Abstract Nonsense and the Curry-Howard correspondence

Jacob Denson

March 19, 2015

1 Introduction

"A single simple principle can masquerade itself as several difficult results"
— Michael Spivak

The prime importance of generalization is its ability to simplify complicated problems. Big ideas often shroud the key pieces of information which will signal to us why some logical conclusion can be granted. Given the correct extraction, a complex argument will unravel into a straightforward calculation. Throughout Computing Science and Mathematics, abstractions have popularized some method, allowing others to understand and apply this method in its generalized form to other problems.

In 1931, a man named Kurt Gödel proved the following result: no consistant mathematical theory whose theorems are 'effectively generated' can prove itself consistant. Seeking to extract the method which Gödel used to prove his result, and expand the meaning of 'effective generation', a man named Alan Turing discovered the study of computability through his formal method of Turing machines. His generalization has probably had quite a bit of impact on our lives.

Three hundred year before this, in the year 1668 the mathematician James Gregory developed a fundamental method which relates the slopes of tangent lines to the areas under the curves those tangent lines relate. What we now know as Calculus was invented by Sir Isaac Newton to generalize this result. Perhaps the reason why we do not place James in the place of Isaac Newton and Gottfried Leibnitz is that his theory was not generalized enough for our tastes at the time. Generalization permits understanding, which is the point of an undertaking of a theoretical study in the first place.

A correct generalization of an idea will naturally lead to insight; the thesis of this talk is that category theory is a correct generalization of many concepts in Computing science. Our goal today is to understand the basics of category theory and its nature in computing science. A generalizing theory's importance lies only in its applications to specific ideas, so today, we will focus on one specific application in computing science. Focussing on one specific idea, and explain it through the language category theory provides will provide ample evidence of category theory's intution and use. With category theory, the Curry-Howard isomorphism will be established, a fairly unintuitive idea related to the kinds of transformations that category theory explains so well. Category theory will not only give us the Curry-Howard isomorphism, but show WHY it is true as well (which is equally important to the use of the isomorphism). We can't even state precisely what the Curry Howard isomorphism is without the tools category theory will provide. For now, I offer a preliminary statement:

Theorem 1 (The Curry Howard Isomorphism)**.** *Proofs kinda equal Programs, if you squint your eyes a bit and turn your head sideways.*

Less facetiously, there is a natural correspondence of proofs and programs in the Lambda calculus. This gives us intuition behind many corresponding ideas between the two fields; if you are proving something in mathematics, thinking about it algorithmically may help you to understand the problem. Vice versa, proving something may be thought of as computing an algorithm to solve a problem. This is not just a vague insight though, because there is a deep connection between the two fields. On the other side, we can think of a program as proving something, carrying truth from input to output. It is a beautiful idea. More concretely, it allows us to check logical arguments by converting them into their natural equivalents in the Lambda calculus. In turn, the Lambda calculus can be interpreted by an actual programming language, and thus, provided they can be computed (well-typed program), then we will have shown the theorem to be correct. This is the key idea behind modern proof checking programs.

The reason we can apply category theory to computing science is that it provides a different emphasis than the standard foundations for mathematics. Before category theory, the sole language of mathematics was developed through the language of sets. This system aimed to described the collections of objects that are studied by mathematicians. The problem of using set theory in computing science is that the emphasis is in the wrong place. When we program, or design algorithms, our focus is not on the data, but on the method of combining data with data transformations to form an output from a designated input. This is what category theory describes best: structure preserving transformations on sets of objects – its the arrows that count! This explains why category theory helps to explain much of computing science. Because of the generalization, there will be quite a few definitions in this presentations that do not seem to lead anyway – you will have to trust me that by the end of the presentation these ideas will unite into something that permits a common understanding of our discussion.

We should probably define what a category is before the reader becomes too lost in our falandering about. The way to think of it from our side of the picture is as a bunch of data transformations that can be described by 'arrow diagrams'.

