature in an "ad hoc" manner.

In this survey, we will introduce the basics of category theory and categorical semantics, as well as some latest results in the type theoretical domain. Detailed structure of the survey is the following: in Section 2, we briefly introduce some basic category theory. In Section 3, we make a historical remark and discuss why we would like to apply categorical approach to type theoretical study. In Section 4, we give a simple and standard example for the categorical semantics of simply typed λ calculus. In Section 5, we extend the calculus with dependent types as well as introduce a suitable categorical model for it. Section 6 discusses more recent results in categorical semantics related to hierarchies of universes and modality in type theory and outlines related research problems. Finally we conclude in Section 7.

2 BASIC CATEGORY THEORY

Category theory [Awodey 2010; MacLane 1971] is a branch of mathematics studying an abstract kind of mappings, called morphisms, and their algebraic relations. Intuitively, category theory is similar to algebraic theories like groups and rings which employ an axiomatic approach to structures, except that category theory is more general. Originally designed for algebraic geometry, category theory has found a large number of applications in other branches of mathematics and in other fields, including analytic philosophy, linguistics, and computer science. Its generality serves as a common language for investigating connections between different concepts in different areas. Category theory is also applied in the field of programming languages and many profound connections between categories and type theories have been found, as to be introduced later in this report. In the interest of self-containment, we will first go over some basic concepts in category theory.

2.1 Categories

Formally, a definition in category theory consists of two parts: data and axioms. Data records what to manipulate and axioms record how to manipulate. In category theory, the most basic definition is the one of a category:

Definition 2.1. A category C consists of the following data:

(1) a collection of objects C_0,
(2) between any two objects $A, B \in C_0$, a collection of morphisms, $C[X, Y]$, whose members are denoted by $f \in C[A, B]$ or $f : A \Rightarrow B$,
(3) an identity morphism $1_A : A \Rightarrow B$ for each object A, and
(4) for morphisms $f : X \Rightarrow Y$ and $g : Y \Rightarrow Z$, the composed morphism $g \circ f : X \Rightarrow Z$.

The data must satisfy the following axioms:

(1) identity: for any morphism $f : X \Rightarrow Y$, $1_Y \circ f = f = f \circ 1_X$,
(2) associativity: for any morphisms $f : Z \Rightarrow W, g : Y \Rightarrow Z, h : X \Rightarrow Y$, $(f \circ g) \circ h = f \circ (g \circ h)$.

Examples for categories. We can spot various categories by identifying their corresponding objects and morphisms and proving the axioms. For example, given any preorder[1] set P, we let each $x \in P$ be an object and for each $x, y \in P$ such that $x \leq y$, we let a unique morphism between them. Thus, identity morphisms correspond to reflexivity of the preorder and composition corresponds to transitivity. The axioms are satisfied trivially: between any pair of objects, there is at most one morphism. Thus all preorder sets form categories. In the case of natural numbers, we have the following diagram:

$$0 \longrightarrow 1 \longrightarrow 2 \longrightarrow 3 \longrightarrow \cdots$$

We often use diagrams to represent (part of) a category. In a diagram, nodes are objects and edges are morphisms.

Another example for a category is the collection of all sets. In the category of all sets, $\mathcal{S}et$, objects are sets and morphisms are functions between sets. Identity morphisms are identity functions and the composition is the usual function composition. Then it is easy to show that both axioms hold.

A Non-example. Though definition of categories is quite flexible, not necessarily all mathematical structures form categories. Consider the (strict) less-than relation of natural numbers. If we let the relation to be the morphism, then since less-than is irreflexive, we are not able to define an identity morphism, and thus we fail to obtain a categorical structure in this case.

Sizes of collections. In Definition 2.1, we intentionally leave the collections for objects and morphisms vague; specifically, we do not specify whether they are *sets* or *classes*. Just like set theory, in category theory, we are also subject to the problem of the kind of collections that we choose to work with. In the previous examples, the preorder categories are constructed from sets (as they are given) and there is at most one morphism between two objects, which fits in a set. When both objects and morphisms fit in sets, the category is *small*.

However, we do not always work with small categories. The category of all sets, $\mathcal{S}et$, is *not* small, because the collection of objects must contain all sets and thus is too "large". However, all functions between any two given sets can indeed be contained in a set. In this case, the category is *locally small*. When morphisms between two objects fit in a set, we call the set *Hom-set*.

Though in computer science, our problem domain very often fits in small categories, set theory is an important device for defining semantics. Thus $\mathcal{S}et$ is one essential and exceptional large category which we often encounter.

[1] A preorder is a reflexive and transitive relation.

2.2 Category Theory for Categories

So far, we are only concerned about sets and categories. Nonetheless, we can already build up a fair amount of complexity. Since category theory is constructed to study structures and a category itself is a structure, we should be able to apply "category theory on top of itself". This gives us the category of all small categories, $\mathcal{C}at$.

Objects in $\mathcal{C}at$ are surely all small categories. What about the morphisms? Recall that in an algebraic theory, e.g. group theory, we are only interested in *homomorphisms*, structure-preserving mappings between groups. For categories, we also have a similar concept:

Definition 2.2. A functor $F : \mathcal{C} \Rightarrow \mathcal{D}$ from category $\mathcal{C}$ to $\mathcal{D}$ has the following data:

(1) a mapping F_0 from objects in $\mathcal{C}$ to those in $\mathcal{D}$, and
(2) for $A, B \in \mathcal{C}$, a mapping F_1 from the Hom-set $\mathcal{C}[A, B]$ to the Hom-set $\mathcal{D}[F_0(A), F_0(B)]$.

We often omit the subscripts of F when they can be inferred from the context. The following axioms are required:

(1) identity: for an object A in $\mathcal{C}$, $F(1_A) = 1_{F(A)}$, and
(2) homomorphism: for morphisms $f : X \Rightarrow Y, g : Y \Rightarrow Z$ in $\mathcal{C}$, $F(g \circ f) = F(g) \circ F(f)$.

Functors between small categories are morphisms in $\mathcal{C}at$. An example for functors is the identity functors: given a category $\mathcal{C}$, both F_0 and F_1 map an input to itself. Functor composition is done by composing F_0 and F_1. The axioms of $\mathcal{C}at$ can be shown by following the definitions.

By letting the categories be the objects, we see the generality of category theory: it simply provides a common language for discussing various structures. We can push this idea further by considering functors as objects. Given fixed categories $\mathcal{C}$ and $\mathcal{D}$, a functor category has functors between $\mathcal{C}$ and $\mathcal{D}$ as objects and *natural transformations* as morphisms.

Definition 2.3. Given two functors $F, G : \mathcal{C} \Rightarrow \mathcal{D}$, a natural transformation $\alpha : F \Rightarrow G$ between them has components as data, which map each object $X \in \mathcal{C}_0$ to a morphism $F(X) \Rightarrow G(X)$ in $\mathcal{D}$, denoted by α_X, so that for each morphism $f : X \Rightarrow Y$ in $\mathcal{C}$, the following diagram commutes (naturality condition):

$$\begin{array}{ccc} F(X) & \xrightarrow{\alpha_X} & G(X) \\ \downarrow{\scriptstyle F(f)} & & \downarrow{\scriptstyle G(f)} \\ F(Y) & \xrightarrow{\alpha_Y} & G(Y) \end{array}$$

Here, we use a *commutative diagram*. When a diagram commutes, morphisms composed by different paths with the same end points are equal. In this case, the diagram represent the following equality:

$$\alpha_Y \circ F(f) = G(f) \circ \alpha_X$$

We often use commutative diagrams to concisely and compactly represent equalities.

It is easy to show that functors with natural transformations as morphisms do form a category. Identify morphisms are identity natural transformations, and the composition is the composition of components. Given three functors F, G, H and two natural transformations $\alpha : F \Rightarrow G$ and $\beta : G \Rightarrow H$, the naturality of composition holds due to the outermost rectangle:

$$\begin{array}{ccccc} F(X) & \xrightarrow{\alpha_X} & G(X) & \xrightarrow{\beta_X} & H(X) \\ \downarrow{\scriptstyle F(f)} & & \downarrow{\scriptstyle G(f)} & & \downarrow{\scriptstyle H(f)} \\ F(Y) & \xrightarrow{\alpha_Y} & G(Y) & \xrightarrow{\beta_Y} & H(Y) \end{array}$$

Two squares commute due to the naturality of α and β, and the rectangle commutes simply by composing the squares. Since the components are simply morphisms, the axioms of a category are directly inherited from $\mathcal{D}$. This verifies the functor category.

From the previous constructions, we start to see why category theory is powerful. It provides a very general framework which is capable of internalizing ideas of the theory itself and helps discover new concepts in a very structured way.

2.3 Categories with Structures

In the previous sections, we discussed some basic constructions in category theory. Though the theory is already quite rich by only considering sets and categories, the theory would not be interesting enough if all it does is to "bite its own tail". Very often, we would want to know a little more about the category that we are working with. This is usually achieved by requiring some additional structures in a category and these structures link to concepts in other areas in categorical ways. For example, we can characterize cartesian products based on the following definition:

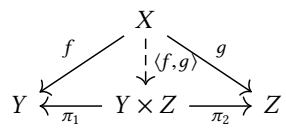
Definition 2.4. A category C has binary cartesian products if it is equipped with additional data:

(1) a mapping $\times$ from two objects to one (cartesian products),
(2) a mapping $\langle -, - \rangle$ of two morphisms, so that for each $f : X \Rightarrow Y$ and $g : X \Rightarrow Z$, $\langle f, g \rangle : X \Rightarrow Y \times Z$, and
(3) two morphisms $\pi_1 : X \times Y \Rightarrow X$ and $\pi_2 : X \times Y \Rightarrow Y$.

The axioms are:

(1) commutativity: for any morphism f, g, $\pi_1 \circ \langle f, g \rangle = f$ and $\pi_2 \circ \langle f, g \rangle = g$, and
(2) uniqueness: for any morphism $h : X \Rightarrow Y \times Z$, $h = \langle \pi_1 \circ h, \pi_2 \circ h \rangle$.

The definition can be summarized by the following commutative diagram:

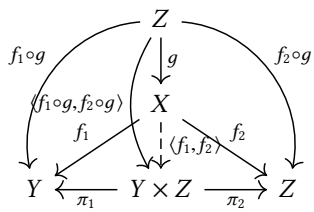


All triangles in this diagram commute. The dashed arrow denotes "unique existence", namely, $\langle f, g \rangle$ uniquely exists. From this definition, we can derive many properties that all products satisfy.

Examples. In a category with binary products, we can prove the following equation:

$$\langle f_1 \circ g, f_2 \circ g \rangle = \langle f_1, f_2 \rangle \circ g$$

represented by this diagram:



First notice that by commutativity and associativity:

$$\pi_i \circ \langle f_1 \circ g, f_2 \circ g \rangle = \pi_i \circ f_i \circ g = \pi_i \circ \langle f_1, f_2 \rangle \circ g$$

Then the target equation follows by uniqueness.

The following morphism combines two morphisms. We overload the notation because we will later prove that it is part of a functor:

$$f \times g \equiv \langle f \circ \pi_1, g \circ \pi_2 \rangle : A \times B \to X \times Y$$

where $f : A \to X$ and $g : B \to Y$. This morphism can be thought of as f and g "in parallel".

We are able to show the following equation:

$$(f \times 1) \circ (1 \times g) = f \times g = (1 \times g) \circ (f \times 1)$$

We prove the first part of the equation and the second part can be proved analogously.

First we apply π_i's to both sides and then apply commutativity and associativity:

$$\begin{aligned} &\pi_1 \circ (f \times 1) \circ (1 \times g) && \pi_2 \circ (f \times 1) \circ (1 \times g) \\ =&f \circ \pi_1 \circ (1 \times g) && =\pi_2 \circ (1 \times g) \\ =&f \circ \pi_1 && =g \circ \pi_2 \end{aligned}$$

Then the target equation is concluded via uniqueness like the first example.

Examples for categories with binary products. We have already mentioned two categories with this structure previously: $\mathcal{S}et$ and $\mathcal{C}at$. In $\mathcal{S}et$, the products are provided by the cartesian products of two sets. π_i's are provided by the projection functions. $\langle -, - \rangle$ is provided by the following definition:

$$\langle f, g \rangle (x) = (f(x), g(x))$$

That is, the outputs of two functions are bundled as a pair.

For $\mathcal{C}at$, this structure is given by the *product category* of two small categories. A product category is formed by putting objects and morphisms in both categories in pairs. Since the input categories are small, the resulting category is also small. π_i's are then functors extracting the objects and morphisms in the corresponding positions. The object and morphism parts of $\langle -, - \rangle$ are implemented as follows:

$$\begin{aligned} \langle F, G \rangle (X) &= (F(X), G(X)) \\ \langle F, G \rangle (f) &= (F(f), G(f)) \end{aligned}$$

for any functors F and G. We can verify that $\langle F, G \rangle$ does form a functor. We write the product category of $\mathcal{C}$ and $\mathcal{D}$ as $\mathcal{C} \times \mathcal{D}$. Here we overload the $\times$ symbol again and its meaning can be disambiguated from the context.

Functoriality of $\times$. In a previous example, we intentionally overloaded the $\times$ symbol for mapping both objects and morphisms because $\times$ possesses a functorial structure (namely $\times$ is a functor). Formally, $\times : \mathcal{C} \times \mathcal{C} \Rightarrow \mathcal{C}$ is a bifunctor, a functor with a product category as its domain. We have defined the data so there are only the axioms for functors to show. For identity, we need to show $1_A \times 1_B = 1_{A \times B}$ for any objects A and B, which is straightforward from uniqueness. For homomorphism, we need to show $(f_1 \times g_1) \circ (f_2 \times g_2) = (f_1 \circ f_2) \times (g_1 \circ g_2)$. This equation can also be easily proved by precomposing π_i's and then apply uniqueness after some computation.

Cartesian categories. In a category with binary products, we obtain a product of any positive number of objects. It is then natural to ask what can serve as a nullary product. This concept is characterized by terminal objects.

Definition 2.5. A terminal object in a category is a special object $\top$ so that for any object X, there must be a unique morphism $! : X \Rightarrow \top$.

This definition is characterized by the following diagram:

$$X \dashrightarrow^{!} \top$$

With a terminal object, we are able to characterize a product of any number of objects.

Definition 2.6. A cartesian category is a category with a terminal object and all binary products.

Examples. For any object X, we have an isomorphism:

$$X \simeq X \times \top$$

That is, there exists two morphisms $f : X \Rightarrow X \times \top$ and $g : X \times \top \Rightarrow X$, such that $f \circ g = 1_{X\times\top}$ and $g \circ f = 1_X$. This isomorphism captures the intuition that the terminal object is a unit element of products. We give their definitions as follows:

$$f \equiv \langle 1_X, ! \rangle$$
$$g \equiv \pi_1$$

$g \circ f = 1_X$ is immediate. $f \circ g = 1_{X\times\top}$ can be proved by composing π_1 and π_2:

$$\begin{aligned} &\pi_1 \circ f \circ g \\ =&1_X \circ \pi_1 \\ =&\pi_1 \end{aligned} \qquad \begin{aligned} &\pi_2 \circ f \circ g \\ =&! \circ \pi_1 \\ =&! \qquad \text{(due to uniqueness of the terminal object)} \\ =&\pi_2 \end{aligned}$$

Then by uniqueness of products, we have $f \circ g = \langle \pi_1, \pi_2 \rangle = 1_{X\times\top}$, which concludes the isomorphism.

Examples for cartesian categories. $\mathcal{S}et$ and $\mathcal{C}at$ are cartesian categories. It remains to show they both have terminal objects. The terminal object of $\mathcal{S}et$ is the singleton set $\{*\}$ for any element $*$. The terminal object of $\mathcal{C}at$ is the category 1: a category with only one object and one morphism; the only morphism is necessarily the identity morphism of the only object.

2.4 Universal Mapping Properties

In the previous section, we introduced two structures: products and terminal objects. Both structures share a similar pattern in their definitions: a morphism exists uniquely for all morphisms (potentially with some conditions). This pattern is called a universal mapping property (UMP). In category theory, UMPs are a canonical way to characterize the "best" entities of some concepts. What the "best" means is left unspecified until we instantiate the concept in a concrete category. Consider categorical products in $\mathcal{S}et$, which characterize cartesian products. It is undoubted that a product characterizes itself the best.

