The Equivalence of Typed λ Calculi and Cartesian Closed Categories

Kei Imada

January 2019

Abstract

Typed λ calculi are abstract programming languages that form the bases for typed functional programming languages like Haskell. It enables the formalization of type system designs in programming languages, which helps reduce possibilities for bugs in computer programs. It turns out that typed λ calculi are structurally equivalent to a kind of category in category theory called the Cartesian closed category. This entails the potential to use category theory to prove properties of a type system in functional programming languages, and vice versa. Most texts on this topic are difficult to understand without prior knowledge of graduate level mathematics and programming language theory. So we seek to make this topic accessible to undergraduate mathematics majors, assuming only basic knowledge of set theory and group theory.

Introduction

What do we think when we're asked, "so what's the relationship between mathematics and computer science?" Some may think of the field of computation theory, where we define the notion of a Turning machine to abstract computers and prove that some problems are computationally hard or impossible. Others may think of statistics, employed in machine learning and artificial intelligence to create predictive models and heuristics. Other than these two, we have cryptography, linear programming, and a plethora of other applications.

Despite our knowledge of the diverse applications of mathematics in computer science, many who study both mathematics and computer science are unaware of the inherent connection between category theory and functional programming languages. It is surprising to find that a field of mathematics as theoretical as category theory can be directly applied to something so practical.

Originating from algebraic topology around the 1940s, category theory is a relatively new field of mathematics. In short, category theory is a branch of mathematics generalizing mathematical structures and structure-preserving maps. Over the past few decades, this field, deemed one of the most theoretical, has demonstrated its ability to act as a powerful conceptual framework, allowing us to see the relationships between a collection of structures of any kind.

This paper is about building a bridge between two massive continents, mathematics and computer science. In particular, we will traverse the borders of category theory and programming language theory, or more specifically, where the land of Cartesian closed categories meets the land of typed λ calculi.

In order to build this bridge, we must first understand the two coasts it connects. So we will first define Cartesian closed categories. To do that, we discuss the fundamentals of category theory. We will define what a category is, and then provide important definitions to understand the rest of the paper. This includes the utility of diagrams, isomorphisms, terminal objects, products, and exponentiation. After we define these ideas, we find that the definition of Cartesian closed categories is effortless.

We then describe the concept of the bridge we are constructing, namely the equivalence between categories. We define mappings between categories called functors, and equivalences between functors called natural isomorphisms.

We then define the other side of the bridge, typed λ calculi, which are abstract programming languages that have types, expressions, and equations.

After we build a foundation in category theory and programming language theory, we finally construct the bridge between the lands of Cartesian closed categories and typed λ calculi, in other words, the equivalence between the two ideas.

So without further ado, let's get on with it!

Basic Category Theory

In order to truly appreciate the applications of category theory, we must first understand category theory. Here are all the category theory we need to know to understand this paper.

Categories

Category theory is a study of objects and arrows between them, called morphisms which together are called a **category**.

Definition 1 (Category)**.** $\mathbf{C}$ is a **category** if it has the following:

1. A collection of **objects**, denoted $\operatorname{ob}\mathbf{C}$.
2. A collection of **morphisms** between objects, denoted $\operatorname{Mor}\mathbf{C}$. We often want to examine the collection of morphisms between two objects, say $A, B \in \operatorname{ob}\mathbf{C}$, which we will denote $\mathbf{C}(A, B)$.
3. Well defined operations mapping each arrow f its domain $\operatorname{dom} f$ and its codomain $\operatorname{cod} f$. We denote $f : A \to B$ to show that $\operatorname{dom} f = A$ and $\operatorname{cod} f = B$.
4. An associative **composition** binary operator for morphisms. This operator assigns two morphisms, f and g with $\operatorname{cod} f = \operatorname{dom} g$, a composite morphism $g \circ f : \operatorname{dom} f \to \operatorname{cod} g$. This operator must also be associative, in other words, for all $f : A \to B$, $g : B \to C$, and $h : C \to D$,
$$(f \circ g) \circ h = f \circ (g \circ h).$$
In most cases, we define morphism equality as the set-theoretic function equality.
5. Unique **Identity morphisms** for each object. In other words, for every object A, there is a morphism $\mathbf{1}_A : A \to A$ where for any arrow $f : A \to B$,
$$\mathbf{1}_B \circ f = f \quad \text{and} \quad f \circ \mathbf{1}_A = f.$$
We will call this equality the *identity law.*

Example (**Set**)**.** The objects of **Set** are sets. The morphisms of **Set** are *total functions*, functions that are well defined for all input, with specified domains and codomains. The composition is the usual composition between two functions, and the identity morphisms are the identity functions.

Remark. We need to specify the domain and the codomain to make sure our dom and cod operators are well defined. Take the total function that maps $x \in \boldsymbol{R}$ to x^2 for example. f maps real numbers to real numbers, so $\operatorname{cod} f = \boldsymbol{R}$. On the other hand, f also maps real numbers to $\boldsymbol{R}^{\geq 0}$, so $\operatorname{cod} f = \boldsymbol{R}^{\geq 0}$.

Here's an example of verifying that something is a category:

Proposition 1. **Set** *is a category.*

Proof. Criteria 1 through 3 are trivial (we just defined them above), so we focus on criteria 4 and 5.

4. By definition, the composition morphism of total functions $f : A \to B$ and $g : B \to C$ is $g \circ f : A \to C, x \mapsto g(f(x))$. The associative property of total function compositions as well as the property that total functions are closed under compositions are both well known in set theory.
5. For any set A, we know that identity functions are total functions. We can also check the identity law with simple calculations.

With this we verified that **Set** is a category. □

It so turns out that many categories are comprised of objects as "sets with structure" and morphisms as "structure preserving maps." Table 1 lists several that may be familiar.

Category	Objects	Arrows
Set	sets	total functions
Grp	groups	group homomorphisms
Vect	vector spaces	linear transforms
Top	topological spaces	continuous spaces

Table 1: Categories where objects are sets with structure and morphisms are structure preserving maps

We end the introduction to categories with an example of peculiar categories which show an exception to the intuition above.