Definition 1. *A* **Category** $\mathcal{C}$ *consists of two components, the first are the objects of the category, segments of data which may be operated on, denoted* $Obj(\mathcal{C})$. *The second are transformations between pieces of data. If* A *and* B *are two objects in* $\mathcal{C}$ $(A, B \in Obj(\mathcal{C}))$ *then we have a collection of transformations* $Mor(A, B)$ *between them, called morphisms. We write* $f : A \to B$ *for such a transformation. If I transform parts of* A *into* B*, and parts of* B *into* C*, then surely I can transform parts of* A *into* C*: If* $f : A \to B$ *and* $g : B \to C$ *are morphisms, then we should be able to compose them to form a new function which takes the data in* A *and gives us data in* C*. We write* $g \circ f : A \to C$ *for the new transformation formed by composing these two morphisms. We should have that*

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

Which holds defacto for any reasonable transformation anyway. Also, we should have a 'do nothing' transformation for each object A*, which we denoted* $\mathbf{1}_A$*, satisfying, for any* g *and* h *for which it makes sense,*

$$g \circ \mathbf{1}_A = g$$

$$\mathbf{1}_A \circ h = h$$

Any object satisfying the properties above is a category.

Looking at pictures is much more interesting than reading definitions (its part of the reason category theory was thought up in the first place); thus we introduce the commutative diagram.

$$\begin{tikzcd} & A \arrow[dr, "h"] & \\ B \arrow[ur, "g"] \arrow[rr, "f"] & & C \end{tikzcd}$$

This diagram is commutative if $h \circ g = f$.

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

is commutative when $g \circ f = h \circ k$. Summarizing,

Definition 2. *A Diagram is commutative if, whenever we can go from one object to another by two different sequences of arrows, then the composition of the two sequences is equal. If*

$$A_1 \to A_2 \to \cdots \to A_n$$

$$A_1 = B_1 \to B_2 \to \cdots \to B_m = A_n$$

Then $f_n \circ \cdots \circ f_1 = g_m \circ \cdots \circ g_1$. *Often one can justify a complicated statement simply by showing that a diagram is commutative. These arguments, which seem fairly nonsequitur to the uninitiated reader, are the reason why category theory is, affectionately, known as general abstract nonsense.*

The rules for associativity and identity for a general category can be summed up by two diagrams below.

$$\begin{tikzcd}
& A \arrow[loop left, "\mathrm{id}_A"] \arrow[dr, "g"] & \\
B \arrow[ur, "f"] & & C
\end{tikzcd}$$

$$\begin{tikzcd}
A \arrow[r, "(g\circ f)"] \arrow[d, "f"] & C \arrow[d, "h"] \\
B \arrow[ur, "g"] \arrow[r, "(h\circ g)"] & D
\end{tikzcd}$$

You've probably met some at least one category before, but he may not have been polite enough to introduce himself as such. Try to identify the identity and composition mappings that we defined above.

- **Set**: Objects are all sets, Morphisms are set theoretic maps. Transformations preserve 'Settiness', and are just functions between two sets.
- **Vect**: Objects are vector spaces, Morphisms preserve 'Linearity', and we know them as Linear Transformations.
- **Graph**: Objects are graphs, morphisms are graph morphisms (a Graphine transformation?), which preserve the properties of graphs. It is a fundamental computational problem to identify whether there exists certain bijective morphisms between graphs, and it is unknown whether this problem can be computed in polynomial time or if it is NP-complete (or both, if $P = NP$).

These categories are just illustrations to enable you to understand what we mean when we say that morphisms 'preserve structure', which means little to an abstract category but almost everything in some specific category. To talk about the Curry-Howard isomorphism in categorical terms, we must describe the structures which establish the isomorphism. First, we want a logic category, developed from some deductive system (method of arguing things). The objects should be statements, and the arrows arguments showing how one statement implies the other.

Definition 3. *The objects of this logic category are defined by a (at least countable) set of variables* $\mathbf{V}$ *together with a designated variable* $\top$*, and two connectives* $\wedge$ *and* $\vee$*. Our formulas are defined as follows:*

1. *A variable is a formula.*
2. *If* Γ *and* Σ *are formulas, then so are* $\Gamma \wedge \Sigma$ *and* $\Gamma \vee \Sigma$*.*

In any logic we give, we will assume that the set of variables is the same, and that it contains all symbols we use in the talk (and countably more).