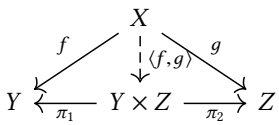
Now consider natural numbers as a preorder category introduced in Section 2.1. Given two numbers m and n, what should be their categorical product? By definition, $m \times n$ must be the "best" number which has morphisms into m and n. Recall that $a \Rightarrow b$ in a preorder category denotes $a \leq b$. That means $m \times n$ has to be the largest number that is smaller than both m and n and thus $\min(m, n)$; otherwise, $m \times n$ would be a number $l > \min(m, n)$ and smaller than both m and n which is impossible. Then what is a terminal object of natural numbers? Every number must have a morphism into it, meaning that it is the biggest number of all, which does not exist for natural numbers. If we consider only natural numbers up to a fixed number u, then a terminal object does exist and it is u. Previously, we proved the isomorphism $X \simeq X \times \top$, which corresponds to $m = \min(m, u)$ in this case. Via the UMPs of products and terminal object, we obtain a fact about many preorder categories without considering them specifically. UMPs introduce a great generality to the definitions in category theory and allows us to speak about some general truths without being too specific to a problem domain.

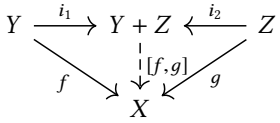
2.5 Principle of Duality

When making mathematical statements, we often encounter terminologies that are somewhat the opposite to another: a set and its complement, maximum and minimum, supremum and infimum, and so on. In category theory, this phenomenon is captured by the *Principle of Duality*. Intuitively, duality gives us a "free" dual theorem from the original one. The dual theorem is derived in the *opposite category*:

Definition 2.7. The opposite category of a category C, denoted by C^{op}, has the same objects as C. Morphisms in C^{op} are defined by flipping the directions of those in C.

For instance, given a morphism $f : A \Rightarrow B$ in C, the *same* morphism also exists in C^{op}, except that it becomes $f : B \Rightarrow A$. The direction of morphism composition needs to be flipped too. $g \circ f$ in C becomes $f \circ g$ in C^{op}. In the opposite category, all UMP definitions correspond to a dual concept, e.g. for products:





The diagram on the left shows the definition of a product. By applying duality, we obtain the diagram on the right in the opposite category, defining a *coproduct*[2]. The definition of coproducts is also a UMP, which asserts the unique existence of a morphism $[f, g] : Y + Z \Rightarrow X$ for $f : Y \Rightarrow X$ and $g : Z \Rightarrow X$. In Section 2.3, we proved the equation $\langle f_1 \circ g, f_2 \circ g \rangle = \langle f_1, f_2 \rangle \circ g$. In the opposite category, it becomes an equation for coproducts due to the principle:

$$[g \circ f_1, g \circ f_2] = g \circ [f_1, f_2]$$

The Principle of Duality assigns us the flexibility of definitions as well as the productivity of deriving facts. This convenience is another advantage provided by category theory.

3 LOGIC, TYPES AND CATEGORIES

3.1 Correspondences of Types

One important (potentially the most important) principle in type theory is Propositions as Types or Curry-Howard Isomorphism [Curry 1934; de Bruijn 1970; Howard 1980; Martin-Löf 1984; Wadler 2015]. This principle describes a connection between two previously separated fields: logic and types. In logic, people think about what and how facts are concluded, while types are a concept in computer science, which emerges much later and naturally carries the notion of computation. It happens that both areas are tightly connected: a program can represent a logical argument, and operations in logic find meaningful correspondences in program execution.

The origins of this principle can be dated back as early as the 1930s. Curry [1934] developed combinatory logic, a logical system which eliminates the need for quantifying variables and was found to serve as a model of computation. Though Curry received half of the credit of the principle, the idea of using *terms* to represent logical proofs is discovered by the intuitionists Brouwer and his student Heyting [1934] and independently Kolmogoroff [1932]. This idea is collectively called the BHK interpretation. In the BHK interpretation, a proof in (intuitionistic) first order logic is inductively defined based on the connectives. For example, for two propositions P and Q, their conjunction $P \wedge Q$ is the pair of the proof of P and the one of Q. The implication $P \rightarrow Q$ is a function which transforms a proof of P into one of Q. Further guidelines were laid by Gentzen [1934, 1969]. In his PhD thesis, Gentzen introduced natural deduction, in which each connective

[2]The prefix co- is frequently used for dual concepts.

has introduction rules and elimination rules. The introduction rules describe how the connective is constructed, while the elimination rules describe how the connective can be used. For example, the following three rules formulates logical conjunctions:

$$\frac{\Gamma \vdash P \ \mathtt{true} \qquad \Gamma \vdash Q \ \mathtt{true}}{\Gamma \vdash P \wedge Q \ \mathtt{true}} \qquad \frac{\Gamma \vdash P \wedge Q \ \mathtt{true}}{\Gamma \vdash P \ \mathtt{true}} \qquad \frac{\Gamma \vdash P \wedge Q \ \mathtt{true}}{\Gamma \vdash Q \ \mathtt{true}}$$

The first rule is the introduction rule. It states that a conjunction is true when two component propositions are true. The later two are the elimination rules and they ensures the truth of components can be extracted from a conjunction. Later systems like intuitionistic type theory [Martin-Löf 1984], Calculus of Constructions [Coquand and Huet 1988] and System F [Girard 1971; Reynolds 1974] employ and/or extend natural deduction in various ways. Natural deduction has become a standard formulation in programming languages research.

The principle was later extended with category theory due to Lambek [1974, 1980, 1985]. In his work, Lambek showed a correspondence between simply typed λ-calculus (STLC) and cartesian closed categories (CCC) [3] and revealed the connection between types and categories. Together with correspondence with logic, the principle became Curry-Howard-Lambek Isomorphism, suggesting a very appealing interconnection among three originally seemingly unrelated fields.

3.2 Syntax and Semantics

Generally speaking, there are two approaches to understand or design a type system: the syntactic point of view and the semantic point of view. In the syntactic point of view, we inherit methods from logic and proof theory which focus our attention on the syntactic structures of a type system. We study many important properties like subject reduction [Wright and Felleisen 1994], cut elimination, subformula property, and many others, simply by looking at the *syntax*. One advantage of syntactic approaches is that the study is usually more direct and easier to understand than semantic approaches as it is usually conducted via induction on some syntactic structures. Moreover, syntactic approaches usually suggest algorithms which can be implemented. However, syntactic approaches do not make any use of the *meanings* of the type system to derive properties. This disadvantage is quite a limitation and makes some properties like normalization very difficult to approach.

Semantic approaches in general are more powerful because they base the discussion of a type system on some mathematical models which exhibits many useful properties. Thus one usually obtains more structures to work with than the mere syntax. For instance, Giarrusso et al. [2020] developed a by-far the most complex dependent type system with subtyping by using a semantic method called step-indexed logical relations, significantly more complex than any former systems designed using syntactic methods. In exchange of the strength of the methods, semantic approaches are usually more heavyweight, as one needs to first develop an intuition of the concepts and then establish relations between syntax and the semantic model.

One classical semantic approach to logic (and thus types) is algebraic logic. In this approach, a logical system is modeled by some algebraic theory (very often related to lattices or semi-lattices). Famous examples include Boolean algebra modeling classical propositional logic [Boole 1854] and Heyting algebra modeling the intuitionistic counterpart [Heyting 1930]. The algebraic approach has a fundamental constraint that all operations are limited in one structured set.

Category theory, contrarily, is more generalized[4]. Given any pair of objects, we still reason about morphisms between them algebraically, but we have the flexibility of considering morphisms

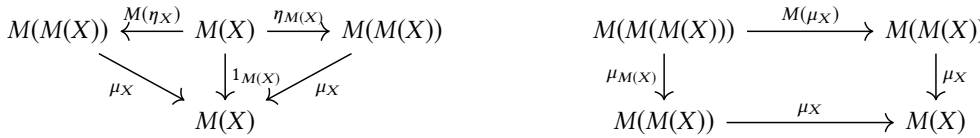
[3] A cartesian closed category is a cartesian category with a closed structure, which will be discussed later.
[4] Indeed, the theory of categories is a generalized algebraic theory as per Cartmell [1986].

between different pairs of objects and switching structures by following functors. In the next few sections, let us how the abundant freedom allows us to model various concepts in computer science.

3.3 Computations as Monads

One of the most impactful results from categorical semantics is probably monads. In his seminal paper, Moggi [1991] introduced a categorical semantics for general computations based on monads. In this work, he showed that many computations (including the side-effectful ones) can be considered as monads.

Definition 3.1. A monad of a category C is an endofunctor $M : C \Rightarrow C$ with two natural transformations $\eta : 1_C \Rightarrow M$ and $\mu : M \circ M \Rightarrow M$. Note that here $M \circ M$ is a functor of M composed with M. The following axioms hold:



The left diagram expresses identity and the right diagram expresses associativity.

Moggi showed that following computations can be modelled by monads among others:

(1) Stateful computations: We can consider program states captured by a set of states S. In a stateful programming language like C or C++, a function receiving A and returning B implicitly involves a global state. Moggi showed that this function can be modelled by $M(A \to B) = S \to ((A \to B) \times S) = (S \to (A \to B)) \times (S \to S)$. Namely, a stateful function has two parts, first computing B based on the state and A and second transforming the state.
(2) Exception: Some programming languages like Java and C# implement exception mechanisms which allow programmers to introduce abrupt breaks of executions at any point of a program. This is again captured by monads by considering a set of exceptional states E. Thus a function $A \to B$ with potential exceptions is modelled as $A \to M(B)$ where $M(B) = B + E$ and $+$ denotes disjoint unions. That is, this function either returns B in a normal execution, or "throws" an exception by returning E.

Monads as a concept have been ubiquitous in functional programming community and adapted by languages like Haskell as a basic design principle for encapsulating computations. It serves as a framework for functional programming languages to handle side effects while stay as pure as possible. In his supplementary materials, Harper [2016] defined a system PCFv which embeds general, potentially nonterminating computations in a terminating language. The idea is to use a monad to separate nonterminating computations from the terminating once. In PCFv, there are two kinds of judgments: $\Gamma \vdash t : T$ is a judgment for a term which must reduce to a value and $\Gamma \vdash m \div T$ is for one representing a general computation. These two judgments are connected by the following rules:

$$\frac{\Gamma \vdash m \div T}{\Gamma \vdash comp(m) : C\ T} \qquad \frac{\Gamma \vdash t : C\ S \qquad \Gamma, x : S \vdash m \div U}{\Gamma \vdash bind(t, x.m) \div U} \qquad \frac{\Gamma \vdash t : T}{\Gamma \vdash ret(t) \div T}$$

In this type system, the type former C can be regarded as a monad. The required natural transformations can be defined as follows:

$$\begin{aligned} \eta_T &: T \to C\ T \\ \eta_T(t) &= comp(ret(t)) \\ \mu_T &: C\ C\ T \to C\ T \end{aligned}$$

$$\mu_T(t) = comp(bind(t, x.bind(x, y.ret(y))))$$

Verifying the axioms requires operational semantics or term equivalence relation which we omit here. In PCFv, C as a monad can be considered as a segregation of two different kinds of computations and provide a mathematical handle for investigation of this type system.

To summarize, monads not only give us guidance in overcoming limitations in functional programming but also helps us to understand and design novel type systems.

3.4 Logic as Adjoint Functors

Compared to the long history of mathematical study of logic, categorical logic is relatively new but quite illuminating. The idea of using category theory to study logic was due to Lawvere. In his influential paper, Lawvere [1969] showed that logical constructs are fundamentally just adjoint functors. This result allows us to capture many logical constructs by using only one categorical concept. Moreover, adjointness provides a strong guarantee about the derived syntactic formulation.

Definition 3.2. A pair of functors $L : \mathcal{C} \Rightarrow \mathcal{D}$ and $R : \mathcal{D} \Rightarrow \mathcal{C}$ are adjoint, written as $L \dashv R$, when the following isomorphism given $X \in \mathcal{C}$ and $Y \in \mathcal{D}$

$$\mathcal{D}[L(X), Y] \simeq \mathcal{C}[X, R(Y)]$$

is natural in X and Y. Naturality in X and Y means that the following diagram commutes given $f : X' \Rightarrow X$ and $g : Y \Rightarrow Y'$:

$$\begin{array}{ccc} \mathcal{D}[L(X), Y] & \xrightarrow{\simeq} & \mathcal{C}[X, R(Y)] \\ \downarrow{\scriptstyle \mathcal{D}[L(f), g]} & & \downarrow{\scriptstyle \mathcal{C}[f, R(g)]} \\ \mathcal{D}[L(X'), Y'] & \xrightarrow{\simeq} & \mathcal{C}[X', R(Y')] \end{array}$$

More compactly, we have as an equivalence judgment:

$$\frac{\mathcal{D}[L(X), Y]}{\mathcal{C}[X, R(Y)]}$$

We call L is the *left adjoint* and R is the *right adjoint*. The left-to-right effect of the isomorphism is *left adjunction* and the other direction is *right adjunction*. As we suggested, many logical constructs can be formulated by adjoint functors. Consider the formulation of conjunctions in natural deduction shown in Section 3.1. We here change the formulation slightly into the adjoint equivalence judgment:

$$\frac{\Gamma \vdash P \ \mathtt{true} \qquad \Gamma \vdash Q \ \mathtt{true}}{\Gamma \vdash P \wedge Q \ \mathtt{true}}$$

This formulation is similar to the introduction rule, except that we use double horizontal line, meaning that it is an equivalence relation. That is, having $P \wedge Q$ is equivalent to having P and Q. Consider a category C where intuitionistic propositional logic resides[5]. We regard $\Gamma \vdash P \ \mathtt{true}$ as a morphism $\Gamma \Rightarrow P$ in the category. This allows us to transform the formulation above into the following more compact forms:

$$\frac{C[\Gamma, P] \times C[\Gamma, Q]}{C[\Gamma, P \wedge Q]} \qquad \Longrightarrow \qquad \frac{C \times C[\Gamma \times \Gamma, P \times Q]}{C[\Gamma, P \wedge Q]}$$

The formulation on the right finds a pair of adjoint functors formulating logical conjunctions. Specifically, the adjunction is between $C \times C$ and C. The left adjoint $L = \Delta$, the diagonal functor

[5]We do not make this precise but it does not affect the discussion here.

which is defined by $\Delta(X) = (X, X)$. The right adjoint R is defined by conjunctions $R(P, Q) = P \wedge Q$. Thus logical conjunctions are modelled jointly by one pair of functors $L \dashv R$.

Lawvere realized that adjoint functors are a very general concept that it can even be used to model stronger logical constructs like universal and existential quantifiers. This can be seen from the following equivalence judgments:

$$\frac{\phi^*(\overline{x}, y) \vdash \psi(\overline{x}, y)}{\phi(\overline{x}) \vdash \forall y.\psi(\overline{x}, y)} \qquad \frac{\exists y.\phi(\overline{x}, y) \vdash \psi(\overline{x})}{\phi(\overline{x}, y) \vdash \psi^*(\overline{x}, y)}$$

Here we operate in the category of first order logic. $\phi(\overline{x})$ represents a first order formula with free variables $\overline{x}$. Given $\phi(\overline{x})$, $-^*$ *weakens* this formula to $\phi^*(\overline{x}, y)$, so that ϕ^* has one more free variable y. We can show that the following adjunctions exist:

$$\exists y \dashv -^* \dashv \forall y$$

We can quite easily see the natural deductive formulation of universal quantification from the adjunction on the left. The introduction rule can be just copied down:

$$\frac{\phi^*(\overline{x}, y) \vdash \psi(\overline{x}, y)}{\phi(\overline{x}) \vdash \forall y.\psi(\overline{x}, y)}$$

The elimination rule is achieved by substitution y with some appropriate term which has only $\overline{x}$ free, say t:

$$\frac{\phi(\overline{x}) \vdash \forall y.\psi(\overline{x}, y)}{\phi(\overline{x}) \vdash \psi(\overline{x}, y)[t/y]}$$

Substituting y in $\phi^*(\overline{x}, y)$ has no effect because y does not occur in $\phi(\overline{x})$, so we just write $\phi(\overline{x})$ instead.

Existential quantifications are not as straightforward. We start from its elimination form:

$$\frac{\chi(\overline{x}) \vdash \exists y.\phi(\overline{x}, y) \qquad \phi(\overline{x}, y) \vdash \psi^*(\overline{x}, y)}{\chi(\overline{x}) \vdash \psi(\overline{x})}$$

One can see that this elimination form is actually obtained by applying the inverse effect of the adjunction. From the elimination form, we can obtain the introduction form by substitution:

$$\frac{\psi(\overline{x}) \vdash \phi(\overline{x}, y)[t/y]}{\psi(\overline{x}) \vdash \exists y.\phi(\overline{x}, y)}$$

Another reason why we use adjoint functors to capture logical constructs is that adjoint functors provide a very strong correctness guarantee.

