A Primer on Type Theory

Tora Ozawa

May 2023

1 Introduction

This paper was initially meant to culminate what I have learned through an independent study on type theory and related topics. I mainly learned through a series of online lectures presented by the HoTTest Summer School which took place during the summer of 2022. I did look through and sift through lots of other resources, and the interested reader can find them here with some annotations. This paper was also paired with a talk I gave at the University of Rochester Computer Science Systems Seminar, the slide deck for which can also be found at the link. This is my first expository work and I quickly learned that condensing material, be it broad or technical, is rather difficult. More specifically with regards to type theory and logic, the subject spans such a huge space, and a lot of it is hard to motivate to a large audience. Moreover I realized it is impossible to do all the topics I learned throughout semester proper justice within these pages, either for the sake of rigor or pedagogy. Even if the entirety of this paper were dry technical details and definitions, I am convinced it will not be able to fit even half of what I would now consider the "basics" of type theory. Moreover in terms of utility of the paper it would largely overlap with wonderful resources such as Awodey's *Homotopy Type Theory*, the HoTTest lectures as mentioned, and more recently Egbert Rijke's *Introduction to Homotopy Type Theory* and many more. Comparatively, a paper written after a semester's worth of learning which attempts to exhaustively and rigorously cover what I learned (in a relatively short page count) would undeniably be of lower quality in all regards. Thus I took an entirely different approach. The intended purpose of this paper for the reader is to have it serve as a primer on the subject (as the title suggests). When I first started learning type theory before this semester started, I found everything extremely difficult to digest despite my computer science background and exposure to logic. The formal syntax seemed bulky

and unfamiliar, the ideas seemed rudimentary when explained at the high level, deceptively so, as the finer details were hard to internalize. I benefited the most when I simply exposed my self to the basics by giving it a relatively quick pass (with varying levels of understanding and comprehension), and then returning to it later with more attention to details. My hope is that this paper can serve as part of that initial pass for those curious about the subject. As a result the details, ideas and connections to other areas will be presented in varying degrees: from thorough explanations to handwaving to brief allusions. As such, I've intended for this to be accessible to people with various backgrounds: the only real prerequisite being exposure and familiarity with propositional/first order logic, inference rule notation, some programming, and set theory. Of course any more familiarity with topics in mathematics and computer science will only help as well.

With formalities out of the way, let us now discuss some motivations and broad ideas for and within type theory. One of the main prospects of the subject is that it can serve both as a foundation for mathematics as well as a logical, deductive system. It can at many times be both simultaneously, because of the nature of how it is setup. Set theory and first order logic in tandem are the basis for the practice of modern mathematics. In contrast, type theory internalizes lots of mathematical constructions, in a sense taking them for granted, thus bringing foundations closer to practice. It also turns out that the formal system has convenient computational interpretations and mechanizations which makes it suitable to underlie machine checked proofs. Because of its relation to the lambda calculus, it also turns out that lots of type theoretic ideas, specifically dependent types, can be embeded into programming languages, especially functional ones. The idea being that good programming abstractions may actually be type theoretic ideas, and vicea versa. Similarly,

proof assistants and methods which incorporate them are also of interest because of their power to fully verify software systems. And although users may not necessarily have to be aware of all the intricacies of type theory, in order to get better theorem proving systems, one needs to first spread awareness and properly teach the foundations. And to this end, I believe type theory is a great entry point for formal methods, logic, and perhaps mathematics and computer science as a whole. Paige Randall North, a mathematics and computer science professor at Utrecht University, illustrates connections type theory has to other areas quite nicely through this diagram:

This diagram of course is to be taken with a grain of salt, partly because it comes from the HoTTest lectures directly which is in a way advertising homotopy type theory (hence why it is front and center). However it gets the point across that these areas, which may seem distant at first, are in reality connected quite nicely and deeply. Sure one can argue about the arrangement and the choice of edge connections, but it doesn't take away from the idea that the ties to mathematics, logic and computer science give it huge potential to serve as a pedagogical bridge between the areas.