It is this 'positive' subset of logic that corresponds more naturally to computation. This segment of logic is designed to remove implication from logic – after all, what does it mean to 'negate' a program? One can develop these other logics in a category theoretic manner, and some of these relate back to computing science, but this adds unneeded complexity for the moment. For now, lets discuss the morphisms in the category we have defined.

Definition 4. *The axioms for the logical system above consist of some set* $\mathbf{A}$*, where each element is of the form* $\Gamma_1, \ldots, \Gamma_n \vdash \Delta$*, where* Γ_i *and* Δ *are formulas in the logic. We read this as saying provided that all* Γ_i *are satisfied (true), then* Δ *is implied. We assume every axiom system* $\mathbf{A}$ *contains five base axiom schemas that are so self evident to avoid being true:*

1. *(Terminality)* $\Sigma \vdash \top$*, for all formulas* Σ
2. *(Projection)* $\Sigma \wedge \Omega \Rightarrow \Sigma$ *and* $\Sigma \wedge \Omega \Rightarrow \Omega$*, for all formulas* Σ *and* Ω*.*
3. *(Production)* $\Gamma, \Sigma \vdash \Gamma \wedge \Sigma$*.*
4. *If we have an axiom* $\Gamma_1, \ldots, \Gamma_n \vdash \Delta$*, then for each* Γ_k *we have an axiom* $\Gamma_1, \ldots, \Gamma_{k-1}, \Gamma_{k+1}, \ldots, \Gamma_n \vdash (\Gamma_k \Rightarrow \Delta)$*.*
5. *(Modus Ponens)* $\Gamma \Rightarrow \Delta, \Gamma \vdash \Delta$

Note that we **do not** *necessarily have the axiom Modus Tollens, a proof by contradiction, in our system. Logicians introduced the variable* $\top$ *because it makes explicit when a statement* A *is a tautology – exactly when* $\top \vdash A$*.*

Axioms are the rules for the games we play when we perform deduction on a logical system, so that we know what is right and wrong when making a deduction. Now we must define the game.

Definition 5. *Let* Γ *and* Δ *be formulae. A* **proof** *from* Γ *to* Δ *is a finite sequence of formulae,* $(\Phi_1, \Phi_2, \ldots, \Phi_n)$ *where* $\Phi_1 = \Gamma$*, and* $\Phi_n = \Delta$*, and such that each* Φ_k *is justified by a logical principle. That is, for each* Φ_k *there exists an axiom* $\Psi_1, \ldots, \Psi_n \vdash \Phi_k$*, and each* Ψ_i *is either* Γ *or some* Φ_i*, for* $i < k$*. If* P *is a proof from* Γ *to* Δ*, we write* $P : \Gamma \vdash \Delta$*, rather than* $P : \Gamma \to \Delta$*, to denote the morphism.*

What use are adding more axioms to our system? Some logicians have decided that they do not like the standard manner of reasoning, and correspondingly, they weaken or strengthen the system at hand. For instance, we could add the axiom $(\Gamma \subset \Delta) \subset \Gamma \vdash \Gamma$ to our system. This is in fact equivalent to proof by contradiction without every introducing negation. Try proving this statement in your naive notion of logic; you will notice that you will entually have to use a proof by contradiction, implicitly or explicitly. We can also add $\top \vdash \Gamma$, which adds Γ as an assumption to our system. Now lets categorize this logic. We want to formulate a method of composition of proofs. When we want to compose two proofs, we take the bottom of the proof, and connect it to the top of the next. Informally, if we described one proof in a paragraph, and another proof in the second, then together the paragraphs would justify the conclusions of the second paragraph from the premises of the first.

Definition 6. *If $P : \Delta \vdash \Gamma = (\Phi_1, \dots, \Phi_k)$ is a proof of Γ from Δ, and $Q : \Gamma \vdash \Sigma = (\Psi_1, \dots, \Psi_m)$ is a proof of Σ, then we form the composed proof by concatenating one proof with the other.*

$$Q \circ P : \Delta \vdash \Sigma = (\Phi_1, \dots, \Phi_k = \Psi_1, \Psi_2, \dots, \Psi_m)$$

Note that this is a valid proof according to the definition given above. Given any statement Σ, the identity proof $id_\Sigma : \Sigma \vdash \Sigma$ is defined equal to the singular tuple (Σ). One may verify that this is a valid proof according to the definition, and that it truly is an identity in the category of proofs, which we denote by $\mathbf{Prf}(\mathbf{A})$. *For our purposes, we shall identify any two proofs from one statement to another as being equal.*