Example. Let G be a group. The category $\mathbf{G}$ has a single object, and the morphisms are the elements of G. The identity element $e \in G$ is the identity for the object, and the composition between two morphisms (elements of G) is the binary operation $\cdot$ associated with G.

We can define categories based on what we want to examine. If we want to study groups, we may look at **Grp**, and if we want to investigate the elements of a specific group, we would instead analyze the example above.

Diagrams

Many of our constructions in category theory will be complicated without a way to visually represent statements about objects and morphisms. This is why we use a graphical style of presentation, called a **diagram**.

Definition 2 (Diagram)**.** Given a category $\mathbf{C}$, a diagram is a collection of vertices and directed edges, where every vertex v is labeled with an object A and every edge e is labeled with a morphism $f : A \to B$ such that its head is A and tail is B.

Here's a simple example describing $f : A \to B$.

$$A \xrightarrow{f} B$$

To fully utilize diagrams, we must be able to translate properties of categorical constructions to diagrams.

We do this by stating that a particular diagram **commutes**.

Definition 3 (Commuting diagram)**.** Given a diagram D in a category $\mathbf{C}$, D commutes if for every pair vertices $A, B \in V(D)$, all morphisms representing paths in the diagram from A to B are equal.

Example. The diagram below commutes if $f' \circ g = g' \circ f$.

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "g"'] & B \arrow[d, "g'"] \\ C \arrow[r, "f'"'] & D \end{tikzcd}$$

Remark. We can always find a representing morphism from a path by the definition of the category, namely compositions.

To motivate the power of diagrams, we define another type of category, the arrow category.

Definition 4 (Arrow category)**.** Given category $\mathbf{C}$, its corresponding arrow category, denoted $\mathbf{C}^{\rightarrow}$, is a category where $\operatorname{ob}\mathbf{C}^{\rightarrow} = \operatorname{Mor}\mathbf{C}$ are the objects. A $\mathbf{C}^{\rightarrow}$-morphism between $\mathbf{C}$-morphisms $f : A \to B$ and $f' : A' \to B'$ is a pair of $\mathbf{C}$-morphisms $(a,\, b)$ where the diagram below commutes.

$$\begin{tikzcd} A \arrow[r, "a"] \arrow[d, "f"'] & A' \arrow[d, "f'"] \\ B \arrow[r, "b"'] & B' \end{tikzcd}$$

The composition of $\mathbf{C}^{\rightarrow}$-morphisms $(a,\, b) : (f : A \to B) \to (f' : A' \to B')$ and $(a',\, b') : (f' : A' \to B') \to (f'' : A'' \to B'')$ is $(a',\, b') \circ (a,\, b) = (a' \circ a,\, b' \circ b)$, which corresponds to "gluing" the two squares together:

$$\begin{tikzcd} A \arrow[r, "a"] \arrow[d, "f"'] & A' \arrow[r, "a'"] \arrow[d, "f'"] & A'' \arrow[d, "f''"] \\ B \arrow[r, "b"'] & B' \arrow[r, "b'"'] & B'' \end{tikzcd}$$

Given a $\mathbf{C}$-object $f : A \to B$, its identity is $(\mathbf{1}_A,\, \mathbf{1}_B)$.

The diagrams give a helpful visual that helps us understand how the arrow category is constructed. Because of their utility, we use diagrams extensively throughout this paper.

Morphisms

In category theory, we focus on morphisms rather than objects, as we will see in later sections. In general, category theory doesn't examine the internal structure of objects, but rather, treat objects as black boxes. This is because objects can have various types of structure. For example, they are sets in **Set**, elements in a group G when G is treated as a category, and even morphisms in a $\mathbf{C}^{\rightarrow}$ category. So instead we focus on the properties of morphisms between objects. The next four sections show constructions in category theory that are analogous to common structures in mathematics. But before that, we will define what it means for two objects to be the "same," or **isomorphic**.

Definition 5 (Isomorphism)**.** A morphism $f : A \to B$ is an isomorphism if $\exists\ f^{-1} : B \to A$ such that $f^{-1} \circ f = \mathbf{1}_A$ and $f \circ f^{-1} = \mathbf{1}_B$. Two objects A, B are called isomorphic if there exists an isomorphism between them.

Example (Isomorphisms in **Set**)**.** Isomorphisms in **Set** are bijective functions. By definition, an isomorphism $f : A \to B$ is a total function that has an inverse f^{-1} such that $f^{-1} \circ f = \mathbf{1}_A$ and $f \circ f^{-1} = \mathbf{1}_B$.

Terminal Objects

A **terminal object** in a category is an object where every object has a unique morphism to it.

Definition 6 (Terminal object)**.** An object $\mathbf{1}$ is terminal if, for every object A, there is one and only one morphism from A to $\mathbf{1}$.

Example (Terminal objects in **Set**)**.** Every one-element set $\{x\}$ is a terminal object in **Set**. Given any set $A \in \operatorname{ob}\mathbf{Set}$, there is only one total function from $f : A \to \{x\}$, namely $a \mapsto x$. The empty function is vacuously a total function.

Terminal objects provide a categorical analogue of singletons, as shown in the previous **Set** example.

Products

Here we provide a category theory analogue of the Cartesian product of sets, the **product** of objects in a category. In set theory we define the product of two sets as

$$A \times B = \{(a,\, b) : a \in A \text{ and } b \in B\}.$$

However, we shouldn't use this definition in category theory because we want to treat every object as a black box, as mentioned in Morphisms. Instead, we should define products with the properties of morphisms between them and other objects.

What we realize is that when we create a set product $A \times B$, we also implicitly create *projection functions* $\pi_1 : A \times B \to A,\ (a,\, b) \mapsto a$ and $\pi_2 : A \times B \to B,\ (a,\, b) \mapsto b$. That means we can think of a product as a tuple $(A \times B,\, \pi_1,\, \pi_2)$.