2 Judgements, Formations, and Some Types

2.1 A Note on Interpretations: Curry-Howard, Sets, Spaces

Like any scientific or mathematical theory, useful insights are derived when we have different interpretations of certain ideas. Type theory is no different. For the rules we are teaching here, we can think of types as sets to get us started with intuition. We would also like the reader to remember the maxim: "propositions are types, proofs are programs". This idea comes from the Curry-Howard Correspondence, which refers to an umbrella of results linking things ranging

from natural deduction to the simply typed lambda calculus, and by extension, type theory. The specifics of these ideas will emerge slowly throughout our presentation of the technical aspects of type theory. In a similar vein, we will also allude to and briefly mention the idea that types can be thought of as spaces, which is a core idea in homotopy type theory. These three viewpoints of types as sets, propositions, and/or spaces need not be explained in full detail here to help with intuition and to properly ground the technical ideas. Simply having them in mind and trying to connect the dots oneself is sufficient for our purposes.

2.2 Judgements

To start us off, we will begin by defining what we will call "judgements". Within type theory, we typically have 4 kinds of judgements to work with:

- Type A is well defined in a context Γ:

$$\Gamma \vdash A \text{ type}$$

- a is an element of type A:

$$\Gamma \vdash a : A \text{ type}$$

- Types A and B are judgmentally equal:

$$\Gamma \vdash A \doteq B \text{ type}$$

- Elements a and b are judgmentally equal:

$$\Gamma \vdash a \doteq b : A \text{ type}$$

From here it is a natural question to ask what exactly is a judgement? Type theory is in many ways a language as much as it is a precise, formal system and

so you can think of these judgements as "grammatically correct" sentences in the language of type theory. Meaning all of our proofs in type theory will take the form of one of these four structures; the types, elements and terms may look more complicated, but the essence of what judgment is being used will clearly be among these 4. We also note that $\vdash$ can be read as "derivable" or "entails" so all of the judgements can be interpreted as: "in context Γ, we can derive...".

We would like to note that the second example is referred to in many ways. a can to be referred to as a term, element, proof, or witness (among other names) depending on the person or source. These terminologies are essentially interchangeable. It is just that they hint at the different interpretations of constructions in type theory and so different people will tend to stick to different subsets of these terms.

It can also be observed that all of these judgments start with Γ, which is the context, and what comes after the turn-style is an "assertion". Rijke defines context as the following:

Definition 2.1 *A **context** is a finite list of variable declarations [*Rijke].*

$$x_1 : A_1, x_2 : A_2(x_1), ..., x_n : A_n(x_1, x_2, ..., x_{n-1})$$

such that for all $1 \leq i \leq n$, the following judgement is derivable:

$$x_1 : A_1, x_{i-1} : A_{i-1}(x_1, ..., x_{i-2}) \vdash A_i(x_1, ..., x_{i-1}) \ \textit{Type}$$

Here both the types and terms are "dependent" or "parameterized" by previous terms. however it is possible that the type/proposition does not depend on previous terms in which case the additional annotation is unnecessary. Moreover it is important to note that terms depend on previous terms implicitly, however this is generally not annotated explicitly. To remember this, it helps to think of contexts as a list of variable declarations in a programming language. A sort of referencing enviornment if you will. For instance if you declare "Int

$x_1 = 3$" in some programming language, then the next variable you declare can depend on x_1. More concretely, we can declare "Int $x_2 = x_1 + 2$". And even if b did not actually depend on a, the actual program execution of making the assignment of x_2 still implicitly depends on x_1. Of course this comparison is ignoring details such as instruction re-ordering that a compiler may do. However the metaphor is still illustrative that terms depending on previous terms isn't such a foreign idea. It also emphasizes the importance of the ordering of the terms in any given context, which we will see an example of later on. As a small side note, our context can also be empty. It also need not be as complicated as the general form in the formal definition. As such, we will not use the full power of the definition too much, as most of contexts we will use only consist of a few terms and types at most. It is good to be aware of it when encourtering more complicated setups in outside of this paper. The definition also does have an inductive flavor, something which is at the heart of essentially every type theory.