The other category we wish to speak about is the category of 'programming'. Though unfortunately esoteric at a first glance, we phrase our discussion of computing science in the language of lambdas, since it provides the highest level (but still formal) manner to represent computation. Really, when you get down to it, computation is just manipulating binary symbols to change some input symbols to output symbols. This is purely what this language achieves:

Definition 7. *The terms of the Lambda calculus are constructed inductively from a set of variables* $\mathbf{V}$ *and constants* $\mathbf{C}$*:*

1. *If x is a variable or a constant, then x is a term.*
2. *If x and y are terms, then so is xy*
3. *If x is a variable, and y is a term, then $\lambda x.(y)$ is a term. (this is a function which takes some value x and returns y).*
4. *If x and y are terms, then so is their tuple (x, y).*
5. *If x is a term, then so are the projection maps $\pi_1(x)$ and $\pi_1(y)$.*

If Γ associates only a finite set of variables $x_1, \ldots, x_n$ with types $A_1, \ldots, A_n$, then we shall write Γ in shorthand as $x_1 : A_1, \ldots, x_n : A_n$

To better model actual programming languages, it will be helpful for our functions to be restricted to only take certain types of inputs and return only certain types of outputs. Furthermore, without types the Lambda calculus can easily become inconsistant. This leads to the definition of a type system for the Lambda calculus, which will be the category in which the Curry Howard isomorphism takes place.

Definition 8. *The types (objects) of our language consist of types of data that you could have in your programming language. We start off with a base set of types $\mathbf{T}$, assumed to contain an 'empty tuple' type 1. And define our types as follows:*

1. *If X is a base type, then X is a type.*
2. *If X and Y are types, then so is $X \times Y$ (tuple of X and Y) and $X \to Y$ (function from X to Y).*

In C, $\mathbf{T} = \{Int, String, Float, Bool, etc\}$. In Python, $\mathbf{T} = \{Everything\}$, since the language is untyped (or at least typed behind the scenes). Using these types, we can actually interpret how our terms will interact with each other.

Definition 9. *A typing context Γ is a rule which associates to some finite subset of the variables x in the Lambda calculus a type A. We write $\Gamma \vdash x : A$ to say that x has type A in context Γ. The type of a constant is invariable among any context; $*$ always has type 1. We inductively define the typing context of a term t by:*

1. *If t is a variable, and Γ associates t with A, then $\Gamma \vdash t : A$*
2. *If $\Gamma \vdash t : A \to B$, where t is a term, and $\Gamma \vdash u : A$, where A is a term, then $\Gamma \vdash tu : B$. (What t returns with input u).*
3. *If $\Gamma, x : A \vdash t : B$, where t is a term, 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. If t is a term which has a typing context, then it is well-typed. Otherwise, it is not well-typed.

The main role of the Lambda calculus is to define computation in precise terms. In brutally simple terms, all a computer does is manipulate binary symbols in a screen. The Lambda calculus embodies this through symbol manipulation, by defining terms to be equivalent – When we compute we manipulate a sequence of symbols to the base form which we output. This is what the Lambda calculus effectively does.

Definition 10. *Equivalence of well-typed terms based on a context is based on some set of axioms* $\mathbf{A}$ *of the form* $\Gamma \vdash t \equiv u$*, where* Γ *is a type context and* t,u *are terms, assumed to both be of the same type when interpreted by* Γ*. We assume the following are satisfied in any logic (with reflexitivity, associativity, and symmetry):*