Theorem 3.3. *Given two pairs of adjoint functors $L \dashv R$ and $L \dashv R'$, then R and R' are isomorphic functors.*

Given two pairs of adjoint functors $L \dashv R$ and $L' \dashv R$, then L and L' are isomorphic functors.

That is, fixing one side of the adjoint functors, if the other side exists, it is uniquely determined up to isomorphism. This theorem has two delightful consequences:

(1) There is only one set of correct formulations of a logical construct, and all correct formulations can be shown equivalent.
(2) No matter what formulation we pick at the end, the mathematical denotation of the logical constructs remains the same (up to isomorphism).

In particular, the first consequence still gives us the type theorists freedom to refine the right rules based on various criteria on the syntactic realm and the second consequence secures the proper categorical semantics.

There have already been many applications of using adjoint functors to find categorical semantics and to find good syntactic rules. For example, Clouston [2018]; Kavvos [2017] gave categorical semantics for a number of modal type theories in Prawitz style and Fitch style, respectively. Birkedal et al. [2020] defined a dependently typed version of Fitch-style modal type theory and modelled it using a dependent generalization of adjunction. We will have more discussions on modal type theories in Section 6.2 after we have more technical setup.

3.5 Other Applications of Categorical Semantics

In the previous sections, we gave some overview on how categorical semantics are applied to research in type theories and how useful category theory is in modelling logic and type theories.

In general, we want to apply categorical methods for two reasons. One is that we want to understand a given type system through categorical lenses. The early work was done by Lawvere [1963], who explored *functorial semantics*, stating a dual relation between syntax and semantics. In functorial semantics, models form categories and are connected by functors. Among all models, the model defined by the syntactic rules is the best in that there is a functor from the syntactic model to other semantic model. That we are able to speak about all models is a particular strength of category theory. This allows a syntactic system and its corresponding categorical model to be used interchangeably. This kind of correspondences have been discovered for many type theories, including simply typed lambda calculus and cartesian closed categories [Lambek 1974, 1980, 1985], first order logic and hyperdoctrines [Seely 1983], extensional dependent type theory and locally cartesian closed categories [Seely 1984], etc.

Another reason for applying categorical methods is that we want to let category theory to guide us to extend an existing type theory or to design a new type theory. Hofmann [1999] presented a type system, SLR, in which all first order functions are polynomial-time computable. This type system was motivated by combining some affine linear category with a comonadic modality. We have also mentioned some work on comonadic modal type theory towards the end of the previous section. Another work falling into a similar line is Pientka and Schöpp [2020], who developed a connection between a presheaf model in Hofmann [1997] and a simply typed version of a type system in Pientka et al. [2019]. In this work, the categorical model is not quite the same as the ones in Birkedal et al. [2020]; Clouston [2018]; Kavvos [2017] and therefore it is interesting to compare the different models in these papers. We will have more discussion in Section 6.2. Another recent line of important work is related to homotopy type theory (HoTT) [Univalent Foundations Program 2013]. Homotopy type theory is designed to bring formal mathematics closer to informal mathematics by empowering the *propositional equality* with equivalences. Homotopy type theory extends Martin-Löf type theory [Martin-Löf 1984] by allowing equality to have nontrivial computational contents. A recent profound result from Cohen et al. [2017] described a constructive formulation for HoTT motivated by some category of cubical sets, solving a long standing open problem. More discussion on the cubical set models can be found in Cavallo et al. [2020]. Shulman [2018] extended HoTT with modalities and show that the modalities form adjoint functors.

4 SIMPLY TYPED λ CALCULUS

Due to the principle of Propositions as Types, we know that simply typed λ calculus (STLC) corresponds to intuitionistic propositional logic. Essentially, STLC adds computational contents to logic. Correspondence between STLC and cartesian closed categories (CCC) is further added by Lambek [1974, 1980, 1985] from a categorical angle. This correspondence serves a good example

for categorical semantics in type theory: we prove a soundness theorem to show a particular kind of categories modeling a type theory and a completeness theorem to show that the type theory is itself among those categories. In Section 6.2, we will see that this correspondence is extended with the necessity modality.

4.1 Definition of STLC

STLC is defined by its syntax and judgments. Its syntax defines what items compose the language and the judgments specify what items are meaningful. In STLC, the syntax is extremely easy: there are types, terms, and contexts defined as follows. For the scope of this report, let us only consider a version STLC with a unit type $*$, product types $\times$, and (simple) function types $\to$.

$$
\begin{aligned}
x, y, z & && \text{variables} \\
S, T, U &:= * \mid S \times U \mid S \to U && \text{types} \\
s, t, u &:= () \mid (s, u) \mid \pi_1(t) \mid \pi_2(t) \mid \lambda x.t \mid s\ u && \text{terms} \\
\Gamma, \Delta &:= \cdot \mid \Gamma, x : T && \text{contexts}
\end{aligned}
$$

Following Barendregt [1985], we assume α equivalence which regards terms equal if they only differ by variable names. Whenever we refer to a name, we silently assume that the name is different from all used names. Moreover, contexts bind variables to types. The set of all variables in a context Γ is its domain, denoted as $dom(\Gamma)$. Due to the same spirit as α equivalence, for a context to be well-formed, variables in the domain must be unique.

A typing judgment in STLC has the form $\Gamma \vdash t : T$, meaning that the term t has type T in the context Γ. Typing judgments define well-formed terms:

$$
\frac{}{\Gamma \vdash () : *}\ *\text{-I} \qquad \frac{\Gamma \vdash s : S \qquad \Gamma \vdash u : U}{\Gamma \vdash (s, u) : S \times U}\ \times\text{-I} \qquad \frac{\Gamma \vdash t : S \times U}{\Gamma \vdash \pi_1(t) : S}\ \times\text{-E}_1 \qquad \frac{\Gamma \vdash t : S \times U}{\Gamma \vdash \pi_2(t) : U}\ \times\text{-E}_2
$$

$$
\frac{\Gamma, x : S \vdash t : U}{\Gamma \vdash \lambda x.t : S \to U}\ \to\text{-I} \qquad \frac{\Gamma \vdash t : S \to U \qquad \Gamma \vdash s : S}{\Gamma \vdash t\ s : U}\ \to\text{-E} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \textsc{var}
$$

Following the style of natural deduction, there are two kinds of typing rules: introduction and elimination. Introduction rules describe how to construct a term of a type while elimination rules describe how to use a term of a type. If we interpret the unit type as the trivial truth, the product types as conjunctions, and function types as implications, we can see that STLC does correspond to propositional logic.

Another kind of judgments of STLC is $\Gamma \vdash s = u : T$, denoting the equivalence between well-formed terms s and u of type T in the context Γ. This equivalence relation satisfies standard equational properties like reflexivity, symmetricity, transitivity, and congruence which We omit here. Additionally, the equivalence rules characterize how the introduction and elimination rules of each kind of types interact.

$$\frac{\Gamma \vdash t : *}{\Gamma \vdash t = () : *}\ *\text{-}\eta \qquad \frac{\Gamma \vdash s : S \qquad \Gamma \vdash u : U}{\Gamma \vdash \pi_1(s,u) = s : S}\ \times\text{-}\beta_1 \qquad \frac{\Gamma \vdash s : S \qquad \Gamma \vdash u : U}{\Gamma \vdash \pi_2(s,u) = u : U}\ \times\text{-}\beta_2$$

$$\frac{\Gamma \vdash t : S \times U}{\Gamma \vdash t = (\pi_1(t), \pi_2(t)) : S \times U}\ \times\text{-}\eta \qquad \frac{\Gamma, x : S \vdash u : U}{\Gamma \vdash (\lambda x.u)\ s = u[s/x] : U}\ \to\text{-}\beta$$

$$\frac{\Gamma \vdash t : S \to U}{\Gamma \vdash t = \lambda x.t\ x : S \to U}\ \to\text{-}\eta$$

In STLC, there are two kinds of equivalence rules: β reduction rules and η expansion rules. Later on, we will see that β rules correspond to commutativity in category theory and η rules correspond to uniqueness. In the $\to$-β rule, $u[s/x]$ denotes capture-free substitution, which substitutes all occurrences of x in u with s, with potential renaming of variables to avoid clashes.

4.2 Cartesian Closed Categories

In Section 2.3, we have discussed cartesian categories, which generally characterize finite products. In order to fully characterize STLC, we need an additional structure, exponentials:

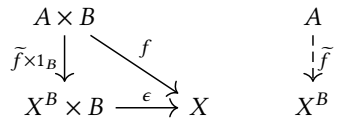
Definition 4.1. A cartesian category $\mathcal{C}$ has exponentials, if it has the following data:

(1) for objects X and B, an object X^B as their exponential,
(2) for any morphism $f : A \times B \to X$, a morphism $\widetilde{f} : A \Rightarrow X^B$ as its transpose, and
(3) an evaluation morphism $\epsilon : X^B \times B \Rightarrow X$.

So that the following axioms hold:

(1) commutativity: for $f : A \times B \Rightarrow X$, $\epsilon \circ (\widetilde{f} \times 1_B) = f$,
(2) uniqueness: for $g : A \Rightarrow X^B$, $\widetilde{\epsilon \circ (g \times 1_B)} = g$.

The definition is summarized by the following commutative diagram:



Exponentials are used to characterize functions. The idea is to consider X^B as an internal representation of a morphism $B \Rightarrow X$.

Definition 4.2. A cartesian closed category (CCC) is a cartesian category with exponentials.

Examples. One example for a CCC is the category of sets, $\mathcal{S}et$. We have shown $\mathcal{S}et$ is cartesian, so we just need to show $\mathcal{S}et$ has exponentials. With no surprise, given two sets, X and B, X^B in fact is the set of all functions from B to X. For $f : A \times B \Rightarrow X$, $\widetilde{f}$ is *currying*, defined as follows:

$$\widetilde{f}(a)(b) = f(a,b)$$

where $a \in A$ and $b \in B$. Finally the evaluation morphism ϵ simply evaluates a function with an argument:

$$\epsilon(f,x) = f(x)$$

The axioms can be proved by simply expanding the definitions.

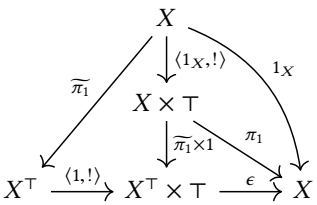
One can show many properties of exponentials. With a terminal object, we expect the following isomorphism for any object X:

$$X^{\top} \simeq X$$

That is, $\top$ is the exponent unit. The isomorphism consists of the following morphisms.

$$X \xrightarrow{\widetilde{\pi_1}} X^{\top} \qquad X^{\top} \xrightarrow{\langle 1,! \rangle} X^{\top} \times \top \xrightarrow{\epsilon} X$$

For conciseness, we only show $\epsilon \circ \langle 1, ! \rangle \circ \widetilde{\pi_1} = 1_X$:



The target is to show that the outermost diagram commutes. The triangle on the left commutes due to properties of products. The diagram on the right commutes due to commutativity of exponentials. The outermost diagram commutes by combining both diagrams.

We omit the proof of $\widetilde{\pi_1} \circ \epsilon \circ \langle 1, ! \rangle = 1_{X^{\top}}$. We can prove this equation by applying uniqueness of exponentials. This concludes the isomorphism.

4.3 Categorical Semantics for STLC

One crucial observation Lambek made was that given a CCC $\mathcal{C}$, one can interpret typing judgments as its morphisms and equivalence judgments as equality between the morphisms. More concretely, there is an interpretation function $[\![-]\!]$ from STLC to $\mathcal{C}$ so that

Theorem 4.3. *If $\Gamma \vdash t : T$, then there is a morphism $[\![\Gamma \vdash t : T]\!] : [\![\Gamma]\!] \Rightarrow [\![T]\!]$.*
If $\Gamma \vdash s = u : T$, then $[\![\Gamma \vdash s : T]\!] = [\![\Gamma \vdash u : T]\!]$.

The interpretation of types and contexts are defined recursively:

$$\begin{aligned} [\![*]\!] &= \top \\ [\![S \times U]\!] &= [\![S]\!] \times [\![U]\!] \\ [\![S \to U]\!] &= [\![U]\!]^{[\![S]\!]} \end{aligned} \qquad \begin{aligned} [\![\cdot]\!] &= \top \\ [\![\Gamma, x : T]\!] &= [\![\Gamma]\!] \times [\![T]\!] \end{aligned}$$

From the interpretation, we see that the types do become corresponding categorical structures as promised. Contexts are modelled by products of types. We then proceed to interpreting the well-formed types:

$$\begin{aligned} [\![\Gamma \vdash () : *]\!] &= ! \\ [\![\Gamma \vdash (s,u) : S \times U]\!] &= \langle [\![\Gamma \vdash s : S]\!], [\![\Gamma \vdash u : U]\!] \rangle \\ [\![\Gamma \vdash \pi_1(t) : S]\!] &= \pi_1 \circ [\![\Gamma \vdash t : S \times U]\!] \\ [\![\Gamma \vdash \pi_2(t) : S]\!] &= \pi_2 \circ [\![\Gamma \vdash t : S \times U]\!] \\ [\![\Gamma \vdash \lambda x.t : S \to U]\!] &= \widetilde{[\![\Gamma, x : S \vdash t : U]\!]} \\ [\![\Gamma \vdash t\ s : U]\!] &= \epsilon \circ \langle [\![\Gamma \vdash t : S \to U]\!], [\![\Gamma \vdash s : S]\!] \rangle \\ [\![\Gamma \vdash x : T]\!] &= \pi_2 \circ \pi_1^k \qquad \text{where } \Gamma = \Gamma_1, x : T, \Gamma_2 \text{ and } |\Gamma_2| = k \end{aligned}$$

This interpretation ensures the interpreted morphisms do live in the Hom-set $[\![\Gamma]\!] \Rightarrow [\![T]\!]$. Finally, in order to verify equivalence judgments, we need to interpret substitutions in CCC. We can prove

the following equation:

$$[\![\Gamma \vdash u[s/x] : U]\!] = [\![\Gamma, x : S \vdash u : U]\!] \circ \langle 1_{[\![\Gamma]\!]}, [\![\Gamma \vdash s : S]\!] \rangle$$

Thus the corresponding equation of the $\rightarrow$-β rule can be examined as follows:

$$\begin{aligned}
[\![\Gamma \vdash (\lambda x.u)\ s : U]\!] &= \epsilon \circ \langle \widetilde{[\![\Gamma, x : S \vdash u : U]\!]}, [\![\Gamma \vdash s : S]\!] \rangle && \text{(definition of } [\![-]\!]) \\
&= \epsilon \circ (\widetilde{[\![\Gamma, x : S \vdash u : U]\!]} \times 1) \circ \langle 1_{[\![\Gamma]\!]}, [\![\Gamma \vdash s : S]\!] \rangle \\
& && \text{(expanding the definition of } - \times -) \\
&= [\![\Gamma, x : S \vdash u : U]\!] \circ \langle 1_{[\![\Gamma]\!]}, [\![\Gamma \vdash s : S]\!] \rangle && \text{(commutativity of exponentials)} \\
&= [\![\Gamma \vdash u[s/x] : U]\!]
\end{aligned}$$

Verification for other equivalence rules follows immediately from the axioms of the corresponding structures:

(1) $*$-η corresponds to the uniqueness of the terminal object;
(2) $\times$-β_1 and $\times$-β_2 correspond to the communativity of products discussed in Definition 2.4;
(3) $\times$-η corresponds to the uniqueness of products;
(4) $\rightarrow$-η corresponds to the uniqueness of exponentials.

This concludes that STLC can be interpreted in any cartesian closed category. If the given $\mathcal{C}$ is a commonly believed sound category, then we obtain the semantic soundness of STLC. For example, we can apply Theorem 4.3 to $\mathcal{S}et$ to consider STLC operating in set theory.

4.4 Completeness of Cartesian Closed Categories

Theorem 4.3 draws the conclusion that cartesian closed categories are sound models of STLC. Soundness ensures an embedding from STLC into any CCC and that equations provable in STLC must also hold in that CCC. However, the converse is not automatic: some CCCs might have additional structures, so that there could be provable equations in the model that are not provable in STLC. One simple counterexample for this is the category 1. Since this category has only one object, we can let the product and the exponential of itself be itself. The axioms are automatically true since there is only one morphism which must be the identity morphism. Finally, we obtain some provable equations which might look somewhat arbitrary, e.g.

$$\langle 1, 1 \rangle = \epsilon = 1$$

This equation clearly does not have any correspondence in STLC. Thus, besides soundness, we would also be interested in showing that equations in the domain of the interpretation go back to STLC. This property is called *completeness*. More formally,

Theorem 4.4. *There exists a cartesian closed category, so that if* $[\![\Gamma \vdash s : T]\!] = [\![\Gamma \vdash u : T]\!]$, *then* $\Gamma \vdash s = u : T$.