We also find that for every set C and functions $f : C \to A$ and $g : C \to B$, we can find a function $h : C \to A \times B$ such that $f = \pi_1 \circ h$ and $g = \pi_2 \circ h$, namely, $x \mapsto (f(x),\, g(x))$. This function h is called a *mediating morphism* and is denoted $\langle f,\, g\rangle \coloneqq h$. We now define products with this motivation.

Definition 7 (Products)**.** Given two objects A and

B in a category $\mathbf{C}$, its product, denoted $A \times B$, is an object of $\mathbf{C}$ with two projection functions $\pi_1 : A \times B \to A$, $\pi_2 : A \times B \to B$ such that for every object C and morphisms $f : C \to A$ and $g : C \to B$, $\exists!\ \langle f,\, g\rangle : C \to A \times B$, called a *mediating morphism*, such that $\pi_1 \circ \langle f,\, g\rangle = f$ and $\pi_2 \circ \langle f,\, g\rangle = g$.

We can show this definition as the following commuting diagram.

$$\begin{tikzcd} & C \arrow[dl, "f"'] \arrow[d, dashed, "\langle f,g\rangle"] \arrow[dr, "g"] & \\ A & A \times B \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & B \end{tikzcd}$$

We say that **a category C has products** if for any two objects A and B, there exists a product.

Now we show that a set product $A \times B$ fits the definition above.

Proposition 2. *In* **Set***, the product of objects A and B is their Cartesian product $A \times B$.*

Proof. Let X be a set and $x_1 : X \to A$ and $x_2 : X \to B$. Consider the function $\langle x_1,\, x_2\rangle : X \to A \times B$, $x \mapsto (x_1(x),\, x_2(x))$. By definition $x_1 = \pi_1 \circ \langle x_1,\, x_2\rangle$ and $x_2 = \pi_2 \circ \langle x_1,\, x_2\rangle$. To show uniqueness of $\langle x_1,\, x_2\rangle$, suppose there is a morphism $f : X \to A \times B$ such that $x_1 = \pi_1 \circ f$ and $x_2 = \pi_2 \circ f$. Take $x \in X$, and denote $\langle x_1,\, x_2\rangle(x) = (a,\, b)$ and $f(x) = (a',\, b')$. By definition $a = \pi_1((a,\, b)) = \pi_1(f(x)) = x_1(x) = \pi_1(\langle x_1,\, x_2\rangle(x)) = \pi_1((a',\, b')) = a'$, and similarly, $b = b'$, and thus $(a,\, b) = (a',\, b')$. So $f = \langle x_1,\, x_2\rangle$. Since $(X,\, f_1,\, f_2)$ was arbitrary, $(A \times B,\, \pi_1,\, \pi_2)$ is the product of A and B in **Set**. □

Products are not unique. For example, in **Set**, $(A \times B,\, \pi_1,\, \pi_2)$ and $(B \times A,\, \pi_2,\, \pi_1)$ are both products of sets A and B. However, there are bijections between $A \times B$ and $B \times A$, $(a,\, b) \mapsto (b,\, a)$. This means that they are the same up to isomorphism. It turns out that in any category, every product of two objects is unique up to unique isomorphisms.

Proposition 3. *Products of two objects are unique up to unique isomorphisms.*

Proof. Let $\mathbf{C}$ be a category and $A, B, C, C' \in \operatorname{ob}\mathbf{C}$ such that C and C' are products of A and B. Let $(\pi_1,\, \pi_2)$ and $(\pi_1',\, \pi_2')$ be the respective pairs of projection functions. By definition, there exists unique mediating morphisms $f = \langle \pi_1,\, \pi_2\rangle : C \to C'$ and $g = \langle \pi_1',\, \pi_2'\rangle : C' \to C$. Consider the diagram below.

$$\begin{tikzcd} C \arrow[r, "f"] \arrow[dr, "\mathbf{1}_C"'] & C' \arrow[d, "g"] \arrow[dr, "\mathbf{1}_{C'}"] & \\ & C \arrow[r, "f"'] & C' \end{tikzcd}$$

By the uniqueness of mediating arrows, $f \circ g = \mathbf{1}_C$ and $g \circ f = \mathbf{1}_{C'}$, so the diagram commutes. With the same uniqueness we deduce that f and g are the only functions that satisfy this condition. We have found a unique isomorphism between C and C', so products of two objects are unique up to unique isomorphisms. □

Remark. We can replace products with terminal objects and mediating arrows with the unique morphisms with terminal objects as codomains. With this we find a more general theorem:

Proposition 4. *All terminal objects are unique up to unique isomorphisms.*

Proof. Essentially the same as the proof above. □

The consequence of this theorem is that, if it exists, we can refer to *the product* of A and B, without having to worry about the statement being well defined. We will denote the product object of A and B as $A \times B$, courtesy of the Cartesian product notation in set theory.

It will be useful in the next section to talk about morphisms between two product objects

Definition 8 (Product map)**.** If $A \times B$ and $C \times D$ are product objects, for every pair of functions $f : A \to C$ and $g : B \to D$ there exists a product map $f \times g : A \times B \to C \times D$, defined as $\langle f \circ \pi_1,\, g \circ \pi_2\rangle$[1].

Exponential objects

The last object we must examine to understand Cartesian closed categories abstracts the notion of the set of morphisms from one object to another. In set theory it would be equivalent to the set of all functions from set A to set B.

$$B^A = \{f : A \to B\}$$

As we did with products, we would like to characterize B^A with morphisms instead of elements.

When defining products, we found that a product is accompanied with two projection morphisms. Similarly, we notice that B^A is associated with a special evaluation morphism $\operatorname{eval} : (B^A \times A) \to B$, $(f,\, a) \mapsto f(a)$.