1. *If* $\Gamma \vdash t \equiv u$*, and* $\Gamma \vdash v \equiv w$*, then* $\Gamma \vdash (t, v) \equiv (u, w)$*.*
2. *If* $\Gamma \vdash t \equiv u$*, then* $\Gamma \vdash \pi_1(t) \equiv \pi_1(u)$*, and* $\Gamma \vdash \pi_2(t) \equiv \pi_2(u)$*.*
3. *If* $\Gamma \vdash t : A \times B$*, then* $\Gamma \vdash t \equiv (\pi_1(t), \pi_2(t))$*.*
4. $\Gamma \vdash \pi_1((u, v)) \equiv u$*,* $\Gamma \vdash \pi_2((u, v)) \equiv v$*.*
5. *If* $\Gamma \vdash t : 1$*, then* $\Gamma \vdash t \equiv *$*.*
6. *If* $\Gamma \vdash s \equiv t$*, and* $\Gamma \vdash u \equiv v$*, then* $\Gamma \vdash su \equiv tv$*.*
7. *If* $\Gamma \vdash (\lambda x.(t))u \equiv t_{x:=u}$*.*
8. *If* $\Gamma \vdash t \equiv u$*, then* $\Gamma \vdash \lambda x.t \equiv \lambda x.u$*.*
9. $\Gamma \vdash (\lambda x.(tx)) \equiv t$*, if* t *contains no instances of* x*.*

Definition 11. *The types* $\mathbf{T}$ *and constants* $\mathbf{C}$ *define a category, whose objects consists of types and whose morphism between a type* A *and a type* B *consist of a term* t *such that* $x : A \vdash t : B$*. If* t *is a term such that* $x : A \vdash t : B$*, and* u *is a term such that* $x : B \vdash u : C$*, then* $\lambda x.(u)(t)$ *is a term such that* $x : A \vdash \lambda x.(u)(t) : C$*; To elaborate, if* $x : B \vdash u : C$*, then for any context* Γ*,* $\Gamma \vdash \lambda x.u : A \to B$*, so that* $x : A \vdash \lambda x.(u) : A \to B$*, and thus* $x : A \vdash \lambda x.(u)(t) : B$ *(function application). We shall take as the composition of our two morphisms. We identify two terms which are equivalent when considering them as morphisms. The identity morphism on a type* A *is the term* x *in the context* Γ *such that* $\Gamma \vdash x : A$*. We denote this category* $\mathbf{Comp}(\mathbf{T}, \mathbf{C}, \mathbf{A})$*.*

Graph theory can be summarized as a lambda calculus with two base types, E and V, and two constants, $dom : E \to V$ and $range : E \to V$; no additional equations are added. Group theory is an interesting theory for someone studying symmetry. It's lambda calculus consists of a single base type G, and three constants $e : G$, $i : G \to G$, and $m : G \times G \to G$, with the following additional axioms

$$x : G \vdash m(x, e) \equiv m(e, x) \equiv x$$

$$x : G \vdash m(x, ix) \equiv m(ix, x) \equiv e$$

$$x : G, y : G \vdash m(x, m(y, z)) \equiv m(m(x, y), m(z))$$

You can do this with pretty much any algebraic theory, be it a vector space, a ring, a field, etc.

Now we have defined the two categories in which the Curry Howard isomorphism holds, what is the technique by which we will show these categories are equivalent? This equivalence is most intuitively explained by the relations of the operators in the languages – and these are most easily explained through the categorical language we will develop. One by one, we will show that many constructions in logic have a corresponding construction in the lambda calculus, related by their transformative properties. When we have expressed enough of these properties, we will simply be able to conclude that the two categories are equivalent. First we must abstractly describe these properties.

2 General Abstract Nonsense

Definition 12. *Let A and B be objects of a general category $\mathcal{C}$. A product for A and B is an object $A \times B$ 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 below.*

$$\begin{tikzcd}
 & C \arrow[dl, "f"'] \arrow[d, "f \times g"] \arrow[dr, "g"] & \\
A & A \times B \arrow[l, "\pi_A"] \arrow[r, "\pi_B"'] & B
\end{tikzcd}$$

Transformatively, $A \times B$ is the most specific object that merges data in A and data in B into one object. The product of two sets A and B in the category of sets, is simply the cartesian product. If $x \mapsto f(x)$, and $x \mapsto g(x)$, then the corresponding function h is $x \mapsto (f(x), g(x))$, and the projection mappings are $(x, y) \mapsto x$, and $(x, y) \mapsto y$. Do $\mathbf{Prf}(\mathbf{A})$ and $\mathbf{Comp}(\mathbf{T}, \mathbf{C}, \mathbf{A})$ have products? (Probably, since otherwise we wouldn't be talking about products)

Let A and B be two statements in $\mathbf{Prf}(\mathbf{A})$. What is the most specific statement that melds A and B into one statement? More precisely according to the definition, what is a statement $A \times B$ such that $A \times B \vdash A$, $A \times B \vdash B$, and if $C \vdash A$ and $C \vdash B$, then $C \vdash A \times B$? This is of course $A \wedge B$.