Before we get into some of the more technical and esoteric ideas, it's also important to take a moment here to really grasp what these seemingly simple judgments buy us in terms of abstractions. Essentially, with what we have now, we are able to define collections of objects and assert that an object belongs to it. Additionally the system intrinsically allows us to state when two terms or types are equal. Conversely, it also important to acknowledge what is **not** here. There is no way to state a judgmental equality between two elements which belong to different types. Similarly, there is also no notion of "sub-types". Given our interpretation that types are sets, One might expect that we should postulate sub-types as subsets since they are ever-present in traditional mathematics, as are sub-types in programming languages. Conversely however, there is nothing inherently restricting us to create similar notions of subsets or

subclasses. Suppose we have we have types $\mathbb{Z}$ and $\mathbb{N}$ and elemetns $1_z : \mathbb{Z}$ and $1_n : \mathbb{N}$. Given that these elements are both meant to represent the number 1, it isn't hard to believe that we will have a way to "correspond" these two elements, it's just that they are not inherently the same. If we want to treat types as sets, it is as if we have prevented any notion of sets intersecting, that while we may create correspondences and associate elements of one set with another, we have got rid of any ambiguity as to which set an element belongs to. And in this sense type theory can be compared to a strongly typed programming language, whereas set theory is closer to, ironically enough, an untyped machine language.

2.3 Type Formers

Put simply, type formers are the rules which define and specify our types. We will shortly see examples of these in this section, namely pi, implication types, and product types. And although this paper, among other resources, will mention "working with" or "teaching type theory", it is important to mention that there is no strictly canonical type theory. Some who works with a type theory is perfectly free to choose which type formers they include so long as the system is still consistent [*HoTT 2]. You can sort of think of it as "picking your axioms" in your formal system. One can always be more specific and also implicitly specify what theory they are working with, whether it be dependent type theory, cubical type theory, homotopy type theory, etc. We will be teaching the basics of Per Martin-Löf's dependent type theory, which is the base of many type theories. Because of this, the distinction is not too important for our purposes but it is good to be aware of this fact nonetheless.

Throughout the rest of this paper, we will showcase each type former in a structured way. Starting with a **formation** rule, **introduction** and **elimination** rules, and finally **computation** rule(s). The **formation** rule will tell us

that a type actually exists. The **introduction** and **elimination** rules will tell us how to construct and deconstruct terms of a type, respectively. The **computation** rule(s) will define judgmental equality between terms used to construct a type, and terms produced when deconstructing a type. There is also an additional nuance that **introduction** rules will in a sense define the canonical terms of a type, while the deconstructive nature of **elimination** rules will be used to give proofs for dependent types. We will reiterate these points as we go through examples.

2.4 Product Types

As mentioned previously, lots of underlying ideas and assumptions made by type theory are motivated by the notion of internalization. As such we will first introduce product types. And first, we have to postulate the existence of such a type. Therefore we need to specify a $\times$-**formation** rule. We will use the following:

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type}}{\Gamma \vdash A \times B \text{ type}}$$

In plain English, we can read this inference rule as saying that given type A and type B exist in (the same) context, then we can assert that the type $A \times B$ exist. When introducing a new set of rules for a type, we will always start with the formation rule. Also note that we put the first two judgements to make it clear that the existence of A does not depend on B, and also so that our judgements fits one of the forms we listed earlier (remember a context allows **only** variable/element declarations).