The crux of the categorical definition is based on the observation that this eval function has the property that for every $g : X \times A \to B$, there exists a unique function $\operatorname{curry}(g) : X \to B^A$ such that the diagram below commutes.

$$\begin{tikzcd} B^A \times A \arrow[r, "\operatorname{eval}"] & B \\ X \times A \arrow[u, "\operatorname{curry}(g)\times\mathbf{1}_A"] \arrow[ur, "g"'] & \end{tikzcd}$$

[1]We don't need to say which products the π_i's are associated with because of the *typed* nature of category theory.

To further explain, for every $x \in X$ we have a function $g_x : A \to B$, $a \mapsto g(x, a)$, in other words, the function that fixes the first argument of g to be x. curry(g) is then defined as $x \mapsto g_x$.

We now show that this definition of curry(g) is correct and is unique.

Proposition 5. curry$(g) : X \to B^A$, $x \mapsto g_x$ *is the unique function that makes the diagram above commute.*

Proof. For any $(x, a) \in X \in A$, we have

$$\begin{aligned}(\mathrm{eval} \circ (\mathrm{curry}(g) \times \mathbf{1}_A))(c, a) &= \mathrm{eval}(\mathrm{curry}(g)(x), a) \\ &= \mathrm{eval}(g_x, a) \\ &= g(x, a).\end{aligned}$$

This shows that the above diagram commutes. We show uniqueness by noting that $g(x, a) = \mathrm{eval}(\mathrm{curry}(g)(x), a) = (\mathrm{curry}(g)(x))(a)$, curry$(g)(x)$ maps $a \mapsto g(x, a)$, so curry$(g)(x) = g_x$. □

Now that we have a motivation for the definition of exponential objects in **Set**, let us generalize this for a general category **C** with products.

Definition 9 (Exponential objects)**.** Let **C** be a category with products and let A and B be objects of **C**. An object B^A is called the exponential object if there is a morphism $\mathrm{eval}_{AB} : (B^A \times A) \to B$ such that for any object X and morphism $g : (X \times A) \to B$, there is a unique morphism curry$(g) : C \to B^A$ where the diagram below commutes.

$$\begin{tikzcd}
B^A \times A \arrow[r, "\mathrm{eval}"] & B \\
C \times A \arrow[u, "\mathrm{curry}(g) \times \mathbf{1}_A"] \arrow[ur, "g"'] &
\end{tikzcd}$$

Proposition 6. *In* **Set***, the exponential object of A and B is $B^A = \{f : A \to B\}$.*

Proof. We need only state that curry as defined above fits the bill, because we showed in Proposition 5 that curry is the unique morphism that makes the above diagram commute in **Set**. □

Similarly to products, we say that a category **C** **has exponentiation** if it has an exponential B^A for every pair of objects A and B.

Cartesian Closed Categories

Now that we defined categories, terminal objects, products, and exponentiation, we can give the full definition for Cartesian closed categories.

Definition 10 (Cartesian closed categories)**.** A category **C** is called a Cartesian closed category if it has products, exponentiation, and a terminal object.

Example. Set is a Cartesian closed category. The terminal object is a singleton $\{a\}$, the product of sets A and B is its Cartesian product $A \times B = \{(a, b) : a \in A, b \in B\}$, and their exponential object is the function set $B^A = \{f : A \to B\}$.

Later we will find that Cartesian closed categories are inherent in all functional programming languages in the form of their type systems. In order to do that, we must talk about what it means for two categories to be the same, or *equivalent*, which we will do in the next section.

Equivalence of Categories

Here we define what it means for two categories to be **equivalent**. To do this in a category theoretic fashion, we must define mappings analogous to morphisms between categories, called **functors**, then define equivalences between functors, or **natural isomorphisms**.

Functors

Functors give structure preserving mappings between categories. Since categories have both objects and morphisms, functors must map them both.

Definition 11 (Functors)**.** A functor F from category **C** to category **D**, denoted $F : \mathbf{C} \to \mathbf{D}$, maps objects of **C** to objects of **D** and morphisms of **C** to morphisms of **D**, such that for all $A \in \mathrm{ob}\,\mathbf{C}$ and composable morphisms $f, g \in \mathrm{Mor}\,\mathbf{C}$,

1. $F(\mathbf{1}_A)$, and
2. $F(f \circ g) = F(f) \circ F(g)$.

Example (Forgetful functors)**.** Take the functor F : **Grp** → **Set** where each group is mapped to its underlying set and each homomorphism to its underlying function. This is called a forgetful functor because the functor "forgets" some structures of the domain, for example, the associative binary operation in **Grp**.

Example (Power set)**.** The Power set functor is $\mathcal{P}$: **Set** → **Set**, where each set is mapped to its powerset, and each $f : A \to B$ is mapped to a function $\mathcal{P}(f) : \mathcal{P}(A) \to \mathcal{P}(B)$, $X \mapsto f(X)$. Such functors that map categories to itself is called an *endofunctor*.

Remark. We can think of functors as morphisms between categories. In fact, they give rise to the categories of categories **Cat** where the objects are categories and morphisms are functors. To avoid contradictions such as Russel's paradox, we generally distinguish between *large* and *small* categories. Small categories are those whose collections of objects and morphisms are both sets. Then **Cat** is the category of all small categories, which is itself a large category.

Example (Identity functors)**.** Given a category **C**, the identity functor $\mathbf{1_C}$ is the identity morphism of **C** when considered as an object in the category **Cat**. In other words, it is the functor that maps objects and morphisms to themselves.

Natural Isomorphisms

In essence, **natural isomorphisms** form equivalences between two functors where we view a collection of isomorphisms as a form of equivalence.

Definition 12 (Natural Isomorphisms)**.** A natural isomorphism between functors $F : \mathbf{C} \to \mathbf{D}$ and $G : \mathbf{C} \to \mathbf{D}$ is denoted $\eta : F \Rightarrow G$, and maps each object c in $\mathbf{C}$ an isomorphism $\eta_c : F(c) \to G(c)$ in $\mathbf{D}$ such that for any morphism $f : x \to y$ in $\mathbf{C}$, the diagram below commutes.

$$\begin{tikzcd}
F(x) \arrow[r, "F(f)"] \arrow[d, "\eta_x"] & F(y) \arrow[d, "\eta_y"] \\
G(x) \arrow[r, "G(f)"] & G(y)
\end{tikzcd}$$

Nontrivial examples of natural isomorphisms are hard to come by. So we give a trivial example.