What is the corresponding product in the category of the Lambda Calculus. If A is a type, and B is a type, then what is the type $A \times B$ that forms the product (I've given it away in how I defined the lambda calculus). It is the product type $A \times B$. The projection morphism from $A \times B$ to A is the term $\pi_1(x)$. We have defined that if x has type $A \times B$ in a context, then $\pi_1(x)$ has type A in this context as well. Similarily, $\pi_2(x)$ is the other projection morphism.

Correspondingly, if $x : C \vdash t : A$, and $x : C \vdash u : B$, then $x : C \vdash (t, u) : A \times B$, and $\pi_1((t, u)) \equiv t$, $\pi_2((t, u)) \equiv u$, so that the morphism composition is satisfied.

In the Lambda calculus, some intuition behind what we're doing mind be eschrewed. Here is the general idea of what we're doing in Psuedocode:

```
def [A X B -> A] pi_1(x):
    return x[0]

def [A X B -> B] pi_2(x):
    return x[1]

def [C -> A X B] prod(x):
    return (f(x), g(x))
```

Definition 13. *A terminal object A in a category* $\mathbf{C}$ *is an object such that every other object B has a unique morphism from B to A. A, in a sense, has the least information out of all objects in the category, since we can relate every other object in the category to it.*

$\mathbf{Prf}(\mathbf{A})$ contains a terminal object, namely the statement $\top$. $A \vdash \top$ for any object A. $\mathbf{Comp}(\mathbf{T}, \mathbf{C}, \mathbf{A})$ also contains a terminal object, namely the type 1, since for any type A, $x : a \vdash * : 1$.

```
    def terminal[A -> 1]:
        return ()
```

There is an important problem in Computing science which I'm sure you've all heard that can be naturally expressed in terms of terminal objects. Consider the category whose objects consist of (NP) decision problems (Is $x + y = z$, Does this graph have a path from A to B with length shorter than x, etc). The morphisms of this category consist of reductions of one problem to another, specifically in polynomial time. In the past 30 years it has come to the attention of computing scientists that their is a special class of problems in this category. This class, known as the set of NP-complete problems, states that any problem in our class can be reduced to an NP-complete problem in polynomial time. Categorically, the NP-complete problems are the terminal objects in the category we have constructed. The $NP = P$ problem simply asks whether every object in the category is terminal (a very strong statement about any category, though still very possible given our knowledge at this time).

The first two transformations we have considered tell us we can construct certain properties in our category. The third property is rather self referential, stating that we can represent our morphisms as objects in the same category we are working with. Intutively, if g is a function which takes an object in C and an object in A and gives us an object B, then if we fix some c in C and consider g as a function from A to B, then c can be considered an indexing set of functions into B^A.

```
def [A -> B] exponent(x):
    return f((x,y))
```

Definition 14. *An exponent for two objects A and B in a category is an object B^A (the set 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 data in A. If C is any object and $g : C \times A \to B$ is a morphism, then there should be exactly one homomorphism $\overline{g} : C \to B^A$ making the following diagram commutative:*

$$\begin{tikzcd}
B^A \times A \arrow[r, "ev"] & B \\
C \times A \arrow[u, "\overline{g}\times id_A"] \arrow[ur, "g"] &
\end{tikzcd}$$

If $A \vdash B$ is a morphism from A to B in the category of logic, then what is the statement which represents morphisms from A to B? We let B^A be $A \Rightarrow B$. The evaluation morphism is $(A \Rightarrow B) \wedge A \vdash B$, which 'evaluates' the implication with the first term, obtaining the second term. If $C \wedge A \vdash B$, then $C \vdash A \Rightarrow B$, showing that the diagram in the definition is commutative.

If $x : A \vdash t : B$ is a type in the Lambda Calculus, then let B^A be equal the type $A \to B$ (an actual function from A to B). We have a term which transforms $(A \to B) \times A$ to B, defined by the term $\pi_1(x)\pi_2(x)$. If x is of type $(A \to B) \times A$, then $\pi_1(x)$ is of type $A \to B$, and $\pi_2(x)$ is of type A, so that $\pi_1(x)\pi_2(x)$ is of type B.