Now we want to define how the terms are constructed. To do so, we postulate the following $\times$-**introduction** rule:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B}$$

Noticing the difference in the clauses before the inference in quite important. For the $\times$-**formation** rule, we only have to know that the types exist. Whereas here we actually have to have elements of the types before constructing a term of the product type. Notice again we the first two judgement are separate to avoid dependency issues. At high level, this specific type is internalizing the notion of a cartesian product on two "sets". Likewise, similar to any Cartesian product in classical mathematics, we can always deconstruct our pairs which are contained in this Cartesian product of sets. For example consider the standard, two dimensional Euclidean space, $\mathbb{R}^2$. We know points in this space are of the form (a, b), and from here we can deduce a is the coordinate of the x-axis, and b is the coordinate of the y-axis. Analogously, once you have a term of a product type, $(a, b) : A \times B$, you can always extract either of the terms from the original types used to construct this new term. Likewise, we have the following $\times$-**elimination** rules to do so:

$$\frac{\Gamma \vdash x : A \times B}{\Gamma \vdash pr_1(x) : A}$$

$$\frac{\Gamma \vdash x : A \times B}{\Gamma \vdash pr_2(x) : B}$$

Where $pr_1(x)$ and $pr_2(x)$ can be read as the left and right projections, respectively. Inserting this into our schema of propositions as types makes it natural to understand and remember the $\times$-**introduction** and $\times$-**elimination** rules. The product type can be interpreted as a logical conjunction, and so once we have a proof of type A and a proof of type B, we have a proof of A and B. Conversely, if we have a proof of $A \times B$, we know that we have a proof of both A and B. Calling these projections is hinting at the idea of thinking of types as spaces. Going back to our previous example, given a point (a, b) in $\mathbb{R}^2$, extracting one coordinate, can be viewed as mapping the point to one of the axes: "pulling" a or b out of the point (a, b) is like mapping (a, b) to the point

$(a, 0)$ and $(0, b)$ respectively. This detail, in addition to the set theory interpretation, also justifies the choice of using the familiar Cartesian product notation as opposed to the more traditional logical conjunction. And it concretely shows how the set theoretic construction is actually closely related to logic.

As promised, We also have $\times$-**computation** rules to precisely define judgmental equality:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash pr_1((a, b)) \doteq a : A}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash pr_2((a, b)) \doteq b : B}$$

$$\frac{\Gamma \vdash x : A \times B}{\Gamma \vdash (pr_1(x), pr_2(x)) \doteq x : A \times B}$$

Although a bit cluttered in terms of notation, these rules may seem fairly obvious and that is part of the reasoning for assuming them. Note that if we wanted to be more precise for each of these rules, and perhaps overly pedantic, we could insert an intermediate inference to clearly state that in context Γ, we have $(a, b) : A \times B$. From this rule we can also get the sense that type theory gives very precise definitions for manipulating syntactic representations of objects, hence why it is the basis for proof assistants. Thinking of our element (a, b) as a tuple, these rules say that extracting the elements out yields the same elements we "put in".

Next, we will introduce implication types. As the name suggests, they will encode propositional implication in our propositions as types paradigm. Again, we must have a judgement that the type actually exists so we first need a $\rightarrow$**formation** rule:

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type}}{\Gamma \vdash A \rightarrow B \text{ type}}$$

And then we can accompany it with the $\rightarrow$**introduction** and $\rightarrow$**elimination** rules below:

$$\frac{\Gamma \vdash a : A \vdash b : B}{\Gamma \vdash \lambda(a : A).b : A \to B}$$