Example. Needless to say, there exists a natural isomorphism between a functor $F : \mathbf{C} \to \mathbf{D}$ to itself, where it maps each $c \in \operatorname{ob} \mathbf{C}$ an identity morphism $\mathbf{1}_{F(c)}$. The proof is clear with the commuting diagram below.

$$\begin{tikzcd}
F(x) \arrow[r, "F(f)"] \arrow[d, "\mathbf{1}_{F(x)}"] & F(y) \arrow[d, "\mathbf{1}_{F(y)}"] \\
F(x) \arrow[r, "F(f)"] & F(y)
\end{tikzcd}$$

Remark. Relaxing the definition from isomorphisms to morphisms gives us *natural transformations.*

Equivalence of Categories

Now that we know what functors and natural isomorphisms are, we can define the equivalence of two categories.

Definition 13 (Equivalence of categories)**.** The equivalence of $\mathbf{A}$ and $\mathbf{B}$ is a pair of functors, $F : \mathbf{A} \to \mathbf{B}$ and $G : \mathbf{B} \to \mathbf{A}$, and natural isomorphisms $\varepsilon : F \circ G \Rightarrow \mathbf{1_A}$ and $\eta : \mathbf{1_B} \Rightarrow G \circ F$.

Note the similarity between this definition and the definition of isomorphisms, where the significant difference is the natural isomorphisms being the equivalence of functors.

Since it's hard to give a nontrivial, clear example, we give the most trivial example, that a category is equivalent to itself

Example. The equivalence of category $\mathbf{C}$ to itself is $(\mathbf{1_C}, \mathbf{1_C})$ where $\mathbf{1_C} : \mathbf{C} \to \mathbf{C}$ is the identity functor.

Remark. Relaxing the definition from natural isomorphisms to natural transformations gives us *adjoints.*

We now turn to typed λ calculi, and provide its definition. With the definition, we will construct an equivalence between Cartesian closed categories and typed λ calculi.

Typed λ Calculus

Similar to Turing machines formalizing sequential computation models, typed λ calculi formalize functional computation models. They are foundational programming language models that form the bases of typed functional programming languages like Haskell. In short, a typed λ calculus $\mathscr{L}$ is a three tuple of **types**, **expressions**, and **equations**. For simplicity, let $\mathscr{L} = (T, \text{Expr}, E)$ denote the typed λ calculus we are defining throughout this section.

Types

A type is an attribute we assign to expressions. This is similar to the way we call 1 a `Number` and a coffee a `Drink`. The **types** of $\mathscr{L}$ has two rules: the existence of a Unit type and that they are closed under the pairing and the exponentiation operators. Formally, they are defined below.

Definition 14 (Types)**.** T are the types of $\mathscr{L}$ if

1. $\text{Unit} \in T$
2. $\forall\ A, B \in T \Rightarrow A \times B$ and $B^A \in T$

Remark. There may be basic types other than Unit. For example, Scott and Lambek assume the numbers type as another type [4] and Pierce assumes a general version of any number of types [11]. In fact, this **extensibility** of the typed λ calculus is what makes the definition so powerful that most if not all typed functional languages can be modeled with typed λ calculi.

Expressions

Informally, expressions are building blocks used in programming using the typed λ calculus. The expressions Expr of $\mathscr{L}$ are generated from variables and expression forming rules with these constraints. All expressions also are assigned a type according to *type rules*, and with it we also give the *expression forming rules.* We will denote $a : A$ to denote that an expression $a \in \text{Expr}$ and has type A.

Definition 15 (Expression)**.** Expr are the expressions of $\mathscr{L}$ if for all $M, M_1, M_2 \in \text{Expr}$ and $A \in T$, the **type rules** and **expression forming rules** below are satisfied.

Unit $() : \text{Unit}$
Var $x_i^A : A\ \forall\ i \in \boldsymbol{N}$
Func $M : B \Rightarrow (\lambda x : A.\ M) : B^A$
Appl $M_1 : B^A,\ M_2 : A \Rightarrow (M_1\ M_2) : B$
Pair $M_1 : A, M_2 : B \Rightarrow (M_1,\ M_2) : A \times B$
Fst $M \in A \times B \Rightarrow \operatorname{fst} M : A$
Snd $M \in A \times B \Rightarrow \operatorname{snd} M : B$

With these rules we can generate infinitely many expressions. For example, $((),\ ()) : \text{Unit} \times \text{Unit}$ is an expression, $(((),\ ()),\ ()) : (\text{Unit} \times \text{Unit}) \times \text{Unit}$ and

$((((), ()), ()), ()) : ((\mathrm{Unit} \times \mathrm{Unit}) \times \mathrm{Unit}) \times \mathrm{Unit}$ are also expressions, and so on.

Let's look at each individual rule. 15 gives us the expression $() : \mathrm{Unit}$, which is called the unit element. Its purpose is to exist as an expression of type Unit. The rule 15 gives us countably many expressions $x_i^A : A$ for every type A.

The 15 rule gives us expressions of form $\lambda x : A.\, M : B^A$, which are functions taking x as an argument with type A and $M : B$ is the definition of the function.

Example. $(\lambda x : \mathrm{Unit}.\, x) : \mathrm{Unit}^{\mathrm{Unit}}$ describes a function that takes in an argument with type Unit and returns itself, hence the type $\mathrm{Unit}^{\mathrm{Unit}}$. We draw an analogue to our previous notation of functions as $x \mapsto x$ where x is of type Unit.

The expressions of form $(M_1\ M_2) : B$ are function applications where M_1 has type B^A (thus are functions taking an argument of type A to an expression of type B), and $M_2 : A$ is the argument for the function.

