Abstract Nonsense and the Curry Howard Isomorphism

Jacob Denson

March 19th, 2015

In 1931, the logician Kurt Gödel announced his groundbreaking incompleteness theorem. No consistant mathematical theory with "effectively generated" axioms can prove itself consistant. Exploring Gödel's "effective generation" methods, a young logician introduced a theoretical machine to benchmark these axioms. By abstracting Gödel's ideas, Alan Turing had discovered the science of computation. Thus began an incredibly fruitful century of interaction between logic and computation. Generalization and abstraction simplify big ideas and complicated thought procedures. With the correct language, complex arguments unravel into straightforward reasoning. In this talk, I argue that category theory is an effective abstraction of fundamental computing science concepts. A theory's importance lies in its power to solve interesting, practical problems. To test category theory's linguistic power, we will untangle a particularly complicated computing science principle. Without the language of category theory, the principle cannot even really be defined precisely:

Theorem 1 (The Curry-Howard Isomorphism: Preliminary Statement)**.** *There is a deep correspondence between sound mathematical proofs and terminating computer algorithms (i.e. algorithms which never run into an 'infinite loop').*

The correspondence allows us to interpret logic as computing science, and vice. For instance, as we can think of computation as manipulating sequences of bits, a proof can also be thought as a manipulation of a prespecified language of symbols, arguing an idea. This insight is not only just intuition though; the isomorphism is a precise statement enabling us to move between the two worlds of mathematical logic and computing science. More concretely, logical arguments can be checked by converting them into computer programs. By running these programs, we can validify the correctness of a proof mechanically. This is the beautiful idea which powers modern proof verification systems!

Category theory not only allows us to state the Curry-Howard isomorphism precisely, but also shows *why* the theorem is true, which is equally important. It is particularly effective in computing science because it shifts the focus of the underlying formal foundations of thought from set theory, which involves the discussion objects, and sets that contain these objects, to a theory of transformations, and the way these transformations operate on data. When deriving

programs, we rarely focus solely on the data, but instead on the segments of computation which *operate on data*. This is why set theory emphasizes the wrong things in computing science theory. If we forget about the data, the discussion of certain problems becomes tenfold simpler.

Definition 1. *A* **Category** $\mathcal{C}$ *is a class of objects* $Obj(\mathbf{C})$*, together with a family* $Mor(A, B)$ *of morphisms (abstract transformations) between any two objects* $A, B \in Obj(\mathcal{C})$*. We write* $f : A \to B$ *to indicate* $f \in Mor(A, B)$*. The most important part of a category is a way to compose transformations. If we have two transformations*

$$f : A \to B \qquad g : B \to C$$

we must agree upon a specified transformation $g \circ f : A \to C$*, which is the composition of the two transformations. The associative law should hold,*

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

and we should have a 'do nothing' transformation $Id_A : A \to A$ *on each* A*, satisfying*

$$f \circ Id_A = f \qquad Id_B \circ g = g$$

thus the transformation operates as an algebraic identity.

Pictures are much more interesting than words; this is part of the reason category theory was invented, because we can introduce the commutative diagram to describe relations between transformations. We proceed by use of examples. The diagram

$$\begin{array}{ccc} & A & \\ {\scriptstyle g}\nearrow & & \searrow{\scriptstyle h} \\ B & \xrightarrow{\quad f \quad} & C \end{array}$$

is commutative when $h \circ g = f$. The diagram

$$\begin{array}{ccc} A & \xrightarrow{f} & B \\ \downarrow{\scriptstyle \phi} & & \downarrow{\scriptstyle g} \\ C & \xrightarrow{\psi} & D \end{array}$$

is commutative when $\psi \circ \phi = g \circ f$. More generally, a diagram, which is really a graph with objects as vertices and transformations as edges, is commutative if, whenever we have two paths

$$A_0 \xrightarrow{f_1} A_1 \xrightarrow{f_2} \ldots \xrightarrow{f_n} A_n$$

$$B_0 \xrightarrow{g_1} B_1 \xrightarrow{g_2} \ldots \xrightarrow{g_m} B_m$$

with $A_0 = B_0$ and $A_n = B_m$, then $g_m \circ \cdots \circ g_1 = f_n \circ \cdots \circ f_1$. Often one can reduce complicated arguments simply by drawing a digram, and verifying the diagram as commutative. The polarity between the two methods is initially very jarring, and seems fairly non-sequitur to the uninitiated. This is why category theory is affectionally called abstract nonsense by users of the theory.