With completeness, CCC and STLC are really the same theory in two superficial disguises. To show completeness, the goal is a term model, which is a cartesian closed category constructed from STLC. The construction is as follows and the constructed category is called the *classifying category* of STLC.

Definition 4.5. The classifying category of STLC $\mathcal{C}l$(STLC) has the following data:

(1) the objects are contexts over α equivalence,
(2) the morphisms between two contexts Γ and Δ are the equivalence class of simultaneous substitutions over equivalence judgments in Section 4.1 between them,
(3) the morphism composition is the substitution composition,

(4) the identity morphisms are identity substitution, and
(5) the terminal object is the empty context $\cdot$.

To show $\mathcal{C}\ell(\text{STLC})$ is a category, one must show the categorical axioms. Simultaneous substitutions in STLC are lists of terms:

$$\sigma \equiv (t_1, \cdots, t_k) : \Gamma_1 \to \Gamma_2 \qquad \text{where } \Gamma_1 \vdash t_i : T_i \text{ and } \Gamma_2 = x_1 : T_1, ..., x_k : T_k$$

When the substitution applies to a term under context Γ_2, it replaces corresponding x_i with t_i. The identity substitution thus takes t_i to be x_i. For another substitution $\delta : \Gamma_0 \to \Gamma_1$, the composition is defined by

$$\sigma \circ \delta \equiv (t_1[\delta], \cdots, t_k[\delta]) : \Gamma_0 \to \Gamma_2$$

We can prove the axioms of a category by expanding the definition.

Then we should show $\mathcal{C}\ell(\text{STLC})$ is a CCC. It is easy to show $\mathcal{C}\ell(\text{STLC})$ has finite products. Since the terminal object is the empty context, any substitution to it contains no term and thus is necessarily unique. Given a context $\Gamma_1 = (x_1 : T_1, \cdots, x_k : T_k)$ and $\Gamma_2 = (y_1 : U_1, \cdots, y_l : U_l)$, their product is their concatenation $\Gamma_1, \Gamma_2 = (x_1 : T_1, \cdots, x_k : T_k, y_1 : U_1, \cdots, y_l : U_l)$. The projection morphisms project the corresponding part out:

$$\pi_1(\Gamma_1, \Gamma_2) = (x_1, \cdots, x_k)$$
$$\pi_2(\Gamma_1, \Gamma_2) = (y_1, \cdots, y_l)$$

For two substitutions $\sigma_1 = (t_1, \cdots, t_k) : \Delta \to \Gamma_1$ and $\sigma_2 = (u_1, \cdots, u_l) : \Delta \to \Gamma_2$, $\langle \sigma_1, \sigma_2 \rangle = (t_1, \cdots, t_k, u_1, \cdots, u_l)$ is their juxtaposition. Uniqueness follows naturally.

It remains to show that $\mathcal{C}\ell(\text{STLC})$ has exponentials. For two contexts $\Gamma_1 = (x_1 : T_1, \cdots, x_k : T_k)$ and $\Gamma_2 = (y_1 : U_1, \cdots, y_l : U_l)$, their exponential $\Gamma_1^{\Gamma_2}$ is exponentiated "point-wise": $(f_1 : U_1 \times \cdots \times U_l \to T_1, \cdots, f_k : U_1 \times \cdots \times U_l \to T_k)$. That is, $\Gamma_1^{\Gamma_2}$ has the same length as Γ_1 and for each binding, the type becomes a function taking products of types in Γ_2 as an input and returning the originally bound type. One can think of exponential of contexts as a function taking products and returning products. Then it is not hard to see exponential axioms are satisfied. The evaluation morphism $\epsilon : \Gamma_1^{\Gamma_2}, \Gamma_2 \to \Gamma_1$ is defined to be $(f_1(y_1, \cdots, y_l), \cdots, f_k(y_1, \cdots, y_l))$. Given a substitution $\sigma = (t_1, \cdots, t_k) : \Delta, \Gamma_2 \to \Gamma_1$, its transpose $\widetilde{\sigma}$ is $(\lambda y_1 \cdots y_l.t_1, \cdots, \lambda y_1 \cdots y_l.t_k)$. Both commutativity and uniqueness follow by expanding the definition.

This shows that $\mathcal{C}\ell(\text{STLC})$ is indeed a CCC. Its completeness is a consequence of the fact that all morphisms are composed of STLC terms, and thus admit no more equations than STLC does. This is seen by plugging $\mathcal{C}\ell(\text{STLC})$ into the interpretation in the previous section. Given a well-formed term $\Gamma \vdash s : T$, the interpretation $[\![\Gamma \vdash s : T]\!]$ is a singleton substitution from Γ to $x : T$. Thus $[\![\Gamma \vdash s : T]\!] = [\![\Gamma \vdash u : T]\!]$ becomes $(s) = (u)$ as a substitution, and therefore $\Gamma \vdash s = u : T$.

5 CATEGORICAL SEMANTICS FOR DEPENDENT TYPES

5.1 Dependent Types

In the previous section, we consider a language with simple types. It is simply typed, in particular because it has only simple functions, which take terms and return terms. In more sophisticated languages like Java, ML, and Haskell, there are generics or parametric polymorphism, which takes types and returns terms. Moreover, there is another dimension, which takes terms and returns types, namely *dependent types*.

A typical example for dependent types is vectors. A vector is just like a list, but it has its length in its type. In Agda [Agda Team 2019], a proof assistant based on dependent type theory, lists and vectors are defined as follows:

```
data List (A : Set) : Set a where
  []  : List A
  _::_ : (x : A) (xs : List A) → List A

data Vec (A : Set) : ℕ → Set a where
  []  : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)
```

where $\mathbb{N}$ represents natural numbers, and `zero` and `suc` are the constructors for 0 and successors, respectively. **`Set`** is the universe of types in Agda, which can be ignored in this example. Both `List` and `Vec` have overloaded constructors `[]` and `_::_`. In the definition of `Vec`, constructor `[]` constructs an empty list, which has length 0, while constructor `_::_` prepends a vector of length `n` with a term of type `A`, resulting a vector of length `suc n`. The lengths of vectors are maintained across all operations, while lists have no such information. Consider the following concatenation function which joins two vectors into one:

```
_++_ : Vec A m → Vec A n → Vec A (m + n)
[]       ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

This function is defined recursively and by pattern matching on the first vector. It takes a vector of length `m` and another of length `n` and returns a vector of length `m + n`. The length information stored in type is implicitly maintained during the recursion.

Another benefit of this length-indexed vector definition is a safe `head` operation, which is impossible for lists:

```
head : List → A
head []       = ? -- what can we fill in here???
head (x :: xs) = x
```

We fail to define a `head` function for lists, because we have to handle the case for `[]`, in which we cannot always provide an term of an arbitrary type `A`.

Contrarily, since vectors are length-indexed, as long as we know it has at least one element, we know we must be able to extract one element out of it:

```
head : Vec A (1 + n) → A
head (x :: xs) = x
```

In particular, the case for `[]` is omitted: since it is impossible for a vector of length 0 to have at least one element, Agda automatically knows that this case is absurd and does not even require this case to be written down.

In this vector example, we glanced a little at the use of dependent types. In general, dependent types can be used to encode software specifications and even serve as a mathematical foundation. Its interactions with many other creative type formers (e.g. modalities) remain to be explored. Thus, it is essential to have a common platform which help find the connections among different explorations. In this section, we consider a very typical categorical formulation of dependent types, *categories with families (CwF)* [Cartmell 1986; Dybjer 1995; Hofmann 1997]. There are other related formulations for dependent types, e.g. Seely [1984]'s model in locally cartesian closed categories and Jacobs [1993]'s comprehension categories.

5.2 Formal Definition of Dependent Types

Let us first define the syntax of our small example dependent type system:

$$S, T, U := * \mid \Pi x : S.U \qquad \text{types}$$

where Π types generalize function types. We omit Σ types, the generalization of product types, in favour of conciseness. Π types are dependent types since U can refer to x and other variables in the context. Since types can refer to variables, unlike the simply typed case, types are not always well-formed. Thus we have separate formation judgments for types and contexts:

$$\frac{\vdash \Gamma}{\Gamma \vdash * \ \texttt{type}} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash S \ \texttt{type} \qquad \Gamma, x : S \vdash U \ \texttt{type}}{\Gamma \vdash \Pi x : S.U \ \texttt{type}} \qquad \frac{}{\vdash \cdot} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash T \ \texttt{type}}{\vdash \Gamma, x : T}$$

where $\Gamma \vdash T \ \texttt{type}$ denotes that T is a well-formed type under context Γ and $\vdash \Gamma$ denotes that the context Γ is well-formed. In addition, we assume all occurrences of variables are fresh. All contexts from this point on are assumed to be well-formed according to the $\vdash \Gamma$ judgments, so every judgment related to Γ has an implicit condition $\vdash \Gamma$. For Π types, U is well-formed in an extended context, allowing U to refer to x.

Once formation rules are settled, we define the typing rules for terms:

$$\frac{}{\Gamma \vdash () : *} \qquad \frac{\Gamma, x : S \vdash t : U}{\Gamma \vdash \lambda x.t : \Pi x : S.U} \qquad \frac{\Gamma \vdash t : \Pi x : S.U \qquad \Gamma \vdash s : S}{\Gamma \vdash t \ s : U[s/x]} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

When introducing a Π type, U can refer to the extended binding $x : S$ so the application rule must substitute s for x in U.

The last step is the equivalence rules. In dependently typed settings, computation of terms can occur on the type level, which necessarily induces a notion of equivalence between types.

$$\frac{\Gamma \vdash t : *}{\Gamma \vdash t = () : *} \qquad \frac{\Gamma, x : S \vdash u : U \qquad \Gamma \vdash s : S}{\Gamma \vdash (\lambda x.u) \ s = u[s/x] : U[s/x]} \qquad \frac{\Gamma \vdash t : \Pi x : S.U}{\Gamma \vdash t = \lambda x.t \ x : \Pi x : S.U}$$

$$\frac{\Gamma \vdash S_1 = S_2 \ \texttt{type} \qquad \Gamma, x : S_1 \vdash U_1 = U_2 \ \texttt{type}}{\Gamma \vdash \Pi x : S_1.U_1 = \Pi x : S_2.U_2 \ \texttt{type}} \qquad \frac{\Gamma \vdash T_1 = T_2 \ \texttt{type} \qquad \Gamma \vdash t : T_1}{\Gamma \vdash t : T_2}$$

where $\Gamma \vdash S = U \ \texttt{type}$ denotes the equivalence relation between types and $\vdash \Gamma_1 = \Gamma_2$ denotes the equivalence relation between contexts, induced by the former. We present the congruence rules for Π types while omit those for terms and axioms for equivalences. The equivalence rules of terms characterizes β and η rules of all type formers. The last rule is the conversion rule, which permits a term to have another type as long as that type is equivalent.

Now we have obtained a dependent type theory. This system, however, is weaker than most dependent type theories which one might usually encounter, e.g. Martin-öf type theory(MLTT) [Martin-Löf 1984], Calculus of Constructions(CoC) [Coquand and Huet 1988]. One important missing feature is *universes* [Palmgren 1998]. Universes in dependent type theories are a hierarchy of types and introduce modularity to the theories. As useful as this feature is, universes are a complex problem and orthogonal to dependent types, so we use this "one-universe" type theory here instead. That said, this system is already more complex than CCCs can model, so we must consider another categorical model, categories with families(CwF), in the next section.

5.3 Categories with Families

Unlike the case in Section 4, where the interpretation of types is automatically stable, the one of dependent types is no longer the case, because types now contain any computable terms and thus are no longer identified syntactically. That is, the interpretation itself must respect all equivalence relations defined above. Substitutions in types particularly draw the difference. Consider the

following application rules for functions in the simply typed and the dependently typed cases:

$$\frac{\Gamma \vdash t : S \to U \qquad \Gamma \vdash s : S}{\Gamma \vdash t\ s : U} \qquad\qquad \frac{\Gamma \vdash t : \Pi x : S.U \qquad \Gamma \vdash s : S}{\Gamma \vdash t\ s : U[s/x]}$$

In the simply typed case on the left, the result type and the return type of t are both U. Since U is used in both places, in the interpretation to CCC, we can canonically refer to $[\![U]\!]$. However, in the dependently typed case on the right, the result type and the return type of t are not generally syntactically the same. Moreover, U is defined in the context $\Gamma, x : S$ while $U[s/x]$ is defined in Γ. If we naively characterize substitutions as compositions of some morphisms as in CCC, the model would be unsound. This implies that substitutions must be an invariant of the intended categorical model and requires special attention. To address this issue, Cartmell [1986] introduces categories with attributes(CwAs) and Dybjer [1995] introduces categories with families(CwFs), an equivalent formulation to CwAs. As opposed to the simply typed case, where substitutions are defined afterwards as products of terms, one big advantage of CwAs and CwFs is that they have substitutions as part of the structure, which makes them suitable for modelling dependently typed theories. Since they are equivalence concepts, in this report, we only introduce CwFs for conciseness.

CwFs are defined via a special category of families of sets, $\mathcal{F}am$:

Definition 5.1. The category of families of sets, $\mathcal{F}am$, consists of the following data:

(1) As objects, it has products (T, t), where T is a set and t is a family of sets indexed by T. That is, for each $\tau \in T$, t_τ is a set.
(2) As morphisms between (S, s) and (U, u), it has products (f, f'), where $f : S \Rightarrow U$ and f' is a family of functions indexed by f. That is, for each $\tau \in S$, $f' : s_\tau \Rightarrow u_{f(\tau)}$.

Identity morphisms are pairs of identity functions. Composition of morphisms are pointwise composition of functions. The definition is well-formed and the axioms can be verified.

Notice that a functor to $\mathcal{F}am$ can be decomposed into two parts: the first part mapping to the index set and the second part mapping to the indexed family of sets. A CwF is defined via one such functor to $\mathcal{F}am$:

Definition 5.2. A category with families is a category $\mathcal{C}$ and a functor $F = (Ty, Tm) : \mathcal{C}^{op} \Rightarrow \mathcal{F}am$ where Ty maps to the set of *semantic types* and Tm maps to the set of *semantic terms*.

The semantic substitution operation for types $-\{\sigma\} : Ty(\Delta) \Rightarrow Ty(\Gamma)$ is defined by the morphism part of Ty and the semantic substitution operation for terms $-\{\sigma\} : Tm(\Delta, T) \Rightarrow Tm(\Gamma, T\{\sigma\})$ is defined by the morphism part of Tm, for $\sigma : \Gamma \Rightarrow \Delta$ and $T \in Ty(\Delta)$.

A CwF in addition is equipped with the following data:

(1) a *terminal object* $\top$ in $\mathcal{C}$,
(2) for an object $\Gamma \in \mathcal{C}$ and $T \in Ty(\Gamma)$, a *comprehension* of T, $\Gamma.T \in \mathcal{C}$,
(3) for an object $\Gamma \in \mathcal{C}$ and $T \in Ty(\Gamma)$, a first projection $p(T) : \Gamma.T \Rightarrow \Gamma$,
(4) for an object $\Gamma \in \mathcal{C}$ and $T \in Ty(\Gamma)$, a second projection $v_T \in Tm(\Gamma.T, T\{p(T)\})$, and
(5) for a morphism $\sigma : \Gamma \Rightarrow \Delta$, a type $T \in Ty(\Delta)$, and a term $t \in Tm(\Gamma, T\{\sigma\})$, there is a unique extension morphism $\langle \sigma, t \rangle : \Gamma \Rightarrow \Delta.T$.

The following equations must hold for extension morphism:

$$\begin{aligned} p(T) \circ \langle \sigma, t \rangle &= \sigma \\ v_T\{\langle \sigma, t \rangle\} &= t \\ \delta &= \langle p(T) \circ \delta, v_T\{\delta\} \rangle \qquad \text{where } \delta : \Gamma \Rightarrow \Delta.T \end{aligned}$$

From the functoriality of (Ty, Tm), we can proof the following equations:

$$T\{1_\Gamma\} = T \qquad \text{where } T \in Ty(\Gamma)$$
$$T\{\sigma\}\{\delta\} = T\{\sigma \circ \delta\}$$
$$t\{1_\Gamma\} = t \qquad \text{where } T \in Ty(\Gamma) \text{ and } t \in Tm(\Gamma, T)$$
$$t\{\sigma\}\{\delta\} = t\{\sigma \circ \delta\}$$

Compared to Section 4.4, a CwF generalizes CCC with dependent types. As suggesed in the definition, for a CwF $\mathcal{C}$, its objects are semantic contexts, Ty maps a context to a set of semantic types, and Tm is a set of semantic terms indexed by a semantic type in a context. As a result, a morphism in $\mathcal{C}$ represents a substitution morphism. The semantic substitution operations $-\{-\}$, thus, apply a substitution morphism to types and terms respectively. A substitution morphism can be extended by a term via a universal mapping $\langle -, - \rangle$, which possesses two projections, similar to products in CCC. It is worth mentioning that this "raw" definition of CwFs does not model any other type formers like Π types or Σ types. Nonetheless, we are already able to prove some useful properties.

Examples. Sometimes, working on a CwF itself feels like working in a dependent type theory, where high level equalities are trivialized. Consider one equality above:

$$t\{\sigma\}\{\delta\} = t\{\sigma \circ \delta\}$$

for $t \in Tm(\Gamma, T)$, $\sigma : \Gamma' \Rightarrow \Gamma$, and $\delta : \Gamma'' \Rightarrow \Gamma'$. The well-formedness of this equation is not immediate. If we inspect the sets which both terms reside in, we have the following:

$$t\{\sigma\}\{\delta\} \in Tm(\Gamma'', T\{\sigma\}\{\delta\})$$
$$t\{\sigma \circ \delta\} \in Tm(\Gamma'', T\{\sigma \circ \delta\})$$

We know these two sets are the same, due to another equality $T\{\sigma\}\{\delta\} = T\{\sigma \circ \delta\}$ which allows us to identify both sets. Thus, well-formedness of equations in CwFs often relies on equational reasoning on "higher dimensions", which themselves might not be immediate.

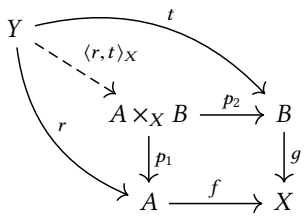
Next we want to characterize the fact that substitution is stable under weakening. That is, if we weaken a substitution morphism, then the weakened morphism should not corrupt the part governed by the original substitution morphism. This is characterized by some pullback property. First, let us give the definition of pullbacks:

Definition 5.3. A pullback of two morphisms $f : X \Rightarrow A$ and $g : Y \Rightarrow A$ has the following data:

(1) an object $A \times_X B$,
(2) two morphisms $p_1 : A \times_X B \Rightarrow A$ and $p_2 : A \times_X B \Rightarrow B$, and
(3) for two morphisms $r : Y \Rightarrow A$ and $t : Y \Rightarrow B$ such that $f \circ r = g \circ t$, a unique morphism $\langle r, t \rangle_X : Y \Rightarrow A \times_X B$.

The following axioms are satisfied:

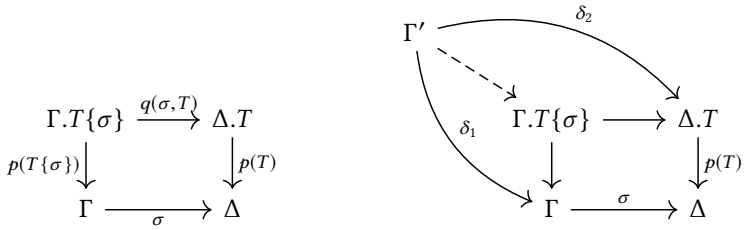
(1) commutativity: $f \circ p_1 = g \circ p_2$, $p_1 \circ \langle r, t \rangle_X = r$, and $p_2 \circ \langle r, t \rangle_X = t$, and
(2) uniqueness: for $h : Y \Rightarrow A \times_X B$, $\langle p_1 \circ h, p_2 \circ h \rangle_X = h$.



Informally, pullbacks are constrained products, hence the notation. If we think in set theory for a moment, then $A \times_X B$ is just a subset of $A \times B$, such that f applied to the first projection is equal to g applied to the second projection. More compactly, we have

$$A \times_X B = \{(a, b) \in A \times B \mid f(a) = g(b)\}$$

With this relation in mind, we can interpret the property above in the following diagram on the left, where $\sigma : \Gamma \Rightarrow \Delta$, $T \in Ty(\Delta)$, and a weakening morphism $q(\sigma, T) = \langle \sigma \circ p(T\{\sigma\}), v_{T\{\sigma\}} \rangle$.



We claim that this is a pullback square. It is easy to see that the square itself commutes, because $p(T)$ just projects the first component of $q(\sigma, T)$ out. To show that it is a pullback, let us assume any Γ', and two substitutions $\delta_1 : \Gamma' \Rightarrow \Gamma$ and $\delta_2 : \Gamma' \Rightarrow \Delta.T$, such that $\sigma \circ \delta_1 = p(T) \circ \delta_2$ as in the diagram on the right. We have $\langle \delta_1, \delta_2 \rangle_\Delta = \langle \delta_1, v_T\{\delta_2\} \rangle$. For commutativity:

$$p(T\{\sigma\}) \circ \langle \delta_1, v_T\{\delta_2\} \rangle = \delta_1$$

$$\begin{aligned}
\langle \sigma \circ p(T), v_{T\{\sigma\}} \rangle \circ \langle \delta_1, v_T\{\delta_2\} \rangle &= \langle \sigma \circ p(T) \circ \langle \delta_1, v_T\{\delta_2\} \rangle, v_{T\{\sigma\}}\{\langle \delta_1, v_T\{\delta_2\} \rangle\} \rangle \\
&= \langle \sigma \circ \delta_1, v_T\{\delta_2\} \rangle \\
&= \langle p(T) \circ \delta_2, v_T\{\delta_2\} \rangle \\
&= \delta_2
\end{aligned}$$

The equations hold due to congruence and projections of the extension morphism.

Last we shall prove uniqueness. By definition, we need to prove the following equation given a morphism $\delta : \Gamma' \Rightarrow \Gamma.T\{\sigma\}$:

$$\langle p(T\{\sigma\}) \circ \delta, q(\sigma, T) \circ \delta \rangle_\Delta = \delta$$

Expanding the definitions, we have

$$\begin{aligned}
\langle p(T\{\sigma\}) \circ \delta, q(\sigma, T) \circ \delta \rangle_\Delta &= \langle p(T\{\sigma\}) \circ \delta, v_T\{\langle \sigma \circ p(T), v_{T\{\sigma\}} \rangle \circ \delta\} \rangle \\
&= \langle p(T\{\sigma\}) \circ \delta, v_T\{\langle \sigma \circ p(T) \circ \delta, v_{T\{\sigma\}}\{\delta\} \rangle\} \rangle \\
&= \langle p(T\{\sigma\}) \circ \delta, v_{T\{\sigma\}}\{\delta\} \rangle = \delta
\end{aligned}$$

Definitions up to this point only enable basic modelling of dependent types themselves. In the calculus presented in Section 5.2, we have other type formers including Σ types and Π types. A "raw" CwF does not have sufficient structures to represent either type former. in the next few sections, we enrich a CwF with sufficient structures such that the dependently typed calculus defined in Section 5.2 can be modelled.