Example. $((\lambda x : \mathrm{Unit}.\ x)\ ()) : \mathrm{Unit}$ applies the function $(\lambda x : \mathrm{Unit}.\ x) : \mathrm{Unit}^{\mathrm{Unit}}$ to the argument $() : \mathrm{Unit}$.

The expression $(M_1, M_2) : A \times B$ shows pairing for expressions $M_1 : A$ and $M_2 : B$. In other words, for any two expressions $M_1 : A$ and $M_2 : B$, we can create a pair $(M_1, M_2) : A \times B$. Conversely, $\mathrm{fst}\, M : A$ and $\mathrm{snd}\, M : B$ correspond to the projections of the expression $M : A \times B$.

Example. $(((), ()), ()) : (\mathrm{Unit} \times \mathrm{Unit}) \times \mathrm{Unit}$ is a pair made of $((), ()) : \mathrm{Unit} \times \mathrm{Unit}$ and $() : \mathrm{Unit}$. $\mathrm{fst}((((), ()), ()) : \mathrm{Unit} \times \mathrm{Unit}$ and $\mathrm{snd}((((), ()), ()) : \mathrm{Unit}$ are projections of the pair $(((), ()), ())$.

For the sake of extensibility, there may be other type rules. Here are a little more non-elementary examples.

Example.

$$(\lambda x : \mathrm{Unit}.\ (x, x)) : (\mathrm{Unit} \times \mathrm{Unit})^{\mathrm{Unit}}$$

$$((\lambda x : \mathrm{Unit}^{\mathrm{Unit}}.\ (x\ ()))\ (\lambda x : \mathrm{Unit}.\ x)) : \mathrm{Unit}$$

$$\mathrm{fst}((\lambda x : \mathrm{Unit}.\ (x, x))\ ((\lambda x : \mathrm{Unit}.\ x)\ ())) : \mathrm{Unit}$$

Equations

Informally, equations give us rules which show that two expressions represent the same value. To give the rules properly, we must define two terms.

Definition 16 (Free and Bound Variables)**.** For all expression x, x is a free variable. The expression with only the unit element $()$ has no free variables. For the expression $\lambda x : A.\, M$, its free variables are those of M, except x, which is bound. For other expressions, its free variables are those of its subexpressions.

Definition 17 (Substitutable)**.** Denote $\varphi(x)$ as an expression that may or may not have x as a free varaible. An expression y is substitible for x in $\varphi(x)$ if no free occurrences of y becomes bound in $\varphi(y)$.

Equations are relations of the form $M_1 =_X M_2$, where M_i are expressions and X is a set of variables such that $X \supseteq \{x : x \text{ is free in } M_1 \text{ or } M_2\}$. These relations are reflexive, symmetric, and transitive, and they must satisfy these **equation rules**.

- If $M : \mathrm{Unit}$, then $M =_X ()$
  e.g. $((\lambda x : \mathrm{Unit}^{\mathrm{Unit}}.\, (x\ ()))\ (\lambda x : \mathrm{Unit}.\, x)) =_\varnothing ()$
- If $X \subseteq Y$ and $M_1 =_X M_2$, then $M_1 =_Y M_2$.
  e.g. $() =_{\{x\}} ()$
- If $M_1 =_X M_2$, then $f\ M_1 =_X f\ M_2$
- If $M_1 =_{X \cup \{x\}} M_2$, then $\lambda x : A.\ M_1 =_X \lambda x : A.\ M_2$
- If $a : A$ and $b : B$, then $\mathrm{fst}(a, b) =_X a$ and $\mathrm{snd}(a, b) =_X b$
- If $c : A \times B$, then $(\mathrm{fst}\, c, \mathrm{snd}\, c) =_X c$
- If $f : A^B$, then $\lambda x : A.\ f\ x =_X f$ if $x \notin X$ (so x is not free in f)
  e.g. $(\lambda x : \mathrm{Unit}.\ ()) =_\varnothing ()$
- If x_2 is substitutible for x_1 in $\varphi(x_1)$ and x_2 is not free in $\varphi(x_1)$, then $\lambda x_1 : A.\ \varphi(x_1) =_X \lambda x_2 : A.\ \varphi(x_2)$
  e.g. $(\lambda x : \mathrm{Unit}.\ x) =_\varnothing (\lambda y : \mathrm{Unit}.\ y)$

Here are some non-elementary examples.

Example.

$$((\lambda x : \mathrm{Unit}^{\mathrm{Unit}}.\ (x\ ()))\ (\lambda x : \mathrm{Unit}.\ x)) =_\varnothing ()$$

$$\mathrm{fst}((\lambda x : \mathrm{Unit}.\ (x, x))\ ((\lambda x : \mathrm{Unit}.\ x)\ ())) =_\varnothing ()$$

$$(\lambda x : \mathrm{Unit}.\ (\lambda y : \mathrm{Unit}.\ y)) =_{\{y\}} y$$

Now that we have fully defined what a typed λ calculus is, we will go on to construct the equivalence between typed λ calculi and Cartesian closed categories.

Construction of the Equivalence

Now that we've laid the groundwork of category theory and programming language theory, we will build the bridge between them. When we examine the definitions of typed λ calculus and Cartesian closed categories, we find many similarities. The types in a typed λ calculus seems to have products and exponentiation, and it is clear that Unit is the analogue of the terminal object. So it is not a surprise that there exists an equivalence between them. Here are the general steps of constructing the equivalence.

1. Define the λ**Calc** and **Cart** categories
   (a) The objects of λ**Calc** are typed λ calculi and its morphisms structure preserving maps
   (b) The objects of **Cart** are Cartesian closed cat-

egories with structure preserving functors as morphisms

2. Define the equivalence between λ**Calc** and **Cart**
   (a) The functor C from λ**Calc** to **Cart** maps types to objects and equivalence classes of expressions with one free variable to morphisms
   (b) The functor L from **Cart** to λ**Calc** maps a Cartesian closed category to its *internal language*, the naturally constructed typed λ calculus
   (c) Finding that (C, L) is an equivalence by showing that there are natural isomorphisms $\eta_1 : C \circ L \Rightarrow \mathbf{1}_{\lambda\mathbf{Calc}}$ and $\eta_2 : L \circ C \Rightarrow \mathbf{1}_{\mathbf{Cart}}$

In the rest of the section, we refer to these steps as we construct this equivalence. Some of these parts will be discussed briefly and further details can be examined in the references included.