Example. *You've almost definitely met a category before. Here are a few examples from the basic mathematics we use in computing science:*

- **Set***: The objects are sets, and transformations are functions between sets.*
- **Vect***: The objects are vector spaces, and transformations are linear maps.*
- **Graph***: The objects are graphs, and the transformations are maps between vertices which map edge-connected vertices to edge-connected vertices.*

We won't need these examples, but they provide an example of how category theory is used in other parts of mathematics.

The Curry-Howard isomorphism is most naturally stated as an 'equivalence' between two seemingly different categories, one representing the structure of logical proofs, and the other representing the structure of computer programs. The first section of this talk introduces these two categories. Then we examine the similarities between the two categories we've constructed, introducing the natural category theoretic terminology along the way. Finally, we discuss the Curry-Howard isomorphism and its consequences.

1 The Category of Logical Statements

In essense, logic discusses two objects: logical statements, and arguments between these logical statements. As such, a category providing a language to discuss logic must incorporate both as its objects and morphisms. We begin by reminding ourselves of the basic syntax of logic.

Definition 2. *The logical formulas of propositional logic are constructed from a set* $\mathbf{V}$ *of variables, together with a designated constant* $\top$*. The construction is recursive, so that*

1. *All variables are formulae.*
2. $\top$ *is a formula.*
3. *If* Γ *and* Σ *are formulae, then* $(\Gamma \wedge \Sigma)$ *and* $(\Gamma \Rightarrow \Sigma)$ *are formulae.*

Think of the variables as Lego pieces, and the formulas as Lego piece constructions – we snap together the variables using the construction rules to make all possible formulas.

We will assume a fixed set of variables over the logic we discuss in the future. Though parenthesis are important syntactically, we will often eschew them in favour of readability. As used in the definition above, we will by convention let capital greek letters like Γ, Σ, and Δ stand for arbitrary formulae. We interpret $\top$ as a 'tautological' statement, which is always true, the $\wedge$ operation as representing the AND logical operation, and $\Rightarrow$ as representing the IF-THEN operation, which describes a statement asserting that Σ is true if Γ is true. If

you've seen the truth-functional completeness of families of Boolean operators before, you might notice that we *cannot* represent the NOT operation, by combining AND and IF-THEN operations. This represents the fact that the logic we are considering here is 'intuitionistic', or 'constructive'. Indeed, we might be able to negate a logical statement, but it is not clear what it means to 'negate' a program. One can develop a category theory of logic including negation, and one can establish correspondences of this category with various interpretions of negation in terms of operators on programs such as 'call-with-current continuation' and 'continuation-passing-style translation', which have computational interest, but we will not discuss this interpretation here, moving on to discussing what a 'transformation' of one statement into another statement is.

Definition 3. *The axioms of a logical system are constructed from some set* $\mathbf{A}$*, where each element is of the form* $\Gamma \vdash \Delta$ *(If* Γ *is true, then* Δ *is true). We assume that each axiom system contains three basic axiom schemas*

1. *(Terminality)* $0 : \Sigma \vdash \top$*, for all formulas* Σ*.*
2. *(Identity)* $Id : \Sigma \vdash \Sigma$*, for all formulas* Σ*.*
3. *(Projection)* $\pi_1 : \Sigma \wedge \Omega \vdash \Sigma$ *and* $\pi_2 : \Sigma \wedge \Omega \vdash \Omega$ *for all* Σ, Ω*.*

The set of proofs are formed from axioms under composition laws.