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B}$$

The first inference rule, the →**introduction** rule, is saying that if we jave a proof a of A, and it can give us a proof b of B, then we have a proof of the type $A \to B$. In other words we have abstracted the process of deriving B from A into a function. The term looks more complicated, and is perhaps unfamiliar, but it is mostly just notation taken from the lambda calculus (partially for historical reasons). To make sense of the syntax, first note that our context Γ is the same above and below the inference. What we have changed comes after: λ simply denotes a function, $(a : A)$ denotes the argument and it's type A (the type is sometimes omitted), and the "." is just notation for separating the input and the output. From here, it should be no surprise as to what the elimination rule is. Given a function, or a proof, of type $A \to B$, and term a $a : A$, we can plug it into the function to get a term of type B. Now to relate these two rules, and to define judgmental equality clearly, we will also specify →**computation** rules as follows:

$$\frac{\Gamma \vdash a : A \vdash b : B \qquad \Gamma \vdash x : A}{\Gamma \vdash (\lambda(a : A).b)x \doteq b[x/a] : B}$$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash \lambda(x : A).f(x) \doteq f : A \to B}$$

To break these inference rules down, we first want to note that we have again "skipped a step" by implicitly applying the →**introduction** and →**elimination** rules. As for the end term in the first →**computation** rule, the component $(\lambda(a : A).b)x$ is just notation we have employed for applying the function $\lambda(a : A).b$ to x, "plugging x in" if you will. Of course other sources may use slightly different forms, but they should still be somewhat similar. On the

other side of the judgemental equality, we have $b[^x/_a]$, meaning that we are substituting a with x. Recall that derived terms always depend on the ones which precede their derivations, meaning the b which gets output from $(\lambda(a : A).b)x$ is not the same as the one in the initial condition, but rather a different one which now depends on x instead of a. From a programmatic standpoint, this should be quite familiar, we are just employing in a different setting with syntax. A function in a programming language may be of the form "fn(a)" when we declare it, and may take the variable a and use it to define $b = a$ and then return b. However when we call the function, say with "fn(x)", a is essentially replaced with x. The idea is exactly the same here, only this time our terms, our arguments, can be interpreted as proofs. For the second $\rightarrow$**computation** rule, it suffices to think of it as introducing a "dummy variable" and then using the $\rightarrow$**introduction** and $\rightarrow$**elimination** rules to create $\lambda(x : A).f(x)$. Intuitively, this is no different than f itself and it seems reasonable to assume it and in fact it is safe to.

As a side note, it can be argued that this introduction to logical implications is much a more well connected to programming than the dry truth table of or 1s and 0s that the computer science student is accustomed to. It relates directly to notions of types and functions in programming to ideas in logic. It's a "cute connection" between type theory and the programming languages, even more so for functional languages, which helps synthesize the two areas pedagogically.

Going back on track, the rules so far can be seen as very natural internalizations of some of the rules of propositional logic. We will continue along this line but also go back to the idea of dependent types: types which depend on terms, much like we saw when defining context. So far, our terms or proofs are implicitly dependent on previous terms, however our type formers are yet to encode types which depend on terms, which is the very essence of (dependent)

type theory. As such, we will first introduce dependent function types, which can also be referred to as "pi" types, with the following $\prod$-**formation** rule:

$$\frac{\Gamma \vdash x : A \vdash B(x) \text{ type}}{\Gamma \vdash \prod_{x:A} B(x) \text{ type}}$$

Note that this is the first formation rule which requires a term to be constructed. This is because $B(x)$ is a type family. Meaning that for different x, $B(x)$ is in a way a different type. The most common example of a dependent type in a programming setting is an array with a fixed size parameter. Of course, all types derived after a term has been declared are implicitly type families, but in this case between each type in the type family is judgmentally equal. Likewise, when the dependency is irrelevant, it is not notated. To follow up, we have the $\prod$-**introduction** and $\prod$-**elimination** rules for dependent function types:

$$\frac{\Gamma \vdash x : A \vdash b : B(x)}{\Gamma \vdash \lambda(x : A).b : \prod_{x:A} B(x)}$$

$$\frac{\Gamma \vdash f : \prod_{x:A} B(x) \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B(x)}$$

and the $\prod$-**computation** rules for dependent function types:

$$\frac{\Gamma \vdash x : A \vdash b : B(x) \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda(x : A).b)a \doteq b[a/x] : B[a/x]}$$

$$\frac{\Gamma \vdash f : \prod_{x:A} B(x)}{\Gamma \vdash \lambda(a : A).f(a) \doteq f : \prod_{x:A} B(x)}$$