These are all the category theoretic definitions we need. In general, we can conclude any category with these properties is 'equivalent' in a sense. To emphasize this generalization of the Curry Howard isomorphism, we shall name this general construction.

Definition 15. *A category with products, exponents, and terminal objects is called a Cartesian (products) Closed (exponents) category.*

These properties are fairly strong and do not occur in any category. We have only cherry picked categories that have these properties. If a category shares many properties with another, it is likely there may be someway to connect them. We now should see why **Prf** and **Comp** are so similar – they have products, terminal objects, and exponents, and thus have many transformations which are similar in the translational sense. The Curry-Howard isomorphism realises this for cartesian closed categories. Before this, we must introduce one final concept of general nonsense, which is a precise definition of 'similarity' between two categories.

Definition 16. *Let $\mathcal{C}$ and $\mathcal{D}$ be categories. A functor F is a rule which associates to each object A in $\mathcal{C}$ an object $F(A)$ in $\mathcal{D}$, and for each morphism $f : A \to B$ a morphism $F(f) : F(A) \to F(B)$. We should have $f \circ g = F(f) \circ F(g)$: morphisms should work like they did in the original category.*

A functor allows us to consider complicated processes in a more simple setting. By providing a metaphor from one category to another, we can study a big category by means of a small category. Throughout the time you have studied mathematics with functions, you have been considering functors. Consider this example, a category whose objects consist of **R** with a point identified, and whose morphisms are functions differentiable at the point identified in the first category. Differentiation is a functor which 'linearizes' a function. The chain rule shows that the functor holds to the definition provided.

Category theory appears almost anywhere you want to draw diagrams with arrows, and in the business of computing science we love making these, so its unsurprising that Category theory shows up in software design. We can view UML as some abstract category. When we program up an instance of a diagram, we are really constructing a category of objects to which a functor applies from the UML diagram to the program you have written, preserving the properties established by the diagram.

3 Closed Cartesian Curry-Howard Isomorphism

A similarity between **Comp** and **Prf** will be described by a functor, but to obtain additional information about how the categories relate, we must investigate how the functor models the categories.

Definition 17. *If* $\mathbf{Comp}(\mathbf{T}, \mathbf{C}, \mathbf{A})$ *is a Lambda theory, then a model of* $\mathbf{Comp}(\mathbf{T}, \mathbf{C}, \mathbf{A})$ *in a cartesian category* $\mathcal{C}$ *is a functor* $F : \mathbf{Comp}(\mathbf{T}, \mathbf{C}, \mathbf{A}) \to \mathcal{C}$ *such that* $F(A \times B) = F(A) \times F(B)$*, and* $F(B^A) = F(B)^{F(A)}$*. From this, we may interpret a context* $x_1 : A_1, \dots, x_n : A_n$ *as the object* $F(A_1) \dots F(A_n)$ *in the category. A model satisfies an equation* $\Gamma \vdash t \equiv u$ *if*

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

What we mean by this is that we replace $\Gamma = x_1 : A_1 \dots x_n : A_n$ *with some variable* $x : A_1 \times \dots \times A_n$*, and replace any occurence of* x_k *with the appropriate projections, which are of course equivalent specifications according to our deduction.*

The biggest impact of this idea on Computing science results on the 'completeness theorem' for models of the Lambda calculus in a cartesian closed category.

Theorem 2. *In* $\mathbf{Comp}(\mathbf{T}, \mathbf{C}, \mathbf{A})$*, one can prove* $\Gamma \vdash t \equiv u$ *if and only if every model of* $\mathbf{Comp}(\mathbf{T}, \mathbf{C}, \mathbf{A})$ *satisfies* $\Gamma \vdash t \equiv u$*.*

Proof. Suppose $\Gamma \vdash t \equiv u$, and let $\mathcal{C}$ be a model of the lambda theory. Then $F(\Gamma \vdash t) = F(\Gamma \vdash u)$, since we identify the morphisms in the category. Conversely, suppose $\Gamma \vdash t \equiv u$ in all models of the lambda theory. Then, specifically, since $\mathbf{Comp}(\mathbf{T}, \mathbf{C}, \mathbf{A})$ is a model of the theory (take the identity functor), we have $\Gamma \vdash t$ equal to $\Gamma \vdash u$, which means exactly that $\Gamma \vdash t \equiv u$. $\square$

In fact, the ideas of Cartesian closed categories and the Lambda calculus are so related that it is hard to distinguish them (one can show that, if we consider all lambda calculi in a category, and all cartesian closed categories in another, then the two categories are isomorphic).