1. *All axioms are proofs.*
2. *If* $P : \Gamma \vdash \Sigma$ *and* $Q : \Sigma \vdash \Delta$ *are proofs, then we can form the 'composition proof'* $Q \circ P : \Gamma \vdash \Delta$*, which is a proof of* Δ *from* Γ*.*
3. *If* $P : \Gamma \vdash \Sigma$*, and* $Q : \Gamma \vdash \Delta$ *are proofs, then we can form the 'product proof'* $\langle P, Q \rangle : \Gamma \vdash \Sigma \wedge \Delta$*.*
4. *If* $P : \Gamma \wedge \Delta \vdash \Sigma$ *is a proof, then we can form the 'abstraction proof'* $\lambda P : \Gamma \vdash (\Delta \Rightarrow \Sigma)$*, which represents a proof by assumption.*

The reason for adding additional axioms to the theory is to consider more complciated logics. If we want a formula Σ *to hold by assumption, then we add the axiom* $\top \vdash \Sigma$ *to our system. Furthermore, the axioms allow us to consider more powerful logical systems. For instance, Pierce's law* $(\Gamma \Rightarrow \Delta) \Rightarrow \Gamma \vdash \Gamma$ *could be added, an axiom which is equivalent in power to the principle of proof of contradiction.*

The notation $P : \Sigma \vdash \Delta$ should remind you of the notation for a morphism $f : A \to B$. This is purposeful: we define the logic category $\mathbf{Prf}(\mathbf{V}, \mathbf{A})$, to be the category whose objects are statements, and whose morphisms are proofs from one statement to another.

Remark. *The only problem with our construction of* $\mathbf{Prf}(\mathbf{V}, \mathbf{A})$ *is that our proofs have no concrete representation, and we therefore have no natural way of considering when two proofs are equal to one another. If one is only interested*

in the semantics of proof theory (i.e. one only cares if a statement is provable from another, rather than the various types of proofs one can construct), then one can identify any two proofs with the same assumption and deduction. For the purposes of proof theory, we can take some other syntactic approach. Hilbert defines proofs as various 'justified' sequences $(\Phi_1, \ldots, \Phi_n)$ *with* $\Phi_1 = \Gamma$ *and* $\Phi_n = \Delta$*. If we have a proof of* Σ *from* Δ *of the form* $(\Psi_1, \ldots, \Psi_m)$*, we can form the composition* $(\Phi_1, \ldots, \Phi_n, \Psi_2, \ldots, \Psi_m)$*. Now we have a syntactic representation, and we can set the two proofs as equal. Of course, in computing science we have the same problem – two functions may be described in code in two different ways, but give the same output for every input, and we can either semantically identify them, or syntactically distinguish them. For simplicity in this talk, we will take the semantic approach, and identify proofs that are equivalent to one another.*

2 The Category of Data Types

The other category we wish to discuss is firmly rooted in the basic theory of computing science. The object will consist of data types, and the morphisms functions between these data types. The λ-calculus provides us with the highest formal way of representing the functions found in computation, concerning itself solely with the way that data is transformed by manipulating strings of characters. The calculus is so purely about transformations that there are no objects in the calculus for the functions to operate on – functions only operate on other functions.

Definition 4. *The terms of the* λ*-calculus are constructed inductively from a set* $\mathbf{V}$ *or variables and constants* $\mathbf{C}$*.*

1. *All variables and constants are terms.*
2. *If* f *and* x *are terms, then so is* (fx)*.*
3. *If* x *is a variable, and* f *is a term, then* $(\lambda x.f)$ *is a term.*
4. *If* x *and* y *are terms, then so is their tuple* (x, y)*.*
5. *If* x *is a term, then so too are the projection maps* $(\pi_1 x)$ *and* $(\pi_2 y)$*.*

This is analogous to the construction of logical formulas of propositional logic.