1a. The λCalc Category

It turns out that we can create a category of typed λ calculi, denoted λ**Calc**, where the objects are typed λ calculi and its morphisms structure preserving maps, called **translations**.

Definition 18 (Translation)**.** Let $\mathscr{L}$ and $\mathscr{L}'$ be typed λ calculi, and $a : A$ in $\mathscr{L}$. $\Phi : \mathscr{L} \to \mathscr{L}'$ is a translation if

- Φ sends types of $\mathscr{L}$ to types of $\mathscr{L}'$ and expressions of $\mathscr{L}$ to expressions of $\mathscr{L}'$.
- $a : A \Rightarrow \Phi(a) : \Phi(A)$.
- *Variable preserving*: Variables are preserved.
  - a has no free variables $\Rightarrow \Phi(a)$ has no free variables.
  - Φ sends every ith variable of a to the ith variable in $\Phi(a)$.
- *Type operation preserving*: Types are preserved.
  - $\Phi(\text{Unit}) = \text{Unit}$
  - $\Phi(A \times B) = \Phi(A) \times \Phi(B)$
  - $\Phi(A^B) = \Phi(A)^{\Phi(B)}$
- *Expression preserving*: Expression forming rules are preserved, for example:
  - $\Phi(\text{fst}(c)) = \text{fst}(\Phi(c))$
  - $\Phi(\lambda x : A.\ M) = \lambda\Phi(x) : \Phi(A).\ \Phi(M)$
- *Equation preserving*: $a =_X b \Rightarrow \Phi(a) =_{\Phi(X)} \Phi(b)$

Given two translations Φ and Ψ, we denote $\Phi = \Psi$ when $\Phi(a) =_{\Phi(X)} \Psi(b)$ whenever $a =_X b$.

Proposition 7. *λ**Calc** is a category.*

Proof. Given a typed λ calculus $\mathscr{L}$, the identity morphism would be the mapping from a type to itself and an expression to itself. Since translations are essentially functions, the usual composition operator suffices as the composition operator for λ**Calc**. $\square$

1b. The Cart Category

The objects of **Cart** are Cartesian closed categories with structure preserving functors as morphisms, called **Cartesian closed functors**.

Definition 19 (Cartesian closed functors)**.** A Cartesian closed functor $F : \mathbf{C} \to \mathbf{D}$ is a functor that preserves products and exponentiation. In other words, $F(1) = 1$ (recall 1 denotes a terminal object), $F(A \times B) = F(A) \times F(B)$, and $F(A^B) = F(A)^{F(B)}$.

Cart is a subcategory of the category **Cat**, and the identity morphisms and composition operators are inherited from **Cat**. The fact that **Cart** is a category follows from the definitions.

2a. From λCalc to Cart

Here we define a functor $L : \lambda\mathbf{Calc} \to \mathbf{Cart}$ that maps types to objects and equivalenct expressions with one free variable to morphisms.

In other words, given a typed λ calculus $\mathscr{L}$, the objects in the category $\text{ob}\, L(\mathscr{L})$ are the types of $\mathscr{L}$. Moreover, given two types $A, B \in T$, the morphisms between A and B are equivalence classes of expressions of type B with one free variable of type A.

We define the equivalence relation as the following. $\varphi(x)$ and $\psi(y)$ (refer to Equations for the notation) are equivalent if they are of the same type, x and y are of the same type, x is substitutable for y in $\psi(y)$, and $\varphi(x) =_{\{x\}} \psi(x)$.

The reason we need this equivalence relation is to fulfill one part of the definition for a category and another one for Cartesian closed categories. Specifically, this equivalence relation ensures the uniqueness of the identity morphisms. In other words, with the equivalence relation, any two variables x and y of the same type A correspond to the same morphism $\mathbf{1}_A$.

The equivalence relation also ensures that Unit is the terminal object by grouping together all expressions with the free variable of type Unit equivalent.

It makes sense that $C(\mathscr{L})$ is a Cartesian closed category. We are essentially converting types to objects and functions between the types to morphisms, and the Cartesian closed structures in the types of $\mathscr{L}$ are inherited over to $C(\mathscr{L})$.

We defer the details of the proof that $C(\mathscr{L})$ is indeed a Cartesian closed category to [4].

2b. From Cart to λCalc

In the previous section, we were able to construct a Cartesian closed category from a typed λ calculus by seemingly forgetting some structure of the typed λ calculus. For example, we have abstracted away the expression forming rules and the equations into the C functor's definition.

So conversely, when we are constructing a typed λ calculus from a Cartesian closed category, we must insert the rules and equations using the structure we have in Cartesian closed categories. The typed λ calculus generated from a Cartesian closed category $\mathbf{C}$ is called the *internal language* of $\mathbf{C}$.

Definition 20 (Internal languages of Cartesian closed categories)**.** The internal langauge of a Cartesian closed category $\mathbf{C}$ is $L(\mathbf{C}) = (T, \text{Expr}, E)$ such that $T = \text{ob}\,\mathbf{C}$ and the expression forming rules and equation rules are satisfied.

We are using the definition of typed λ calculus in order to satisfy our criteria for the equivalence. Barr and Wells [2] use the same definition as us, and Scott and Lambek [4] give us a more thorough definition of the internal language of a category in general, and it turns out that the definition above is equivalent in the case of Cartesian closed categories.

2c. The Natural Isomorphisms

The natural isomorphisms come naturally by essentially defining the output of the functor composition as the input. Here we give the mappings for objects and types and defer the mappings for morphisms and expressions to [4], since they require background information on *functional completeness*, a topic not covered in this paper.