5.4 Semantics for Π Types

Just like cartesian categories require an extra closed structure represented in Section 4.2 in order to model STLC. For a CwF to model a dependent type theory with Π types, we also need the following additional structure.

Definition 5.4. Semantic Π types in a CwF C have the following data:

(1) A semantic type $\Pi(S,U) \in Ty(\Gamma)$ for each $S \in Ty(\Gamma)$ and $U \in Ty(\Gamma.S)$,
(2) a semantic term $\Lambda_{S,U}(t) \in Tm(\Gamma, \Pi(S,U))$ for each $t \in Tm(\Gamma.S, U)$, and
(3) a semantic term $App_{S,U}(t,t') \in Tm(\Gamma, U\{\langle 1_\Gamma, t'\rangle\})$ for each $t \in Tm(\Gamma, \Pi(S,U))$ and $t' \in Tm(\Gamma, S)$.

So the following axioms are satisfied:

(1) $\Pi(S,U)\{\sigma\} = \Pi(S\{\sigma\}, U\{q(\sigma,S)\}) \in Ty(\Delta)$ for $\sigma : \Delta \Rightarrow \Sigma$,
(2) $\Lambda_{S,U}(t)\{\sigma\} = \Lambda_{S\{\sigma\},U\{q(\sigma,S)\}}(t\{q(\sigma,S)\}) \in Tm(\Delta, \Pi(S,U)\{\sigma\})$ for $t \in Tm(\Gamma.S,U)$ and $\sigma : \Delta \Rightarrow \Sigma$,
(3) $App_{S,U}(t,t')\{\sigma\} = App_{S\{\sigma\},U\{q(\sigma,S)\}}(t\{\sigma\}, t'\{\sigma\}) \in Tm(\Delta, U\{\langle 1_\Delta, t'\rangle\}\{\sigma\}) = Tm(\Delta, U\{\langle \sigma, t'\{\sigma\}\rangle\})$ for $t \in Tm(\Gamma, \Pi(S,U))$, $t' \in Tm(\Gamma, S)$ and $\sigma : \Delta \Rightarrow \Sigma$,
(4) $App_{S,U}(\Lambda_{S,U}(t), t') = t\{\langle 1_\Gamma, t'\rangle\} \in Tm(\Gamma, U\{\langle 1_\Gamma, t'\rangle\})$ for each $t \in Tm(\Gamma.S, U)$ and $t' \in Tm(\Gamma, S)$, and
(5) $\Lambda_{S,U}(App_{S\{p(S)\},U\{q(p(S),S)\}}(t\{p(S)\}, v_S)) = t \in Tm(\Gamma, \Pi(S,U))$.

Recall the weakening morphism $q(\sigma, S) = \langle \sigma \circ p(S\{\sigma\}), v_{S\{\sigma\}}\rangle$. The three pieces of data correspond to the Π type former, the λ constructor and function applications, respectively. The first three axioms define how type and term substitution operations should interact with semantic Π types and their semantic terms. We need $q(\sigma, S)$ in these equations because we only want a substitution of the prefix from $\Delta.S\{\sigma\}$ to $\Gamma.S$, but not the term of type S in the second projection. The last two axioms model β and η equivalence of Π types.

Similar to an example in Section 5.3, not all axioms presented above are immediately well-defined. This is because Π types involve somewhat complex substitutions in its type and term formers. For example, the last equation is well-defined, due to the following membership relations:

$$\begin{aligned} t\{p(S)\} &\in Tm(\Gamma.S, \Pi(S,U)\{p(S)\}) = Tm(\Gamma.S, \Pi(S\{p(S)\}, U\{q(p(S),S)\})) \\ v_S &\in Tm(\Gamma.S, S\{p(S)\}) \end{aligned}$$

Due to the equality between two semantic type expressions, we know that the App expression is well-defined:

$$\begin{aligned} App_{S\{p(S)\},U\{q(p(S),S)\}}(t\{p(S)\}, v_S) &\in Tm(\Gamma.S, U\{q(p(S),S)\}\{\langle 1_{\Gamma.S}, v_S\rangle\}) \\ &= Tm(\Gamma.S, U\{q(p(S),S) \circ \langle 1_{\Gamma.S}, v_S\rangle\}) \\ \Lambda_{S,U}(App_{S\{p(S)\},U\{q(p(S),S)\}}(t\{p(S)\}, v_S)) &\in Tm(\Gamma, \Pi(S, U\{q(p(S),S) \circ \langle 1_{\Gamma.S}, v_S\rangle\})) \end{aligned}$$

To finally conclude the well-definedness of the equation, we want to show that

$$q(p(S),S) \circ \langle 1_{\Gamma.S}, v_S\rangle = 1_{\Gamma.S}$$

This can be proved by expanding the definition:

$$\begin{aligned} q(p(S),S) \circ \langle 1_{\Gamma.S}, v_S\rangle &= \langle p(S) \circ p(S\{p(S)\}) \circ \langle 1_{\Gamma.S}, v_S\rangle, v_{S\{p(S)\}\{\langle 1_{\Gamma.S}, v_S\rangle\}}\rangle \\ &= \langle p(S), v_{S\{p(S)\circ\langle 1_{\Gamma.S}, v_S\rangle\}}\rangle \\ &= \langle p(S), v_S\rangle \\ &= 1_{\Gamma.S} \end{aligned}$$

This equation allows us to conclude the following relation:

$$\begin{aligned} \Lambda_{S,U}(App_{S\{p(S)\},U\{q(p(S),S)\}}(t\{p(S)\}, v_S)) &\in Tm(\Gamma, \Pi(S, U\{q(p(S),S) \circ \langle 1_{\Gamma.S}, v_S\rangle\})) \\ &= Tm(\Gamma, \Pi(S, U\{1_{\Gamma.S}\})) \\ &= Tm(\Gamma, \Pi(S,U)) \end{aligned}$$

Thus the well-definedness of the original axioms are concluded.

Reasoning about Π types (or in general, all dependent types) often involve this kind of "higher" or set level equational reasoning. Nevertheless, if we follow the data definitions and axioms tightly, the resulting equations must be well-defined. Therefore, we usually omit equations arguing well-definedness like the one above. Furthermore, for conciseness, we often omit the subscripts of Λ and *App* when they can be inferred. For example, instead of writing

$$\Lambda_{S,U}(App_{S\{p(S)\},U\{q(p(S),S)\}}(t\{p(S)\}, v_S)) = t$$

we write

$$\Lambda(App(t\{p(S)\}, v_S)) = t$$

because the subscripts can be safely inferred from the set which t resides in.

5.5 Semantics for $*$

For rigorousness, we have to formulate the semantics for $*$ types in order to handle all structures in the dependent type theory.

Definition 5.5. Semantic $*$ types in a CwF C has the following data:

(1) a semantic type $* \in Ty(\Gamma)$, and
(2) a semantic term $() \in Tm(\Gamma, *)$ for all Γ.

So the following axioms hold:

(1) $*\{\sigma\} = *$ for $\sigma : \Delta \Rightarrow \Gamma$. Here the first $* \in Ty(\Gamma)$ and the second $* \in Ty(\Delta)$.
(2) $()\{\sigma\} = ()$ for $\sigma : \Delta \Rightarrow \Gamma$. Here the first $() \in Tm(\Gamma, *)$ and the second $() \in Ty(\Delta, *)$.
(3) Finally we have $() = t$ for $t \in Tm(\Gamma, *)$.

5.6 Interpreting into CwFs

Following Section 4.3, a categorical semantics for dependent type theory requires a sound interpretation in CwFs with Π and $*$ types. In Section 4.3, since types are by default in STLC, we can interpret types and contexts just by looking into their structures, while here, we have to worry about when a type or context is well-formed. We thus adjust our interpretation function to take well-formedness judgments as well.

$$\begin{aligned}
[\![\Gamma \vdash * \ \ \texttt{type}]\!] &= * \\
[\![\Gamma \vdash \Pi x : S.U \ \ \texttt{type}]\!] &= \Pi([\![\Gamma \vdash S \ \ \texttt{type}]\!], [\![\Gamma, x : S \vdash U \ \ \texttt{type}]\!]) \\
[\![\Gamma_2 \vdash T \ \ \texttt{type}]\!] &= [\![\Gamma_1 \vdash T \ \ \texttt{type}]\!] \\
[\![\vdash \cdot]\!] &= \bot \\
[\![\vdash \Gamma, x : T]\!] &= [\![\vdash \Gamma]\!].[\![\Gamma \vdash T \ \ \texttt{type}]\!]
\end{aligned}$$

Before move on to interpreting other judgments, we should determine how substitutions are interpreted. In the semantics, substitutions are context morphisms, so we should first define simultaneous substitutions:

$$\frac{}{\Gamma \vdash () \Rightarrow \cdot} \qquad \frac{\Gamma \vdash \sigma \Rightarrow \Delta \qquad \Gamma \vdash t : T[\sigma]}{\Gamma \vdash (\sigma, t) \Rightarrow \Delta, x : T}$$

Like in STLC, simultaneous substitutions are tuples of terms; unlike in STLC, the typing judgments of later terms depend on previous substitutions. We then interpret simultaneous substitutions:

$$\begin{aligned}
[\![\Gamma \vdash () \Rightarrow \cdot]\!] &= \top \\
[\![\Gamma \vdash (\sigma, t) \Rightarrow \Delta, x : T]\!] &= \langle [\![\Gamma \vdash \sigma \Rightarrow \Delta]\!], [\![\Gamma \vdash t : T[\sigma]]\!] \rangle
\end{aligned}$$

During the interpretation of substitutions, we need the interpretation of terms. Let us next determine typing judgments of terms, including the conversion rule:

$$\begin{aligned}
[\![\Gamma \vdash () : *]\!] &= () \\
[\![\Gamma \vdash \lambda x.t : \Pi x : S.U]\!] &= \Lambda([\![\Gamma, x : S \vdash t : U]\!]) \\
[\![\Gamma \vdash t\ s : U[s/x]]\!] &= App([\![\Gamma \vdash t : \Pi x : S.U]\!], [\![\Gamma \vdash s : S]\!]) \\
[\![\Gamma \vdash x : T]\!] &= v_T\{p^k\} \qquad \text{where } \Gamma = \Gamma_1, x : T, \Gamma_2 \text{ and } |\Gamma_2| = k \\
[\![\Gamma \vdash t : T_2]\!] &= [\![\Gamma \vdash t : T_1]\!]
\end{aligned}$$

The last equation interprets the conversion rule. The second last equation interprets the variable rule. v_T only projects the very last binder out. The substitution $-\{p^k\}$ denotes k first projections which projects away Γ_2, so v_T is weakened to live in $[\![\vdash \Gamma]\!]$.

Finally, we interprets the equivalence judgments. The congruence rules which we omitted hold automatically due to the nature of equational theory. This turns equivalences between types and contexts trivial. let us justify the nontrivial equivalences between terms:

(1) $[\![\Gamma \vdash t = () : *]\!]$ requires to justify $[\![\Gamma \vdash t : *]\!] = ()$ which is an axiom of semantic $*$.
(2) $[\![\Gamma \vdash (\lambda x.u)\ s = u[s/x] : U[s/x]]\!]$ becomes

$$App(\Lambda([\![\Gamma, x : S \vdash u : U]\!]), [\![\Gamma \vdash s : S]\!]) = [\![\Gamma, x : S \vdash u : U]\!]\{\langle 1, [\![\Gamma \vdash s : S]\!]\rangle\}$$

This equation holds due to an axiom of Π types.
(3) $[\![\Gamma \vdash t = \lambda x.t\ x : \Pi x : S.U]\!]$ becomes

$$[\![\Gamma \vdash t : \Pi x : S.U]\!] = \Lambda(App([\![\Gamma \vdash t : \Pi x : S.U]\!]\{p\}, v_S))$$

This equation is justified by an axiom of Π types.

We summarize the overall soundness theorem of all interpretation functions as follows:

Theorem 5.6. *(soundness)*

(1) $[\![\vdash \Gamma]\!] \in \mathcal{C}$.
(2) $[\![\Gamma \vdash T\ \ \mathtt{type}]\!] \in Ty([\![\vdash \Gamma]\!])$.
(3) $[\![\Gamma \vdash t : T]\!] \in Tm([\![\vdash \Gamma]\!], [\![\Gamma \vdash T\ \ \mathtt{type}]\!])$
(4) *For any* $\Gamma \vdash \sigma \Rightarrow \Delta$, $[\![\Gamma \vdash T[\sigma]\ \ \mathtt{type}]\!] = [\![\Delta \vdash T\ \ \mathtt{type}]\!]\langle [\![\Gamma \vdash \sigma \Rightarrow \Delta]\!]\rangle$.
(5) *For any* $\Gamma \vdash \sigma \Rightarrow \Delta$, $[\![\Gamma \vdash t[\sigma] : T[\sigma]]\!] = [\![\Delta \vdash t : T]\!]\langle [\![\Gamma \vdash \sigma \Rightarrow \Delta]\!]\rangle$.
(6) *If* $\vdash \Gamma = \Delta$, *then* $[\![\vdash \Gamma]\!] = [\![\vdash \Delta]\!]$.
(7) *If* $\Gamma \vdash S = U\ \ \mathtt{type}$, *then* $[\![\Gamma \vdash S\ \ \mathtt{type}]\!] = [\![\Gamma \vdash U\ \ \mathtt{type}]\!]$.
(8) *If* $\Gamma \vdash t_1 = t_2 : T$ *then* $[\![\Gamma \vdash t_1 : T]\!] = [\![\Gamma \vdash t_2 : T]\!]$.

Typically, the soundness theorem is proved by mutual induction on the judgments. The first three statements assert that contexts, types, and terms respect the semantic counterparts. The next two statements assert that substitutions are respected. Finally the last three state that equivalences are preserved as equality of the corresponding objects. Thus we can soundly interpret the dependent type theory in any CwF satisfying the structures.

5.7 A Term Model Construction

In Section 4.4, we presented a term model construction which shows that STLC is itself a CCC. Following the same line, we are also interested in a completeness result by constructing a term model of the dependent type theory defined in Section 5.2. In this section, our intention is to define a CwF with objects as (syntactical) contexts and morphisms as simultaneous substitutions between contexts, and then we show that this CwF has Π, and $*$ types.

The first step is to show that this category is indeed a category. We can prove the axioms by induction on the proper objects. Thus, we have obtained a syntactical category. Then we show that this category is a CwF. First, it is easy to see that the empty context $\cdot$ is a terminal object, because the definition of simultaneous substitutions shows that there is only one substitution morphism to it. $Ty(\Gamma)$ is the set of all legal types in context Γ and $Tm(\Gamma, T)$ where $T \in Ty(\Gamma)$ is the set of all legal terms of type T in context Γ. The functoriality of (Ty, Tm) has been verified during the process of proving the categorical axioms above. A context comprehension of T is the context extension with T. First projection morphism simply drops the last term in the originally identity substitution morphism. The second projection takes the last variable from a context with at least one binding, which corresponds to the following special case of the variable rule:

$$\frac{}{\Gamma, x : T \vdash x : T}$$

The definition requires the term to have type $T\{p(T)\}$ which is equal to T in the term model because we know that T only depends on the domain of Γ due to well-formedness judgments. The definition of simultaneous substitutions has given a definition of the extension morphisms. The equations of extension morphisms hold by definition. At this point, we have obtained a term model of CwFs.

Furthermore, we can show that the term model has more structures, as expected. In particular, it should have all discussed semantic types. Let us consider semantic Π types first. Thanks to the direct definition of semantic Π types in Section 5.4, we can immediately see that $\Pi(S, U)$ is $\Pi x : S.U$, $\Lambda(t)$ is $\lambda x.t$ and $App(t, t')$ is function application $t\ t'$. Equations of substitutions for type and term formers rely on $q(\sigma, S) = \langle \sigma \circ p(S\{\sigma\}), v_{S\{\sigma\}} \rangle$. This morphism is modelled by the following judgment:

$$\frac{\Gamma \vdash \sigma \Rightarrow \Delta \qquad \Gamma, x : S[\sigma] \vdash x : S[\sigma]}{\Gamma, x : S[\sigma] \vdash q(\sigma, S) \Rightarrow \Delta, y : S}$$

which substitutes every variable except x. The last two equations are proved by β and η equivalences of Π types.

The final part is $*$ types. We let the semantic $*$ types to be the syntactical $*$ types and the semantic () to be the syntactical (). All equations are trivial in this case.

At this point, we have shown that the dependent type theory discussed in this section is a CwF. Like in Section 4.4, we plug in the term model into the interpretation in Section 5.6 and obtain completeness theorem:

Theorem 5.7. *(completeness)*

(1) *If* $[\![\vdash \Gamma]\!] = [\![\vdash \Delta]\!]$, *then* $\vdash \Gamma = \Delta$.
(2) *If* $[\![\Gamma \vdash S\ \texttt{type}]\!] = [\![\Gamma \vdash U\ \texttt{type}]\!]$, *then* $\Gamma \vdash S = U\ \texttt{type}$.
(3) *If* $[\![\Gamma \vdash t_1 : T]\!] = [\![\Gamma \vdash t_2 : T]\!]$, *then* $\Gamma \vdash t_1 = t_2 : T$.

6 OTHER EXTENSIONS

In Section 5, we discussed a categorical model, CwFs, for dependent type theories. Nonetheless, the calculus presented in Section 5 is minimal: it only contains a set of basic type and term formers and a "real-world" type theory usually contain many other features. In this section, we discuss universes and modalities which are common extensions to a dependent type theory.

6.1 Universes

Modern dependently typed systems usually have an infinite hierarchy of universes [Coquand and Huet 1988; Martin-Löf 1984], which is not modelled by default in CwFs. Compared to the dependent type theory defined in Section 5.2, where we use two different groups of judgments to characterize

well-formedness and equations of types and terms, with universes, the distinctions between types and terms are vanished. As a result, dependent type theories equipped with universes has obtained more power in expressing polymorphism. For example, in Agda's syntax, we are able to write down the following predicate:

```
data All {A : Set} (P : A → Set) : List A → Set where
  []  : All P []
  _::_ : ∀ {x xs} (px : P x) (pxs : All P xs) → All P (x :: xs)
```

Here, `P` is a unary predicate, and `P x` for some `x : A` denotes the evidence that `x` satisfies certain property. Notice how `P` is typed. We rely on the universe **`Set`** to modularly express *any* predicate on `A`. This form of definitions is not possible in the dependent type theory in Section 5.2.

Back to the rules, there are typically two formulations of universes [Martin-Löf 1984; Palmgren 1998]: universes à la Russell and universes à la Tarski. Two kinds of universes are distinguished by how type constructors are interpreted in the universes. Between the two, universes à la Russell are more frequently seen and implemented in proof assistants:

$$\frac{}{\Gamma \vdash U_i \ \ \mathtt{type}} \qquad \frac{\Gamma \vdash T : U_i}{\Gamma \vdash T \ \ \mathtt{type}} \qquad \frac{}{\Gamma \vdash U_i : U_{i+1}}$$

That is, a universe is a type of types. There are usually an infinite number of universes, each indexed by a natural number as indicated by the subscripts, forming a hierarchy. The index of a universe is called a *level*. Note the rule $\Gamma \vdash U_i : U_{i+1}$, asserting universes are contained in the universe in the immediately next ones in the hierarchy. One might often encounter another rule:

$$\frac{\Gamma \vdash T : U_i}{\Gamma \vdash T : U_{i+1}}$$

This rule says that a type in a smaller universe also reside in bigger universes. Thus universes *accumulate*. This rule is optional. Without accumulativity, we can always lift the universe level of a type by requiring a special construct:

$$\frac{\Gamma \vdash T : U_i}{\Gamma \vdash L(T) : U_{i+1}}$$

Indeed, some proof assistants (e.g. Coq [The Coq Development Team 2019]) permits accumulativity while some proof assistants turns it off by default (e.g. Agda). Universes à la Russell are popular primarily because of its convenience in programming. They also fit very well with the usual interpretation of types-as-sets, where typing judgment between types are just inclusions.

Universes à la Tarski, on the other hand, do not consider types as members of universes; instead, universes only contain codes and one must rely on a decoding function to convert codes into the actual types:

$$\frac{}{\Gamma \vdash U_i \ \ \mathtt{type}} \qquad \frac{\Gamma \vdash c : U_i}{\Gamma \vdash El(c) \ \ \mathtt{type}} \qquad \frac{\Gamma \vdash c : U_i}{\Gamma \vdash l(c) : U_{i+1}} \qquad \frac{}{\Gamma \vdash u_i : U_{i+1}}$$

$$\frac{}{\Gamma \vdash El(u_i) = U_i \ \ \mathtt{type}} \qquad \frac{\Gamma \vdash c : U_i}{\Gamma \vdash El(l(c)) = El(c) \ \ \mathtt{type}}$$

With universes à la Tarski, we still have universes as type. However, a type is obtained by decoding a code c as an element of some universe with a decoder El. That is, El converts a code of type to a real type. A code c can be lifted to a higher universe by applying lift function l. Moreover, a higher universe contains a code for the lower universe, such that when decoded, one obtains the lower

universe. The last equation states that the lift function does not have any effect when decoded. A type former, for example, can be represented as follows:

$$\frac{\Gamma \vdash c : U_i \qquad \Gamma \vdash d : U_i}{\Gamma \vdash \pi(c,d) : U_i} \qquad\qquad \frac{\Gamma \vdash c : U_i \qquad \Gamma \vdash d : U_i}{\Gamma \vdash El(\pi(c,d)) = \Pi x : El(c).El(d)\ \ \mathtt{type}}$$

One might expect that the distinction between universes à la Russell and à la Tarski is superfluous but is actually quite fundamental [Luo 2012]. Next let us discuss a semantic model supporting universes à la Tarski (with some differences) which extends CwFs.

Categories with universes [Birkedal et al. 2020] are a model of dependent type theories with universes by extending CwFs. We begin with its definition.

Definition 6.1. A category with universes(CwU) is a category $\mathcal{C}$ with the following data:
A CwF in addition is equipped with the following data:

(1) a terminal object $\top$ in $\mathcal{C}$,
(2) for an object $\Gamma \in \mathcal{C}$, a mapping to set $Ty(\Gamma, n)$ denoting semantic types at level $n \in \mathbb{N}$ in context Γ,
(3) for an object $\Gamma \in \mathcal{C}$ and a semantic type $T \in Ty(\Gamma, n)$, an indexed set $Tm(\Gamma, T)$ denoting semantic terms of type T in context Γ,
(4) for a substitution morphism $\sigma : \Gamma \Rightarrow \Delta$, semantic substitutions $-\{\sigma\}$ for types and terms,
(5) for an object $\Gamma \in \mathcal{C}$ and $T : Ty(\Gamma, n)$, a comprehension of T is another object in $\mathcal{C}$, denoted by $\Gamma.T$,
(6) a first projection morphism in $\mathcal{C}$ between a comprehension and its prefix object $p(T) : \Gamma.T \Rightarrow \Gamma$,
(7) a second projection $v_T \in Tm(\Gamma.T, T\{p(T)\})$,
(8) for a morphism $\sigma : \Gamma \Rightarrow \Delta$, a type $T : Ty(\Delta)$, and a term $t : Tm(\Gamma, T\{\sigma\})$, there is a unique extension morphism $\langle \sigma, t \rangle : \Gamma \Rightarrow \Delta.T$.
(9) for each $n \in \mathbb{N}$, $U_n \in Ty(\top, n+1)$, denoting the universe at level n, and
(10) an isomorphism of the decoding function $El(c) \in Ty(\Gamma, n)$ where $c \in Tm(\Gamma, U_n\{!\})$ and $!$ is the unique morphism to the terminal object.

The following equations hold for extension morphism:

$$\begin{aligned} p(T) \circ \langle \sigma, t \rangle &= \sigma \\ v_T\{\langle \sigma, t \rangle\} &= t \\ \delta &= \langle p(T) \circ \delta, v_T\{\delta\} \rangle \qquad \text{where } \delta : \Gamma \Rightarrow \Delta.T \end{aligned}$$

Semantic substitutions are coherent:

$$\begin{aligned} T\{1_\Gamma\} &= T \\ T\{\sigma\}\{\delta\} &= T\{\sigma \circ \delta\} \\ t\{1_\Gamma\} &= t \\ t\{\sigma\}\{\delta\} &= t\{\sigma \circ \delta\} \end{aligned}$$

The inverse of the decoding function, the encoding function, is $\lceil T \rceil \in Tm(\Gamma, U_n\{!\})$ where $T \in Ty(\Gamma, n)$. Since El and $\lceil - \rceil$ form an isomorphism, we have

$$\begin{aligned} El(\lceil T \rceil) &= T \\ \lceil El(c) \rceil &= c \\ \lceil c \rceil\{\sigma\} &= \lceil c\{\sigma\} \rceil \end{aligned}$$

The first part of this definition extends the definition of CwFs naturally by requiring Ty takes an additional natural number as an argument, which serves as the universe level. The last two pieces of data model universes. U_n models a universe at level n. What is not quite the same as the formulation of universes à la Tarski presented in the previous section is the decoding function El. Here El is an *isomorphism*. This just means that in the model codes and types are isomorphic. This turns the model to support universes in an easier style than ones à la Tarski. Since we have the encoding function, we obtain the code for universes quite naturally via $u_n \equiv \lceil U_n \rceil \in Tm(\top, U_{n+1}\{!\}) = Tm(\top, U_{n+1})$.

What is especially nice about this semantic model is that many semantic types also naturally adapts to it, including Π and Σ types, except for some necessary adjustments. For example, the definition of Π types can almost stay untouched, as long as we made an adjustment to the type former $\Pi(S, U)$ by requiring S and U to be in $Ty(\Gamma, n)$ for some universe level n. Then $\Pi(S, U)$ itself is also at level n. The encoding function $\lceil - \rceil$ also simplifies the definition here. We can obtain the code of Π type be the unique function π, such that

$$El(\pi(c, d)) = \Pi(El(c), El(d))$$

for suitable codes c and d. We know the code function π must exist uniquely since the encoding and decoding functions form an isomorphism.

Having Π types in CwUs produces visibly stronger expressive power than CwFs with Π types. For example, the semantic model now is capable of expressing types of predicates by letting the return code to be the code of a universe:

$$El(\pi(c, u_i)) = \Pi(El(c), U_i)$$

This allows us to express the type of the `All` predicate in the example above:

$$All : Ty(\Gamma.A.\Pi(A^+, U_0\{!\}).List(A^{++}), 0)$$

It is worth mentioning that one can parameterize this type by the universe level:

$$All_n : Ty(\Gamma.A.\Pi(A^+, U_n\{!\}).List(A^{++}), n)$$

But this parametricity is *external* in the sense that the universe level is indexed outside of the semantic model instead of within the model. Meanwhile, in Agda and Coq, we are able to have *universe polymorphic* definitions, where the universe level is indexed within the language. More concretely, in Agda, we can have