In it's general form, the Lambda calculus has only a single type of object, the general term f, which represents some computable function in the real world. The application of some term f to some other term x is denoted fx, and represents 'the output of f given input x'. But this implies that the functions in the Lambda calculus have no restriction on their domain – they can take arbitrary terms as input. To better model practical programs, we need to introduce a type system to the Lambda calculus, so that only particular data types can be applied to functions.

Definition 5. *The types of a Lambda calculus are generated by a base set* $\mathbf{T}$ *of types, assumed to contain a 'universal' type, denoted* $*$.

1. *All base types in* $\mathbf{T}$ *are types.*
2. *If* A *and* B *are types, then so are* $(A \times B)$ *(the tuple data type) and* $(A \to B)$ *(the function data type).*

We interpret $A \times B$ as the 'tuple type' consisting of a tuple containing one element of type A, and the other of type B. $A \to B$ is the type consisting of the class of functions which take elements of type A, and return elements of type B. In the C programming language, we can model types by

$$\mathbf{T} = \{\text{Int}, \text{String}, \text{Float}, \text{Bool}, \dots\}$$

whereas in Python, $\mathbf{T} = \{*\}$, since the language is untyped (sort of).

Definition 6. *A typing context* Γ *is a rule which associates some finite subset of the variables* x *a certain type* A*. We write* $\Gamma \vdash t : A$ *to say that the term* t *has type* A *in context* Γ*. If* Γ *associated variables* $x_1, \dots, x_n$ *with types* $A_1, \dots, A_n$*, we will let* Γ *be denoted* $x_1 : A_1, \dots, x_n : A_n$*. Thus* $\Gamma, x : A$ *is the typing context which expands* Γ *to give* x *the type* A*, in addition to specifying the types that* Γ *initially specified. We shall assume that the constants of a particular* λ *calculus already have preassigned types irrespective of constants, and that* $*$ *always has type 1. We inductively define the typing context of certain terms* t *by the following rules – note that not all terms need have a well defined type by a particular context:*

1. *If* t *is a variable, and* Γ *associates* t *with* A*, then* $\Gamma \vdash t : A$*.*
2. *If* $\Gamma \vdash t : A \to B$ *and* $\Gamma \vdash u : A$*, then* $\Gamma \vdash (tu) : B$*.*
3. *If* $\Gamma, x : A \vdash t : B$*, then* $\Gamma \vdash (\lambda x.t) : A \to B$*.*
4. *If* $\Gamma \vdash t : A$ *and* $\Gamma \vdash u : B$*, then* $\Gamma \vdash (t, u) : A \times B$*.*
5. *If* $\Gamma \vdash t : A \times B$*, then* $\Gamma \vdash (\pi_1 t) : A$ *and* $\Gamma \vdash (\pi_2 t) : B$*.*

Some terms will not have an associated type based on this definition. We say a term is **well-typed** *if it has an associated type with a context.*

The main role of the Lambda calculus is to define computation in precise terms. To be brutally simple, all a computer does is manipulate binary symbols underneath the hood. The Lambda calculus formalizes this process as a process of manipulating symbols on a page. The terms defined above represent functions transformation data from one way to another, and the way these functions transform data will be defined in terms of symbol manipulation, or equivalence. Don't be discouraged if you are overwhelmed by the definition, as in the sequel I will include 'psuedomathematics' together with the mathematics to make the ideas more intuitive to a computing scientist.

Definition 7. *Given a context Γ, we can consider an equivalence between well-typed terms based on some set of axioms $\mathbf{A}$ of the form $\Delta \vdash t \equiv u$, where Δ is a particular type context and t, u are well-typed terms, assumed to both be of the same type when interpreted by Δ. We assume the following are axioms of any particular λ-system (where well-typedness is assumed):*