To define the natural isomorphism $\eta : C \circ L \Rightarrow \mathbf{1_{Cart}}$, for each Cartesian closed category $\mathbf{C}$ we must find an isomorphism $\eta_{\mathbf{C}} : \mathbf{C} \to C \circ L(\mathbf{C})$. We notice that an object A of $C \circ L(\mathbf{C})$ is essentially a type of $L(\mathbf{C})$, which is an object of $\mathbf{C}$, so we define $\eta_{\mathbf{C}}(A) = A$ for all $A \in \text{ob}\,\mathbf{C}$. As said before, we defer the mappings for $\text{Mor}\,\mathbf{C}$ to [4].

Likewise, to define the other natural isomorphism $\varepsilon : \mathbf{1_{\lambda Calc}} \Rightarrow L \circ C$, for each typed λ calculus $\mathscr{L} = (T, \text{Expr}, E)$, we must find an isomorphism $\varepsilon_{\mathscr{L}} : L \circ C(\mathscr{L}) \to \mathscr{L}$. We notice that a type A of $L \circ C(\mathscr{L})$ is the object of $C(\mathscr{L})$, which is a type of $\mathscr{L}$, so we define $\varepsilon_{\mathscr{L}}(A) = A$ for all $A \in T$. Similarly, we defer the mappings for Expr to [4].

The proofs that the two mappings above are indeed natural isomorphisms are not covered in this paper and are given in [4].

Consequences

The equivalence of typed λ calculi and Cartesian closed categories is significant in that it gives us the isomorphisms below.

$$L(C(\mathscr{L})) \cong \mathscr{L} \text{ and } C(L(\mathbf{C})) \cong \mathbf{C}$$

This means that in general we can apply any knowledge we know about Cartesian closed categories to typed λ calculi and vice versa. For example, we can apply the entire structure of category theory and the power of commutative diagrams to the theory of typed λ calculi.

Also, since Cartesian closed categories have no variables, we will no longer have to worry about variable names clashing if we use Cartesian closed categories as a way of expressing typed λ calculi [2, 4].

Conclusion

In this paper we built a single bridge. A bridge that is unlike any other bridge because it doesn't have a physical form in this world and doesn't connect any two material objects. We built an abstract bridge that connects two large academic fields of study, mathematics and computer science. More specifically, we built an equivalence between Cartesian closed categories from category theory and typed λ calculi from programming language theory.

And we've come a long way to build this bridge. We first laid out the groundwork in category theory and Cartesian closed categories, to solidify the platform for one side of the bridge. We hardened the platform by drawing inspiration from set theory to define terminal objects from singletons, products from Cartesian products, and exponentiation from function sets.

Next, we defined the equivalence between categories as the blueprint of the bridge. We defined functors as morphisms between categories and natural isomorphisms as an equivalence between functors as components of this blueprint.

We then defined typed λ calculi, the abstract programming language, as the other platform for the bridge. The language comprised of types which were attributes, expressions which were building blocks for the program, and equations which gave equality between expressions.

Finally, we assembled the bridge by constructing the equivalence between typed λ calculi and Cartesian closed categories. We first defined the category of typed λ calculi where the morphisms were translations between two languages, then the category of Cartesian closed categories where the morphisms were Cartesian closed functors. We then constructed the equivalence between the two categories by defining functors between them and the natural isomorphisms between the two functors.

When we're asked about the applications of mathematics in computer science, in addition to Turing machines in complexity theory and statistics in machine learning, we can give the relationship between a foundational mathematical framework and programming language theory. We can say with confidence that typed functional programming languages are equivalent to Cartesian closed categories and because of that, notions from category theory can be directly applied to the languages. In theoretical fields, there are often bridges between two seemingly disconnected topics, and the equivalence of typed λ calculi and Cartesian closed categories is an example of such a beautiful bridge.

Acknowledgements

The construction of this equivalence was pioneered by Joachim Lambek, who was a professor at McGill University. His 1980 work is one of the first that focused on this idea and his paper from 1985 gives an accessible introduction [5, 6]. Most of the works here on typed λ calculi and the equivalence were adapted from the book *Introduction to Higher Order Categorical Logic* by P.J. Scott and Joachim Lambek [4].

We also thank Benjamin Pierce, professor of computer science at the University of Pennsylvania, for providing us with an accessible introduction to category theory in his book *Basic Category Theory for Computer Scientists* [11].

References

[1] A. Asperti and G. Longo. *Categories, Types, and Structures: An Introduction to Category Theory for the Working Computer Scientist.* MIT Press, Cambridge, MA, USA, 1991.

[2] M. Barr and C. Wells. *Category Theory for Computing Science.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[3] G. Huet. Cartesian closed categories and lambda-calculus. In G. Cousineau, P.-L. Curien, and B. Robinet, editors, *Combinators and Functional Programming Languages*, pages 123–135, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[4] P. J. S. J. Lambek. *Introduction to Higher-Order Categorical Logic.* Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1988.

[5] J. Lambek. From lambda calculus to cartesian closed categories. In *To H.B.Curry : essays on combinatory logic, lambda calculus, and formalism*, 1980.

[6] J. Lambek. Cartesian closed categories and typed λ-calculi. In *Proceedings of the Thirteenth Spring School of the LITP on Combinators and Functional Programming Languages*, pages 136–175, London, UK, UK, 1986. Springer-Verlag.

[7] J.-P. Marquis. Category theory. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy.* Metaphysics Research Lab, Stanford University, fall 2019 edition, 2019.

[8] A. Martini and A. Martini. Category theory and the simply-typed lambda-calculus, 1996.

[9] B. Milewski. *Category Theory for Programmers.* Blurb, Incorporated, 2018.

[10] nLab authors. internal logic. `http://ncatlab.org/nlab/show/internal%20logic`, Dec. 2019. Revision 63.

[11] B. C. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, Cambridge, MA, USA, 1991.

[12] A. Pitts. Brief notes on the category theoretic semantics of simply typed lambda calculus.

[13] E. Riehl. *Category Theory in Context.* Aurora: Dover Modern Math Originals. Dover Publications, 2017.