Theorem 3. *There is a natural construction of a Lambda calculus from every Cartesian Closed category $\mathcal{C}$. We denote $\mathbf{L}(\mathcal{C})$ the Lambda calculus constructed from $\mathcal{C}$.*

This construction, descends into a rather nitpicky argument and we avoid it here. It shows that, if we consider the set of all proof systems as a category (with proof preserving maps), and the set of all lambda calculi as a category (with term equivalence preserving maps), then there is a natural functor between the two. Like how partitions and equivalence relations are equivalent, for every Cartesian Closed category we can construct a lambda calculus $\mathbf{Comp}(\mathbf{T}, \mathbf{C})$ which is isomorphic to the original category (up to equivalence).

Corollary 4 (Curry-Howard-Lambek-Lawvere-Scott Isomorphism)**.** *Every cartesian closed category* $\mathbf{Prf}(\mathbf{A})$ *is isomorphic to some Lambda calculus* $\mathbf{Comp}(\mathbf{T}, \mathbf{C})$.

It follows from this that the most basic logic category (with the minimal axioms) must then be isomorphic to the most basic Lambda calculus (with a single constant $*$ and minimal types). If we want to prove a statement in this logic category, by the above completeness theorem we need only construct a well-typed statement in this category. We can write programs to deduce whether a small subset of statements are well-typed, and thus we are able to check proofs with a computer.

4 Consequences & Category Theory elsewhere in CS (If I still have time at the end of the Talk)

The point of all this build up in category theory is that the ideas behind the Curry-Howard isomophism become obvious. Transformationally, the structures have their inner properties revealed. Furthermore, in this technique of identifying some logical object with a specific type of category (here a cartesian closed category), we discover a more general techniques for showing how computing science and logic relate. If we have a new theory of computation, simply find a category that represents it and identify the properties that designate that logic. Many correspondences have been identified in this manner:

1. Conjunctive Logic: Cartesian Categories
2. Positive Logic: Cartesian Closed Categories
3. Intuitionistic Propositional Logic: Bicartesian Closed Categories
4. Classical Prepositional Logic: Bicartesian Closed Categories with -Elimination
5. Linear Logic: *-autonomous logic
6. First-order logic: hyperdoctrines
7. Martin-Löf type theory: locally cartesian closed categories

Studying these abstract types lead up to a recent event, where an entirely new field of computing science was created: Homotopy type theory. The theory relates the idea of Homotopy (a continuous stretch and bend of space studied in topology) to the study of types. Homotopy type theory enables us to translate a more comprehensive set of mathematical arguments into proofs, and thus makes the idea of proving programs correct much more viable in the near future.

Type theory is not the only place that category theory excels in pedagogy, though it is all we are able to talk about in the time alloted here. If you've every programmed in Haskell before, you may have noticed that many terms in this talk are used in the language. This is because Haskell models complicated functional programming concepts in the language of category, simplifying explanations of complicated topics when specified in the unifying categorical language (There is even a formal category **Hask** which models it and is studied academically). Functors, Monads, Monoids, Natural transformations, etc. Unfortunately, most people learn about category theory before learning Haskell, which makes these concepts much harder to learn.

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] Serge Lang *Algebra*

[5] Carsen Berger *A Categorical Approach to Proofs as Problems*

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

[7] The n-lab *Lambda Calculus*

[8] Scott Lambek *Introduction to Higher Order Categorical Logic*

[9] Michael Spivak *Calculus on Manifolds*

[10] Easterbrook *An Introduction to Category Theory for Software Engineers*

[11] Mossakowski, Rabe, De Paiva, Schröder, Goguen *An Institutional View on Categorical Logic and the Curry Howard Isomorphism*