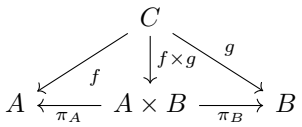
1. *(Identity)* $\Gamma \vdash t \equiv t$.
2. *(Symmetry) If* $\Gamma \vdash t \equiv u$, *then* $\Gamma \vdash u \equiv t$.
3. *(Transitivity) If* $\Gamma \vdash t \equiv u$, *and* $\Gamma \vdash u \equiv v$, *then* $\Gamma \vdash t \equiv v$.
4. *(Redundancy) If* $\Gamma \vdash t \equiv u$, *then* $\Gamma, x : A \vdash t \equiv u$.
5. *(Equivalence of Universal Constants)* $\Gamma \vdash t : 1$, *then* $\Gamma \vdash t \equiv *$.
6. *(Tuple Equivalence) If* $\Gamma \vdash u \equiv v$ *and* $\Gamma \vdash t \equiv s$, *then* $\Gamma \vdash (u, t) \equiv (v, s)$.
7. *(Projection Equivalence) If* $\Gamma \vdash s \equiv t$, *then* $\Gamma \vdash (\pi_1 s) \equiv (\pi_1 t)$ *and* $\Gamma \vdash (\pi_2 s) \equiv (\pi_2 t)$, *where well-typed.*
8. *(Formation Equivalence)* $\Gamma \vdash t \equiv ((\pi_1 t), (\pi_2 t))$.
9. *(Coordinate Equivalence)* $\Gamma \vdash (\pi_1(u, v)) \equiv u$, $\Gamma \vdash (\pi_2(u, v)) \equiv v$.
10. *(Functional equivalence) If* $\Gamma \vdash s \equiv t$ *and* $\Gamma \vdash u \equiv v$, *then* $\Gamma \vdash su \equiv tv$.
11. *(β rule)* $\Gamma \vdash (\lambda x.t)u \equiv t[x : u]$.
12. *(ξ rule) If* $\Gamma, x : A \vdash t \equiv u$, *then* $\Gamma \vdash \lambda x.t \equiv \lambda x.u$.
13. *(η rule)* $\Gamma \vdash (\lambda x.t) \equiv t$ *if* x *is not a free variable in* t.

Together with terms, constants types, and equivalence rules, we obtain a particular instance of the typed λ-calculus. To form a category, we let the types of the calculus form the objects of the calculus, and the morphisms the single-variable functions which take one type, and spit out another type. To be precise, a morphism between two types A and B is a term t and a variable x such that $x : A \vdash t : B$, where x is a free variable in t, defined modulo the equivalences defined above. If we have a term u such that $y : B \vdash u : C$, then $x : A \vdash (\lambda y.u)t : C$ is a morphism between A and C, and so we have a composition operation on terms. The morphism $x : A \vdash x : A$ acts as an identity in this respect, because $(\lambda x.x)y$ always reduces to y where defined, and $(\lambda y.t)x$ always reduces to t. We denote the category of types $\mathbf{T}$ as objects and morphisms as equivalence classes of types by $\mathbf{Comp}(\mathbf{T}, \mathbf{A})$.

3 General Abstract Nonsense

We now have defined two majorly important categories for our talk, $\mathbf{Prf}(\mathbf{V}, \mathbf{A})$ and $\mathbf{Comp}(\mathbf{T}, \mathbf{A})$. The Curry-Howard isomorphism shows these mathematical objects are 'categorically equivalent', in a way we will now see. The advantage of category theory is that we can now develop the abstract 'theory of transformations', independent of the data they manipulate, and in this language we will see the correspondence between the two 'transformations' in the category of proofs and the category of functions in computing science. We will discuss three basic properties which emerge in the elementary discussion of category theory. Here is a first example of how we can define an data independent operation on transformations.