```
data All {a p} {A : Set a} (P : A → Set p) : List A → Set (a ⊔ p)
```

The parameter `a` and `p` are universe levels which is represented internally in the language. Since `All` is parameterized by `a` and `p`, its universe level must be higher than both, and the expression `(a ⊔ p)` precisely denotes this. In Agda, we have the relation `Level : Set` and `Level` can participate in computations.

As we can see, the semantic model does not support the internal representation of universe levels. A proper model achieving this is to extend CwUs, but at the very least it should satisfy the following properties:

(1) There exists a type $Lv = Ty(\top, 0)$.
(2) The set of terms of Lv $Tm(\top, Lv)$ is isomorphic to $\mathbb{N}$.
(3) There exists an operator $Sup(a, b) \in Tm(\top, Lv)$ for $a, b \in Tm(\top, Lv)$ computing the supremum of a and b, and the operator satisfies necessary laws like idempotency, commutativity and associativity.
(4) Interactions between Lv and codes of universe are coherent. That is

$$u(a) \in Tm(\top, U_{El(a)+1})$$
$$El(u(a)) \in Ty(\top, El(a) + 1)$$

where $El(a)$ is an overloaded notation of the function converting a term of type Lv to a $\mathbb{N}$.

To our knowledge, no one has explored a semantic model of internal representation of universe levels.

6.2 Modal Type Theories

In recently years, there are a number of breakthroughs in combining the necessity modality and dependent types, including nominal type theory [Pitts et al. 2014], a Fitch style modal dependent type theory [Birkedal et al. 2020; Gratzer et al. 2019], Cocon [Pientka and Schöpp 2020; Pientka et al. 2019], spatial and crisp type theory [Licata et al. 2018; Shulman 2018]. These type theories are designed for various purposes. Nominal type theory and Cocon handle name representation, Fitch style modal type theory supports metaprogramming, and spatial and crisp type theories find their correspondences in algebraic topology. In these type theories, on top of some standard constructions of dependent types, there is an additional modality $\Box$, satisfying the basic axiom K of modal logic:

$$\Box(S \to U) \to \Box S \to \Box U$$

There are optional axioms associated with this modality. One possible and frequently seen combination is *S4 modal logic*, which is defined by adding the following axioms:

$$\Box T \to T$$
$$\Box T \to \Box\Box T$$

In formal logic and type theory, there are two popular styles of formulation of the $\Box$ modality: the dual-context style and Fitch style. In the next sections, we discuss the latest status of these two styles of study.

6.2.1 Dual-context Modality. In the dual context modal logic, we distinguish two kinds of judgments of facts, the "true" judgments, T `true`, and the "valid" judgments, T `valid`. They are connected by the following two judgments:

$$\frac{T\ \texttt{valid}}{\Box T\ \texttt{true}} \qquad\qquad \frac{\Box T\ \texttt{true}}{T\ \texttt{true}}$$

That is, validity is "stronger" than *truth*, and $\Box$ encapsulates a valid proposition as a true proposition. The dual contexts come into the picture when we formulate the $\Box$ modality in a natural deduction system:

$$\frac{\Delta; \cdot \vdash m : T}{\Delta; \Gamma \vdash \texttt{box}\ m : \Box T} \qquad\qquad \frac{\Delta; \Gamma \vdash m : \Box T \qquad \Delta, u : T; \Gamma \vdash n : U}{\Delta; \Gamma \vdash \texttt{let box}\ u = m\ \texttt{in}\ n : U}$$

Besides this formulation, Davies and Pfenning [2001] also gave a *context-stack* or *Kripke style* formulation which we will not discuss here. Moreover, we only show the introduction and elimination rule for the modality in S4 modal theory. Other variations of modal type theories have different introduction and elimination rules and some have been covered by Kavvos [2017].

In this dual-context formulation, Γ is the usual typing context and we call it the *local* context. The bindings in Γ are true bindings. Δ is the *global* context, the bindings in which are valid bindings. Thus, the first rule says that T is valid if it is derived from valid propositions only. The second rule says that given that $\Box T$ is true, we can use T as a valid proposition to derive a subsequent fact. We consider two categorical models of the dual-context style S4 $\Box$ modality with simple types.

Bierman-de Paiva Categories. This categorical model is outlined in Bierman and de Paiva [2000] and made explicit in Kavvos [2017]. Via the axioms of S4 modal logic, we can see that the axiom K

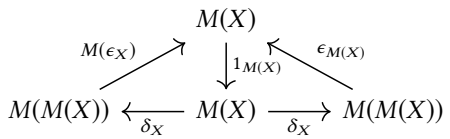
$$\Box(S \to U) \to \Box S \to \Box U$$

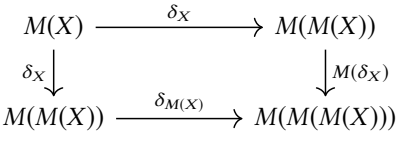
implies the functoriality of $\Box$ and

$$\Box T \to T$$
$$\Box T \to \Box\Box T$$

together imply that $\Box$ is further a comonad. In Section 3.3, we have given a definition of monads. Comonads are the dual of monads. We make the definition explicit:

Definition 6.2. A comonad of a category $\mathcal{C}$ is an endofunctor $M : \mathcal{C} \Rightarrow \mathcal{C}$ with two natural transformations $\epsilon : M \Rightarrow 1_{\mathcal{C}}$ and $\delta : M \Rightarrow M \circ M$. The following axioms hold:





The left diagram expresses identity and the right diagram expresses associativity.

We will need another special notion of functors:

Definition 6.3. A product-preserving functor $F : \mathcal{C} \Rightarrow \mathcal{D}$ is a functor preserving the terminal object and cartesian products:

(1) For $A, B \in \mathcal{C}$, $F(A \times B)$ is a product in $\mathcal{D}$ and $F(\pi_1) : F(A \times B) \Rightarrow F(A)$ and $F(\pi_2) : F(A \times B) \Rightarrow F(B)$ are the projection morphisms.
(2) For a terminal object $\top \in \mathcal{C}$, $F(\top)$ is a terminal object in $\mathcal{D}$.

Then we can define a categorical model for S4 modal type theory:

Definition 6.4. A Bierman-de Paiva Category is a cartesian closed category $\mathcal{C}$, with a product preserving functor F, so that F is a comonad.

The benefit of this model is immediate: it matches our intuition immediately by simply adding to a CCC a comonad, which as we have discussed is a structure the S4 $\Box$ modality necessarily possesses. The product-preserving part of F does not seem essential, and indeed, a (more sophisticated) model given by Bierman and de Paiva [2000] does not require F to be product-preserving. We can tell the product preservation property does not necessarily hold in S4 by considering its elimination rule. Roughly speaking, we call types with projections as eliminators (e.g. the product types $- \times -$) negative types, and the ones using pattern matching as eliminators (e.g. the sum types $- + -$ and $\Box$) positive types. The observation is that positive types do not automatically commute with negative types, but negative types commute with positive types. More intuition can be obtained by considering the sum type $A + -$, which is a positive type. It is obvious that the isomorphism $A + (B \times C) \simeq (A + B) \times (A + C)$ does not generally exist. On the other hand, if we consider $A \times -$ which is a negative type, then the isomorphism $A \times (B + C) \simeq (A \times B) + (A \times C)$ is immediate. In this dual-context formulation, $\Box$ is a positive type, which means the product preservation property of F is a structure which does not generally exist in the syntactic structure. This problem can also be seen in the process of showing the isomorphism $\Box(A \times B) \simeq \Box A \times \Box B$. Consider the effects:

$$f(x : \Box(A \times B)) = (\texttt{let box } u = x \texttt{ in box } \pi_1 u, \texttt{let box } u = x \texttt{ in box } \pi_2 u)$$
$$g(x : \Box A \times \Box B) = \texttt{let box } u = \pi_1 x \texttt{ in let box } v = \pi_2 x \texttt{ in box } (u, v)$$

The problem is that we will not be able to show $f(g(x)) = x$ as it would further require a *commuting conversion* between the projections π_1 and π_2 and `let box`. More additional commuting conversions can be found, for example, in Kavvos [2017, Section 4]. Moreover, this categorical model does not seem to immediately suggest a generalization of the dual-context style S4 $\Box$ modality to the case of CwFs, which would guide us to a dependent S4 modal type theory that we are actively looking into.

Presheaf Categories. Pientka and Schöpp [2020] gave another categorical model of $\Box$ based on a presheaf category, adapting a model given by Hofmann [1997]. $\Box$ in this paper is meant to denote a quoted piece of code. Their syntactic system has a more general construct than $\Box$, which we do not discuss for now, but has the limitation of just having one layer of $\Box$ (e.g. $\Box\Box T$ is unsupported).

A presheaf is a contravariant set-valued functor. That is, given a category C, a presheaf is a functor $C^{op} \Rightarrow \mathcal{S}et$. Recall that C^{op} is the opposite category of C. Fixing a category C, a presheaf category is the category of all presheaves from C^{op} and the natural transformations between them. We are interested in presheaf categories, because usually a presheaf category has more structures than its domain category. For example, a presheaf category is always cartesian closed and has Π and Σ types, but its domain category does not necessarily have any of these. Moreover, a presheaf category, in some sense, "faithfully" characterizes its domain category, due to a very special functor, the Yoneda embedding.

Definition 6.5. Consider the Hom-set of C, $C[-,-]$, is a functor $C^{op} \times C \Rightarrow \mathcal{S}et$. We can curry this functor and obtain a functor to the presheaf category:

$$y : C \Rightarrow \mathcal{S}et^{C^{op}}$$

That is, for an object $X \in C$,

$$y(X) = C[-, X]$$

which is a presheaf returning the Hom-set $C[Y, X]$ given an object $Y \in C$. For a morphism $f : X \Rightarrow Y$, $y(f)$ is the precomposition for any object Z of C:

$$y(f)(Z) : C[Z, Y] \Rightarrow C[Z, X]$$
$$y(f)(Z)(g) = g \circ f$$

The functor y is the Yoneda embedding.

As indicated by its name, y embeds C (which might not have many interesting structures) into a category of set-valued functors. This embedding turns out to be "full and faithful", meaning that everything "happens" in C finds a unique correspondence in the range of the Yoneda embedding and vice versa. What is even better, the Yoneda embedding preserves any structure C possesses. For example, if C has a product $A \times B$, then in the presheaf category, $y(A \times B)$ is a product of yA and yB.

In Pientka and Schöpp [2020], the authors looked into embedding a model of a programming language (the object language) in a language (the meta-language) with much richer structures, e.g. dependent types. The presheaf category is thus used to model this more complex dependently typed meta-language, embedded in which is the simple and small domain category modelling the object language.

Assuming C has a terminal object $\top$, the $\Box$ modality is modelled by the constant endofunctor[6] of $\mathcal{S}et^{C^{op}}$, i.e. $\mathcal{S}et^{C^{op}} \Rightarrow \mathcal{S}et^{C^{op}}$:

$$\Box(F)(X) = F(\top)$$
$$\Box(F)(f) = 1_{F(\top)}$$

[6] An endofunctor has identical domain and codomain category.

One can show that $\Box$ interpreted in this way does satisfy the axioms of S4 modal logic above.

However, this model possesses an unintended structure. Recall that $\Box$ is an endofunctor, so we can apply it iteratively. Let us apply it twice and we can obtain the following equation by expanding the definition:

$$\Box\Box(F)(X) = \Box(F)(X) = F(\top)$$

This equation identifies $\Box\Box F$ and $\Box F$, proving the strong idempotency[7] of $\Box$. However, in the usual S4 modal logic, $\Box\Box T$ and $\Box T$ are generally not identified. It means the $\Box$ modality defined in this way has more structures than we desire. This means this model does not suggest a way to allow more layers of $\Box$s in the system.

Nonetheless, the structure of this presheaf model does generalize to dependent types, which have been explored by the author as a course research project.

Spatial Type Theory. Spatial type theory is a worth mentioning dependent modal type theory defined by Shulman [2018]. Its purpose is to provide a tighter connection between homotopy type theory [Univalent Foundations Program 2013] and algebraic geometry. It is interesting to us because it also employs the dual-context style formulation. In fact, one can regard the semantics given in Pientka and Schöpp [2020] as an embedding into spatial type theory. Though not explicitly mentioned, the $\Box$ modality in spatial type theory is actually idempotent due to a special property it possess, *crisp induction* [Shulman 2018, Lemma 5.1]. This gives another explanation why the semantic model of $\Box$ in Pientka and Schöpp [2020] is idempotent. there is then a question to ask: how do we extend S4 modal type theory with dependent types without bringing in idempotency?

6.2.2 Fitch Style Modality. As opposed to the dual-context style, Fitch style modal logic simply operates on one context. It inherits Fitch's deduction system where a modal deduction has only restricted subdeductions of some form. This idea corresponds to a "marker" or "lock" $\blacksquare$ in the context in natural deduction systems, so that it prevents variable lookups beyond that point:

$$\frac{\Gamma, \blacksquare \vdash T\ \texttt{type}}{\Gamma \vdash \Box T\ \texttt{type}} \qquad \frac{\blacksquare \notin \Gamma'}{\Gamma, x : T, \Gamma' \vdash x : T} \qquad \frac{\Gamma, \blacksquare \vdash m : T}{\Gamma \vdash \texttt{box}\ m : \Box T} \qquad \frac{\Gamma \vdash t : \Box T}{\Gamma, \Gamma' \vdash \texttt{unbox}\ t : T}$$

Intuitively, when we enter a box, $\blacksquare$ has prevented access to all previous known truths, so the drawn conclusion should be valid. Though Fitch style is more intuitive than the dual-context style and looks easier because it only involves one context, as pointed out by Pfenning and Davies [2001], Fitch style formulation has the disadvantage of blurring syntactical distinction between a valid and a true assumption, and thus damage substitution properties of variables and reduction behavior of the system. Nonetheless, since it works with one single context and thus is easier to handle, there are many activities in modelling this style in category theory.

Simply Typed Case. Clouston [2018] gave Fitch style formulation of K and S4 simple modal type theory as shown in the rules above. Consider the introduction rule of $\Box$ and forget about the proof terms for a moment, we have

$$\frac{\Gamma, \blacksquare \vdash T\ \texttt{true}}{\Gamma \vdash \Box T\ \texttt{true}}$$

Recall the adjoint formulation discussed in Section 3.4. This rule seems to suggest that $\Box$, besides being a comonad, is also a right adjoint functor of some functor corresponding to the lock $\blacksquare$.

[7] It is strong because the usual notion of idempotency only requires an isomorphism.

Clouston [2018] used ♦ to represent the left adjoint functor, namely ♦ ⊣ □. Thus the interpretation of the context becomes

$$\begin{aligned}[\![\cdot]\!] &= \top \\ [\![\Gamma, x : T]\!] &= [\![\Gamma]\!] \times [\![T]\!] \\ [\![\Gamma, \blacksquare]\!] &= \blacklozenge [\![\Gamma]\!]\end{aligned}$$

The first two cases are the same as STLC. In the third case, having a lock means applying the left adjoint functor ♦. This interpretation matches the view of adjoint formulation above. In the type interpretation, we interpret □ just as the right adjoint functor:

$$[\![\Box T]\!] = \Box [\![T]\!]$$

This model is quite concise. It is in fact very similar to Bierman-de Paiva Categories given above as a model for the dual-context formulation. Since □ now is a right adjoint functor, we can show that it is automatically product-preserving, following the principle of *right adjoint preserving limits (RAPL)*. Therefore, this model in fact corresponds to some logic stronger than S4. Unlike the dual-context formulation, product preservation is not as a big problem as in Fitch style, □ is formulated as a negative type, as its elimination form is a projection. Thus a question natural arises: would the context-stack formulation in the dual-context style more suitable for categorical semantics, in which □ is also a negative type?

Dependently Typed Case. Despite its generality, one limitation of adjoint functors is that they quantify the relation between two Hom-sets, where the codomain does not depend on the domain. This problem prevents an immediate generalization to dependent types. Birkedal et al. [2020] overcame this limitation by considering a dependent version of adjointness.

Definition 6.6. A category with dependent right adjoint (CwDRA), $\mathcal{C}$, is a CwF with the following additional data:

(1) an endofunctor $L : \mathcal{C} \Rightarrow \mathcal{C}$,
(2) for $\Gamma \in \mathcal{C}$ and $T \in Ty(L(\Gamma))$, $R_\Gamma(T) \in Ty(\Gamma)$, and
(3) an isomorphism $Tm(L(\Gamma), T) \simeq Tm(\Gamma, R_\Gamma(T))$, where the effects of this isomorphism is denoted by $\overrightarrow{t}$ and $\overleftarrow{t}$ and the direction is denoted by the arrows.

such that for $\sigma : \Gamma \Rightarrow \Delta$,

(1)

$$R_\Gamma(T)\{\sigma\} = R_\Delta(T\{L(\sigma)\})$$

(2) given $t : Tm(L(\Gamma), T)$

$$\overrightarrow{t}\{\sigma\} = \overrightarrow{t\{L(\sigma)\}}$$

The idea here is systematic: it extends a CwF with a special right adjoint with sufficient coherence conditions to expose its relation with an endofunctor L. In the interpretation, L is used to represent the marker ■ (just like ♦ in the simply typed case) and R represents the modality □. More concretely,

$$\begin{aligned}[\![\Gamma, \blacksquare]\!] &= L([\![\Gamma]\!]) \\ [\![\Gamma \vdash \Box T \ \ \mathtt{type}]\!] &= R_{[\![\Gamma]\!]}([\![\Gamma, \blacksquare \vdash T \ \ \mathtt{type}]\!]) \\ [\![\Gamma \vdash \mathsf{box}\ m : \Box T]\!] &= \overrightarrow{[\![\Gamma, \blacksquare \vdash m : T]\!]} \\ [\![\Gamma, \blacksquare, \Gamma' \vdash \mathsf{unbox}\ t : T]\!] &= \overleftarrow{[\![\Gamma \vdash t : \Box T]\!]}\{p^k\} \qquad \text{where } |\Gamma'| = k\end{aligned}$$

Moreover, the idea of CwDRAs is very general. It turns out that there are many related concepts are in fact examples of CwDRAs. For example, generalizing the adjointness in a CCC, we can have a similar isomorphism in a CwF too:

$$Tm(\Gamma \times \top.S, T) \simeq Tm(\Gamma, \Pi(S\{!\}, T))$$

for $S \in Ty(\top)$. Here we take $L(\Gamma) = \Gamma \times \top.S$ which concatenates a closed type S to Γ and $R_\Gamma(T) = \Pi(S\{!\}, T)$. That is dependent functions with a closed domain form CwDRAs which captures the intuition from the introduction rule of Π types. More examples are shown in Birkedal et al. [2020, Section 5] related to nominal type theory [Pitts et al. 2014] and guarded recursions [Birkedal et al. 2012]. Finally, Birkedal et al. [2020] also show that this idea naturally extends to CwU, producing categories with universes and dependent right adjoint(CwUDRAs).

One point of dissatisfaction is that Birkedal et al. [2020] only modelled for K modal type theory but did not mention how S4 should be modelled as well. Another related question is whether we are able to motivate a Fitch style dependent S4 modal type theory from the models of CwDRAs.

6.2.3 Connecting the Dual-context and Fitch Styles. It is quite unclear at this moment how the models of modal type theories in the dual-context style and in Fitch style connect. As pointed out previously, the dual-context style has the advantage of better syntactical properties while Fitch style is more active in its semantic study. Thus it would be interesting to see whether the semantic study of both styles would lead to a supreme theory of dependent modal type theory. Since Fitch style has more semantic tools, we could start by looking for an interpretation of the dual-context style $\Box$ modality in CwDRAs or CwUDRAs.

This problem is not immediate, for the dual-context style works with more than one contexts, while our categorical models usually work with one kinds of contexts and substitutions between them. Therefore in order to find a more appropriate model for the dual-context style modality, we need to be able to distinguish two kinds of contexts and relate the global ones with the local ones in the underlying category. Since two kinds of contexts have two different substitution properties, the stability of substitutions is a more complex property than the one we have in a typical CwF.

7 CONCLUSION

In this report, we give a brief overview about the connection between category theory and type theory. After some background in category theory and historical remark, we introduced the relation between simply typed λ calculus and cartesian closed categories and showed the interpretation and the term model of STLC. CCCs are not sufficient to model more complex type theories like the ones with dependent types. Thus we look for more suitable model and that is categories with families, which has stability of substitutions as a coherence condition. It turns out that CwFs has provided a general enough platform for discuss dependent types. We can extend it with a hierarchy of Tarski-like universes, resulting in categories with universes. We also discuss two styles of formulating the necessity modality and their categorical semantics and outline several open questions.

REFERENCES

Agda Team. 2019. Agda 2.6.0.1.

Steve Awodey. 2010. *Category Theory* (2nd ed.). Oxford University Press, Inc., USA.

Hendrik Pieter Barendregt. 1985. *The lambda calculus - its syntax and semantics*. Studies in logic and the foundations of mathematics, Vol. 103. North-Holland.

Gavin M. Bierman and Valeria de Paiva. 2000. On an Intuitionistic Modal Logic. *Studia Logica* 65, 3 (2000), 383–416. https://doi.org/10.1023/A:1005291931660

Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. 2020. Modal dependent type theory and dependent right adjoints. *Math. Struct. Comput. Sci.* 30, 2 (2020), 118–138. https://doi.org/10.1017/S0960129519000197

Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2012. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Log. Methods Comput. Sci.* 8, 4 (2012). https://doi.org/10.2168/LMCS-8(4:1)2012

George Boole. 1854. *An investigation of the laws of thought : on which are founded the mathematical theories of logic and probabilities.* Walton and Maberly, London.

John Cartmell. 1986. Generalised algebraic theories and contextual categories. *Ann. Pure Appl. Log.* 32 (1986), 209–243. https://doi.org/10.1016/0168-0072(86)90053-9

Evan Cavallo, Anders Mörtberg, and Andrew W. Swan. 2020. Unifying Cubical Models of Univalent Type Theory. In *28th EACSL Annual Conference on Computer Science Logic, CSL 2020, January 13-16, 2020, Barcelona, Spain (LIPIcs, Vol. 152)*, Maribel Fernández and Anca Muscholl (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:17. https://doi.org/10.4230/LIPIcs.CSL.2020.14

Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 2001. *Model checking.* MIT Press. http://books.google.de/books?id=Nmc4wEaLXFEC

Ranald Clouston. 2018. Fitch-Style Modal Lambda Calculi. In *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10803)*, Christel Baier and Ugo Dal Lago (Eds.). Springer, 258–275. https://doi.org/10.1007/978-3-319-89366-2_14

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2017. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. *FLAP* 4, 10 (2017), 3127–3170. http://collegepublications.co.uk/ifcolog/?00019

Cyril Cohen and Assia Mahboubi. 2012. Formal proofs in real algebraic geometry: from ordered fields to quantifier elimination. *Logical Methods in Computer Science* 8, 1 (2012). https://doi.org/10.2168/LMCS-8(1:2)2012

Thierry Coquand and Gérard P. Huet. 1988. The Calculus of Constructions. *Inf. Comput.* 76, 2/3 (1988), 95–120. https://doi.org/10.1016/0890-5401(88)90005-3

Haskell B. Curry. 1934. Functionality in Combinatory Logic. *Proceedings of the National Academy of Science* 20, 11 (Nov. 1934), 584–590. https://doi.org/10.1073/pnas.20.11.584

Rowan Davies and Frank Pfenning. 2001. A modal analysis of staged computation. *J. ACM* 48, 3 (2001), 555–604. https://doi.org/10.1145/382780.382785

Nicolaas Govert de Bruijn. 1970. The mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on Automatic Demonstration*, M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 29–61.

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70. https://doi.org/10.1145/3338112

Peter Dybjer. 1995. Internal Type Theory. In *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers (Lecture Notes in Computer Science, Vol. 1158)*, Stefano Berardi and Mario Coppo (Eds.). Springer, 120–134. https://doi.org/10.1007/3-540-61780-9_66

Gerhard Gentzen. 1934. *Untersuchungen über das logische Schließen.* Ph.D. Dissertation. Georg-August-Universität Göttingen.

Gerhard Gentzen. 1969. 3. Investigations into Logical Deduction. In *The Collected Papers of Gerhard Gentzen*, M.E. Szabo (Ed.). Studies in Logic and the Foundations of Mathematics, Vol. 55. Elsevier, 68 – 131. https://doi.org/10.1016/S0049-237X(08)70822-X

Paolo G. Giarrusso, Léo Stefanesco, Amin Timany, Lars Birkedal, and Robbert Krebbers. 2020. Scala step-by-step: soundness for DOT with step-indexed logical relations in Iris. *Proc. ACM Program. Lang.* 4, ICFP (2020), 114:1–114:29. https://doi.org/10.1145/3408996

Jean-Yves Girard. 1971. Une Extension De L'Interpretation De Gödel a L'Analyse, Et Son Application a L'Elimination Des Coupures Dans L'Analyse Et La Theorie Des Types. In *Proceedings of the Second Scandinavian Logic Symposium*, J.E. Fenstad (Ed.). Studies in Logic and the Foundations of Mathematics, Vol. 63. Elsevier, 63 – 92. https://doi.org/10.1016/S0049-237X(08)70843-7

Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. 2019. Implementing a modal dependent type theory. *PACMPL* 3, ICFP (2019), 107:1–107:29. https://doi.org/10.1145/3341711

Robert Harper. 2016. *Practical Foundations for Programming Languages (2nd. Ed.).* Cambridge University Press. https://www.cs.cmu.edu/%7Erwh/pfpl/index.html

Arend Heyting. 1930. *Die formalen Regeln der intuitionistischen Logik.* Verlag der Akademie der Wissenschaften, Berlin.

Arend Heyting. 1934. *Mathematische Grundlagenforschung Intuitionismus Beweistheorie.* Springer Berlin Heidelberg, Berlin, Heidelberg.

Martin Hofmann. 1997. *Syntax and Semantics of Dependent Types.* Cambridge University Press, 79–130. https://doi.org/10.1017/CBO9780511526619.004

Martin Hofmann. 1999. *Type systems for polynomial-time computation.* Ph.D. Dissertation. Technische Universität Darmstadt.

William A. Howard. 1980. The formulae-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, λ-calculus and Formalism*, J. Hindley and J. Seldin (Eds.). Academic Press, 479–490.

Bart Jacobs. 1993. Comprehension Categories and the Semantics of Type Dependency. *Theor. Comput. Sci.* 107, 2 (1993), 169–207. https://doi.org/10.1016/0304-3975(93)90169-T

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. https://doi.org/10.1145/3158154

G. A. Kavvos. 2017. Dual-context calculi for modal logic. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017.* IEEE Computer Society, 1–12. https://doi.org/10.1109/LICS.2017.8005089

Andrey Kolmogoroff. 1932. Zur Deutung der intuitionistischen Logik. *Mathematische Zeitschrift* 35 (1932), 58–65.

Robbert Krebbers and Bas Spitters. 2011. Type classes for efficient exact real arithmetic in Coq. *Logical Methods in Computer Science* 9, 1 (2011). https://doi.org/10.2168/LMCS-9(1:01)2013

Joachim Lambek. 1974. Functional completeness of cartesian categories. *Annals of Mathematical Logic* 6, 3 (1974), 259 – 292. https://doi.org/10.1016/0003-4843(74)90003-5

Joachim Lambek. 1980. From λ-calculus to cartesian closed categories. In *To H.B. Curry: Essays on Combinatory Logic, λ-calculus and Formalism*, J. Hindley and J. Seldin (Eds.). Academic Press, 375–402.

Joachim Lambek. 1985. Cartesian Closed Categories and Typed Lambda- calculi. In *Combinators and Functional Programming Languages, Thirteenth Spring School of the LITP, Val d'Ajol, France, May 6-10, 1985, Proceedings (Lecture Notes in Computer Science, Vol. 242)*, Guy Cousineau, Pierre-Louis Curien, and Bernard Robinet (Eds.). Springer, 136–175. https://doi.org/10.1007/3-540-17184-3_44

F. William Lawvere. 1963. *Functorial Semantics of Algebraic Theories.* Ph.D. Dissertation. Columbia University.

F. William Lawvere. 1969. Adjointness in Foundations. *Dialectica* 23, 3/4 (1969), 281–296. http://www.jstor.org/stable/42969800

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 6355)*, Edmund M. Clarke and Andrei Voronkov (Eds.). Springer, 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. 2018. Internal Universes in Models of Homotopy Type Theory. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK (LIPIcs, Vol. 108)*, Hélène Kirchner (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:17. https://doi.org/10.4230/LIPIcs.FSCD.2018.22

Zhaohui Luo. 2012. Notes on universes in type theory. (2012). http://www.dcs.rhul.ac.uk/home/zhaohui/universes.pdf Lecture notes for a talk at Institute for Advanced Study, Princeton.

Saunders MacLane. 1971. *Categories for the Working Mathematician.* Springer-Verlag, New York. ix+262 pages. Graduate Texts in Mathematics, Vol. 5.

Per Martin-Löf. 1984. *Intuitionistic type theory.* Studies in proof theory, Vol. 1. Bibliopolis.

Eugenio Moggi. 1991. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55 – 92. https://doi.org/10.1016/0890-5401(91)90052-4 Selections from 1989 IEEE Symposium on Logic in Computer Science.

Erik Palmgren. 1998. On universes in type theory. *Twenty-five years of constructive type theory* 36 (1998), 191–204.

Frank Pfenning and Rowan Davies. 2001. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science* 11, 4 (2001), 511–540. https://doi.org/10.1017/S0960129501003322

Brigitte Pientka and Ulrich Schöpp. 2020. Semantical Analysis of Contextual Types. In *Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12077)*, Jean Goubault-Larrecq and Barbara König (Eds.). Springer, 502–521. https://doi.org/10.1007/978-3-030-45231-5_26

Brigitte Pientka, David Thibodeau, Andreas Abel, Francisco Ferreira, and Rébecca Zucchini. 2019. A Type Theory for Defining Logics and Proofs. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019.* IEEE, 1–13. https://doi.org/10.1109/LICS.2019.8785683

Andrew M. Pitts, Justus Matthiesen, and Jasper Derikx. 2014. A Dependent Type Theory with Abstractable Names. In *Ninth Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2014, Brasília, Brazil, September 8-9, 2014 (Electronic Notes in Theoretical Computer Science, Vol. 312)*, Mauricio Ayala-Rincón and Ian Mackie (Eds.). Elsevier, 19–50. https://doi.org/10.1016/j.entcs.2015.04.003

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974 (Lecture Notes in Computer Science, Vol. 19)*, Bernard Robinet (Ed.). Springer, 408–423. https://doi.org/10.1007/3-540-06859-7_148

Robert A. G. Seely. 1983. HYPERDOCTRINES, NATURAL DEDUCTION AND THE BECK CONDITION. *Mathematical Logic Quarterly* 29, 10 (1983), 505–542. https://doi.org/10.1002/malq.19830291005 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/malq.19830291005

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

Michael Shulman. 2018. Brouwer's fixed-point theorem in real-cohesive homotopy type theory. *Mathematical Structures in Computer Science* 28, 6 (2018), 856–941. https://doi.org/10.1017/S0960129517000147

Julien Signoles. 2009. Foncteurs impératifs et composés: la notion de projets dans Frama-C. In *JFLA 2009, Vingtièmes Journées Francophones des Langages Applicatifs, Saint Quentin sur Isère, France, January 31 - February 3, 2009. Proceedings (Studia Informatica Universalis, Vol. 7.2)*, Alan Schmitt (Ed.). 245–280.

The Coq Development Team. 2019. The Coq Proof Assistant, version 8.10.0. https://doi.org/10.5281/zenodo.3476303

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

Philip Wadler. 2015. Propositions as types. *Commun. ACM* 58, 12 (2015), 75–84. https://doi.org/10.1145/2699407

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093