Abstract Nonsense and the Curry Howard Isomorphism

Jacob Denson

March 19th, 2015

“A single simple principle can masquerade itself as several difficult results.”

Michael Spivak

In 1931, the logician Kurt Gödel announced his most important result. 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. His ‘Turing machines’ have become much more than an element of a logician’s toolbelt. Generalization and abstraction simplify big ideas and complicated thought procedures. With the correct language, a complex argument unravels into a straightforward reasoning process. In this talk, I argue that category theory effectively abstracts many fundamental concepts of computing science.

A theory’s importance lies in its power to solve interesting, practical problems. To test category theory’s power, we will untangle a particularly complicated computing science principle. Without the language of category theory, the principle evades precise definition:

Theorem 1 (The Curry-Howard Isomorphism (Preliminary))**.** *There is a deep correspondence between sound mathematical proofs and computer algorithms which always terminate (they do not possess an ‘infinite loop’).*

Through the isomorphism, we can think of logic as computing science, and vice versa. Just as we can think of computation as the manipulation of sequences of binary symbols, a proof is manipulation of a prespecified language of symbols to argue an idea. This insight is not only intuition, however; the isomorphism is a precise statement enabling us to move between the two, separate worlds of mathematical logic and computing science. More concretely, we can check logical arguments 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 in precise terms, but also shows *why* the theorem is true, which is equally important to the use of the isomorphism. It is particularly effective in computing science because it shifts the focus of mathematics from sets, and the objects they contain, to transformations, and the manner upon which they operate on data. When programming, our focus isn't on data, but on the segments of computation which we combine to form a complete algorithm. Set theory places emphasis in the wrong place, on the data which mathematics analyzes. If we forget about the data, the discussion of a certain problem becomes tenfold simpler.

Definition 1. *A* **Category** $\mathcal{C}$ *consists of a class of objects, and a family of transformations between those objects. We specify a collection of objects* $Obj(\mathcal{C})$*, together with a family* $Mor(A, B)$ *of morphisms between any two objects* $A, B \in Obj(\mathcal{C})$*. We write* $f : A \to B$ *for a morphism* $f \in Mor(A, B)$*. The most important part of a category is a way to compose transformations. For* $f : A \to B$*, and* $g : B \to C$*, 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. The diagram

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

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

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

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

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

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

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 these 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.

You've almost definitely met a category before. Here are a few examples from basic mathematics.

- **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 categories, but they provide an example of how category theory is used in other parts of mathematics.

The Curry-Howard isomorphism can be stated as an 'equivalence' of two quite different categories. We must describe the objects and transformation which logic discusses, and the objects and transformations that computing science discusses. By looking into these categories individually, we will begin to see certain similarities which, when all put together, give us the Curry Howard isomorphism.

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 shall introduce propositional logic in a manner which makes it easy to compare with the Lambda calculus, but all the concepts we consider can be discussed in all other models of propositional 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, such 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.*

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 here, we will let capital greek letters like Γ*,* Σ*, and* Δ *stand for arbitrary formulae.*

Note that negation is not considered, and cannot be formulated in our system. It is this 'intuitionistic' side of logic than computation corresponds naturally to. After all, what does it mean to 'negate' a program? One can develop a category of logic including negation, and this has some interest computationally, but we will not discuss this interpretation here.

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 five 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'* $\lambda P : \Gamma \vdash (\Delta \Rightarrow \Sigma)$

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, and the corresponding semantics of this system would reduce to classical propositional logic (truth tables).*

The notation $P : \Sigma \vdash \Delta$ is designed to fit the notation for a morphism $f : A \to B$. This is purposeful: the logic category $\mathbf{Prf}(\mathbf{V}, \mathbf{A})$, will be the category whose objects are statements, and whose morphisms are proofs from one statement to another. The only problem with the proofs we have defined here is they have no concrete representation, and therefore we have no natural way of considering when two proofs are equal. Hilbert defines proofs as sequences $(\Phi_1, \dots, \Phi_n)$ with $\Phi_1 = \Gamma$ and $\Phi_n = \Delta$, and if we have a proof of Σ from Δ of the form $(\Psi_1, \dots, \Psi_m)$, we can form the composition $(\Phi_1, \dots, \Phi_n, \Psi_2, \dots, \Psi_m)$. The identity proof $\text{Id} : \Sigma \vdash \Sigma$ is then the one element sequence (Σ), and satisfies the identity law. Another point of view, if we only care about semantics, would be to identify any two proofs between the same statements. Then any proof $P : \Sigma \to \Sigma$ acts as the identity. These choices will result in different categories with the same logic. This is an inherent problem with representing propositional logic as a category, and the weakest link between propositional logic and computation – functions have a natural syntactic representation in computing, whereas proofs are essentially semantic. For now, we will consider an arbitrary equivalence of proofs, provided that the identity proof $\text{Id} : \Sigma \vdash \Sigma$ acts as a categorical identity. Thus we have a category with which to discuss logical statements and proofs.

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 will be 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 $(\pi_2 y)$.*

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. 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 programming languages, we need to introduce a type system to the Lambda calculus.

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 cartesian product) and $(A \to B)$ (the exponential).*

We interpret $A \times B$ as the 'tuple type' consists 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, $\mathbf{T} = \{\text{Int}, \text{String}, \text{Float}, \text{Bool}, \dots\}$, whereas in Python $\mathbf{T} = \{\text{everything}\}$, since the language is untyped (sort of).

Definition 6. *A typing context* Γ *is a rule which associates to 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* $x_1, \ldots, x_n$ *with types* $A_1, \ldots, A_n$*, we will let* Γ *be denoted* $x_1 : A_1, \ldots, x_n : A_n$*. 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. Unlike in propositional logic, where proofs have no natural definition of equivalence, we already have a standard way to compute the equivalence of two functions, which we can supplement with other equivalences.

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 have 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 are single-variable functions which take one type, and spit out another type. To be precise, a morphism between two objects A and B is a term t and 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. 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.