Definition 8. *Let A and B be objects of some general category $\mathcal{C}$. A product for A and B is a third object, denoted $A \times B$, with two projection morphisms $\pi_A : A \times B \to A$ and $\pi_B : A \times B \to B$, such that, if C is an object with two morphisms $f : C \to A$ and $g : C \to B$, then there exists a unique morphism $(f \times g) : C \to A \times B$ such that $\pi_A \circ (f \times g) = f$, and $\pi_B \circ (f \times g) = g$. We can express this property with the commutative diagram*



In terms of transformations, $A \times B$ is the most specific object which merges data in A and data in B into a single object. For example, the categorical product of two sets is the cartesian product: If $x \mapsto f(x)$ and $x \mapsto g(x)$ are two set theoretic functions, then $x \mapsto (f(x), g(x))$ is the product map. The projection maps are just $(x, y) \mapsto x$ and $(x, y) \mapsto y$. A natural question is whether we can form products of objects in logic and programming:

- In terms of logic, a product for two statements A and B would be the most specific logical statement containing all the data implied by A and B. Intuitively, this statement is $A \wedge B$. The projection maps are just the proofs $\pi_1 : A \wedge B \vdash A$ and $\pi_2 : A \wedge B \vdash B$. If C is a logical statement with two proofs $P : C \vdash A$ and $Q : C \vdash B$, then we can form the product proof $\langle P, Q \rangle : C \vdash A \wedge B$, and then $\pi_1 \circ \langle P, Q \rangle = P$ and $\pi_2 \circ \langle P, Q \rangle = Q$.
- In terms of programming a product for two types A and B is almost given away by the notation for the product data type $A \times B$. The projection morphism is just the projection programs π_1 and π_2. The equivalence rules shows mathematically that these morphisms give a product structure, if we interpret the product of two terms f and g as (f, g). In psuedocode, π_1 and π_2, and the product $f \times g$ can be written as

```
def pi_1(x): return x[0]     def pi_2(x): return x[1]
```

```
def prod(x,y): return (f(x), f(y))
```

Definition 9. *A terminal object A in a category $\mathcal{C}$ is an object such that there is a unique morphism from B to A for any object B.*

In a sense, a terminal object has the least information, or structure, than any object in a category. The logical statement which contains the least logical information is the tautological statement $\top$, since one can prove $\top$ from any other statement. In terms of programs, the terminal type is the 'void type' $*$, since for any other type A, $x : a \vdash * : 1$. In terms of pseudocode, this program is the one which returns nothing:

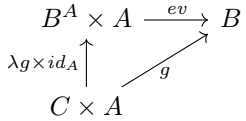
```
def terminal(x): return void
```

The void type definitely contains the least structure of any other data type.

Example. *There is an important problem in computing science which I'm sure you've at least heard of. Consider the category* **NP***, whose objects consist of NP decision problems, and whose morphisms are (equivalence classes) of polynomial time computable maps from inputs to the NP decision problem to inputs to the other decision problem which preserve the decisions the problems make on inputs. The terminal objects are, in a sense, 'universal problems' which we can reduce any other problem to, and in this context we call these decision problems* **NP complete***. The Cook-Levin theorem guarantees the existence of an* **NP** *complete object. The* $\mathbf{NP} = \mathbf{P}$ *conjecture asks whether every object in the category of NP decision problems is terminal.*

The third construction we will discuss in categories is a 'self referential' structure in the category. Self reference is key to many logical and computational paradoxes, so it makes sense that this structure is key to identifying these two classes.

Definition 10. *An exponent for two objects A and B in a category is an object B^A, which we can view as the 'object of morphisms' from A to B, together with an 'evaluation' morphism $ev : B^A \times A \to B$, representing the application of some morphism in B^A to some data in A. This evaluation morphism has the property that if $g : C \times A \to B$ is any morphism, which we can view as a family of functions from A to B indexed by data in C, then there is a unique morphism $\lambda g : C \to B^A$ making the following diagram commute:*



If $A \vdash B$ is a proof of A to B, then the natural object which should represent proofs from A to B is the implication B^A. The evaluation morphism is the modus ponens proof $(A \to B) \wedge A \vdash B$, which 'evaluates' the implication with

the first term to obtain the second term. If $C \wedge A \vdash B$, then the proof $C \vdash A \Rightarrow B$ obtained from asbtraction operates as the evaluation. In terms of programs, given two types A and B, the natural way to represent functions from A to B is the *type* $A \to B$ of functions from A to B. The evaluation morphism is then

$$x : A \to B \times A \vdash (\lambda x.\pi_1(x)\pi_2(x)) : B$$

which in psuedocode, is just

```
def eval(f,x): return f(x)
```

Thus both categorys have exponentials.