We can observe that these rules are very similar to the original implication type we had previously. And in fact the implication type is simply a special case of our dependent function types, where the type the term outputs, does not actually depend on the term which is inputted. Lots of proof assistants when implementing these formation rules also don't make the distinction between dependent function types and implication types, and instead implement it as one "unified" rule. Going back to the idea of "propositions as types", we can

see that dependent function types act as universal quantifiers: given any x of A, $B(x)$ holds.

We will now give an example which incorporates the things we have seen thus far. Suppose we want to prove, or provide a term, of the following type (in an empty context):

$$\prod_{x:A\times B} B \times A$$

To do so fully formally, we can construct the following proof tree:

$$\dfrac{\dfrac{x : A \times B \vdash pr_2(x) : B \qquad x : A \times B \vdash pr_1(x) : A}{x : A \times B \vdash (pr_2(x), pr_1(x)) : B \times A}}{\lambda(x : A \times B).(pr_2(x), pr_1(x)) : \prod_{x:A\times B} B \times A}$$

Notice that in the last line we have defined a term with type $\prod_{x:A\times B} B \times A$ as desired. We admit that here it may seem like we are being a bit terse or some notation is being abused. For example, although we assumed the empty context, it may seem that that $x : A \times B$ is acting as our context throughout the derivation. But what we mean by the empty context is that our final proof of the type we trying to prove is in an empty context. And likewise, it usually helps to try the derivation from the bottom up, determine the types needed at each step and then fill in what terms are being used. Another notation shortcut we use is in the top two clauses: since the turn-style means "derivable", we can shorthand our $\times$-**elimination** rule to one line. Reading this from the bottom up, we can see that if we want to derive a term for $\prod_{x:A\times B} B \times A$, we will need to use the $\prod$-**introduction** rule. So our second to last line should be of the form $A \times B$ derives $B \times A$. Now treating $x : A \times B$ as our context, we can see that to construct a term of type $B \times A$ will require the $\times$-**introduction** rule. And thus our last line requires us to derive a terms of B and A from a term of $A \times B$, which will require the $\times$-**elimination** rule.

Under our logical interpretation of types, $\prod_{x:A\times B} B \times A$ is stating that for propositions A and B:

$$A \wedge B \to B \wedge A$$

given our usual view that products are conjunctions, and functions are implications. We can also often interpret derived terms computationally: given a tuple (a, b), our function returns the tuple (b, a), a detail which again pedagogically synthesizes logic and computation for the learner. Allowing for computational, programmatic intuition can be leveraged to give rise to logical intuition.

Also make that note that we used the case for dependent function types which "reduces" to using it for normal implication types since our implied type does not depend on any terms. So to get the same logical interpretation, we could have formulated our original proposition (as a type) with:

$$A \times B \to B \times A$$

That being said, how do we determine whether these two types are equivalent? Clearly we can see that they have the same logical interpretation, but in order to define any equivalence classes we need to first what we mean by equivalence. In fact, lots of type theory, and homotopy type theory, deals with what it means for terms or types to be equivalent. This notion usually culminates in ideas such as the identity type, universes and the Univalence axiom. One reason for this is that it turns out even proving that for the terms $0 : \mathbb{N}$ and $1 : \mathbb{N}$, where $\mathbb{N}$ is the type of natural numbers (formally introduced later), we cannot prove $0 \neq 1$ without the notion of universes [* HoTT 4]. This difficulty may be unexpected, but given this it is not hard to imagine that proving judgmental, or any equality of types, in general is rather difficult or impossible with our current set of tools. Part of this reason for this complexity is that theories which pass a certain threshold of complexity admit multiple representations of the same objects. Thus the line between propositional equality of types, judgemental equality of types, and definitional equality of types becomes a lot blurrier. Although it doesn't feel great to leave the questions unanswered and

the concepts undefined, we will not have time to approach these things. Like the treatment of some of the other idea so far, our goal is to make the reader aware of them, possibly for further exploration on their own.