The only definition that remains to discuss the Curry-Howard correspondence is the definition of an 'equivalence' between categories.

Definition 11. *Let $\mathcal{C}$ and $\mathcal{D}$ be categories. A* **functor** *F is a map associating objects A in $\mathcal{C}$ to objects B in $\mathcal{D}$, and associating morphisms $f : A \to B$ to maps $F(f) : F(A) \to F(B)$. This map should respect the rules of the original category, in the sense that $F(f \circ g) = F(f) \circ F(g)$.*

A functor allows us to show that categories correspond to one another. We can view a functor as a 'mathematically rigorous metaphor' to study one category by looking at objects in the other. Almost all the correspondences in mathematics can be viewed as functors.

Example. *Let $\mathcal{C}$ be the category whose objects are points in some Euclidean space $\mathbf{R}^n$, and whose morphisms are maps from points $p \in \mathbf{R}^n$ to $q \in \mathbf{R}^m$ are maps $f : \mathbf{R}^n \to \mathbf{R}^m$ with $f(p) = q$, differentiable at p. If $\mathcal{D}$ is the category whose objects are points in some Euclidean space, and whose morphisms f from p to q are affine maps mapping p to q, then differentiation can be viewed as a functor from $\mathcal{C}$ to $\mathcal{D}$, which fixes all objects, and maps a function f from p to q to the affine map $x \mapsto q + Df(p)x$.*

Here is a more interesting example of a functor in computing science.

Example. *Category theory appears almost everywhere we want to draw diagrams of arrows between objects. It's therefore unsurprising that category theory appears in software design. We can view a UML diagram as some abstract category, which describes what a series of functions and classes should satisfy. When we program a particular instance of the diagram, we are really constructing a functor from the category of objects in the UML diagram to the category of functions, preserving the properties established by the diagram.*

4 Closed Cartesian Curry Howard Isomorphisms

The constructions of products, exponentials, and terminals are of vital importance to the correspondence between logic and programming, and as such we have an important sounding name for categories with these operations. The name comes from the fact that such a category is closed under (cartesian) products, and has exponents.

Definition 12. *A category $\mathcal{C}$ with a terminal object, such that any two objects has a product and exponentials is called a* **Cartesian closed category**.

The properties of being a Cartesian closed category are fairly strong. Indeed, we shall find that essentially, there is only a single example of a Cartesian closed category, hence the fact that logic and programming are categorically equivalent. We have cherry picked categories with these properties, and unfortunately, we do not have time to discuss categories which are not Cartesian closed.

The precise description of the Curry-Howard isomorphism will be described with respect to a functor between categories. We rely on the following result, which descends into a rather nitpicky argument which left to the reader (the definitions of this talk are more important than the actual proof).

Theorem 2 (Curry-Howard-Lambek-Lawvere-Scott Isomorphism)**.** *If $\mathcal{C}$ is Cartesian closed, there is a natural Lambda calculus $\mathbf{L}(\mathcal{C})$ equivalent to $\mathcal{C}$. In particular, $\mathbf{Prf}(\mathbf{A})$ is isomorphic to some lambda calculus.*

We now discuss why this thoery can be used to prove mathematical statement by constructing programs on a computer.to prove a statement, it suffices to check proofs with a computer.

Definition 13. *A* **model** *of a lambda theory $\mathbf{Comp}(\mathbf{T}, \mathbf{A})$ is a Cartesian closed category $\mathcal{C}$ together with a functor $F : \mathbf{Comp}(\mathbf{T}, \mathbf{A}) \to \mathcal{C}$ which preserves the properties of a Cartesian closed category; this means precisely that*

$$F(A \times B) = F(A) \times F(B) \qquad F(B^A) = F(B)^{F(A)}$$

In addition, we say our model satisfies $\Gamma \vdash t \equiv u$ holds in our λ theory, we must also have

$$F(\Gamma \vdash t : A) = F(\Gamma \vdash u : A)$$

What we mean by this is that if we replace $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ with a variable $x : A_1 \times \cdots \times A_n$, then the equation holds.

In mathematical logic, one studies both the syntax and semantics of a logical system. The syntax shows which theorems can be formally proved, and the semantics shows which mathematical systems the logical systems model. These systems are known as **models** of the theory. As an analogy, the theory of groups is just an abstract set of operations, and the semantic theory of groups studies *particular* mathematical groups satisfying the axioms of a group. A completeness theorem for a logical system is a statement implying the fact that, if a property holds for all models of a logical system, we can formally prove that property. We have shown that if we interpret cartesian closed categories as models of a particular logic, then the logical rules completely describe properties of the cartesian categories. The important fact here is that we have a completeness theorem for cartesian closed categories.

Theorem 3. *In any λ theory, one can prove $\Gamma \vdash t \equiv u$ if and only if every model satisfies $\Gamma \vdash t \equiv u$.*

It follows from the Curry-Howard isomorphism that the minimal logic category (the category obtained from the smallest axiom set), must be isomorphic to the most basic lambda calculus. If we want to prove a statement in this logic category, it suffices to construct a term of the λ calculus modelling the implication i.e. to construct a program modelling the proof. Thus, if we want to prove a theorem in this logic category, we need only construct a well-typed statement in the λ calculus. There are programming languages which model the lambda calculus , and thus we are able to check proofs with a computer.

5 Computational Category Theory

The point of this talk was that, if we build up the ideas of category theory, the Curry-Howard isomorphism theorem becomes obvious. The technique of identifying logical properties with categorical descriptions generalizes to many more correspondences between more advanced logics and additional features in programming languages. If we have a new feature of computation, we can likely find a logical axiom modelling this feature, and then study the correspondence. Here are some examples:

Conjunctive Logic	Cartesian Categories (not necessarily closed)
Positive Logic Intuitionistic Propositional Logic	Cartesian Closed Categories Bicartesian Closed Categories
Classical Propositional Logic	Bicartesian Closed Categories with Elimination
Linear Logic	$*$-Autonomous Logic
First-Order Logic	Hyperdoctrines
Martin-Löf Type Theory	Locally Cartesian Closed Categories

Studying these categories has led to an entirely new field of computing science: The theory of homotopy type theory. The theory relates the topological concept of homotopy (a continuous bend or stretch of space) to the study of types. Homotopy type theory enables us to more easily translate mathematical arguments into formal proofs, and thus makes formal provability of mathematical statements a viable option in the future.

Type theory is not the only part of computing science where category theory excels in pedagogy, though it is the only place in computing science we are able to talk about here. If you've ever programmed in Haskell, you may have noticed many terms in this talk occur in the lingo of Haskell programmers. This is because Haskell models complicated functional programming concepts in the language of category theory (in addition Haskell uses the concepts of Functors, Monads, Monoids, and Natural Transformations, which we do not discuss in this talk), simplifying explanations of complicated topics when explained in the universal language of category theory. There is even a formal category **Hask** online which models Haskell, and is studied academically. Unfortunately, most people learning Haskell have not had experience with category theory before

learning the language, which makes the jump to the category theory terminology very unpleasant.

The main online source for information on category, especially pertaining to computing science, is the N-lab, and if this talk has enticed you to become interested in the interactions between the trifecta of logic, computing science, and category theory, I strongly recommend you check the website out.

References

[1] Steve Awodey, *Lectures on Category Theory for Computing Scientists*

[2] Steve Awodey, *Introduction to Categorical Logic*

[3] Saunders Maclane, *Categories for the Working Mathematician*

[4] Carsen Berger, *A Categorical Approach to Proofs As Problems*

[5] Jesse Alama, *The Lambda Calculus (Stanford Dictionary of Philosophy)*

[6] The N-lab, *Lambda Calculus*