We would like to end this section by acknowledging that those familiar with propositional logic in it's more "traditional" form will notice that syntactically our proof trees and inferences so far are extremely similar to that of natural deduction.

Figure 1: an example of natural deduction from lean

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A}\,1 \qquad \dfrac{\dfrac{}{(A \to B) \wedge (B \to C)}\,2}{A \to B}}{B} \qquad \dfrac{\dfrac{}{(A \to B) \wedge (B \to C)}\,2}{B \to C}}{C}}{A \to C}\,1}{(A \to B) \wedge (B \to C) \to (A \to C)}\,2$$

It's as if we simply annotated our proof tree in our system of propositional logic, and annotated it with terms. And viewing things at a surface level, this is certainly true. However there is an important distinction to be made. By encoding propositions as types, and then constructing terms (or proofs) for them, we explicitly carry information about how we proved our proposition: meaning that the our current framework is proof relevant. Formally, classical logic is only concerned with the truth value of a proposition. Of course in practice, we do care how we prove things. It's how we learn new skills and apply techniques to tackle new problems. Thus this type theoretic notion of proof relevance is once again bringing the practice of mathematics closer to its foundations. And it is doing it in a way which can be built on top of intuitions built in computer science.

3 Inductive Types

3.1 Natural Numbers

Our first example of inductive types will be the natural numbers, denoted as $\mathbb{N}$. The $\mathbb{N}$-**formation** rule is as follows:

$$\frac{}{\vdash \mathbb{N} \text{ type}}$$

The $\mathbb{N}$-**introduction** rules:

$$\frac{}{0 : \mathbb{N}}$$

$$\frac{\Gamma \vdash n : \mathbb{N}}{\mathrm{s}\ n : \mathbb{N}}$$

Here the "s n" denotes successor term to n, $n + 1$ if you will. If one has never encountered this inductive definition of the natural numbers, this may seem quite roundabout and strange. After all the existence of numbers is something we often take for granted, and certainly most mathematics or computer science courses will have no qualms with one just asserting that an arbitrary number exists. However we are dealing with logical foundations. When defining a type theory, a formal system, we are dealing at a much lower level of abstraction. It is a necessity to specify all constructions precisely, hopefully so a machine can internalize it. And to this end, one might think that something similar to a binary representation traditionally used might be more suitable but this has its problems. However this is a good place to note that by "computational" interpretation and power, we are referring to the ability to carry out inferences, not necessarily to actually compute numbers. This representation of numbers if used for an industrial level programming language would be extremely inefficient in comparison to binary. For example "1" is essentially "s 0", and "2" is essentially "s s 0" and so on. Each application of "s" is similar

to a function call adding an increment to its input. However this representation is extremely good for carrying out inferences. This is because this recursive definition is completely general and makes it suitable to perform mathematical induction. This idea can be demonstrated in the $\mathbb{N}$-**elimination** rule:

$$\frac{\begin{array}{c}\Gamma, x : \mathbb{N} \vdash D(x)\ \text{type} \\ \Gamma \vdash a : D(0) \\ \Gamma, x : \mathbb{N}, y : D(x) \vdash b : D(\text{s}\ x)\end{array}}{\Gamma, x : \mathbb{N} \vdash \text{ind}_{\mathbb{N}}(a, b, x) : D(x)}$$

Note that we align the clauses vertically as opposed to horizontally mostly for the sake of space. Aside from making notation more convenient however, someone familiar with functional programming may recognize this as a match statement of sorts. The first clause is "declaring" the kind of type family we want to produce a term for (hence why the judgment only asserts the existence of the type family). The next two clauses exhaust all cases for the dependent type in a way which thematically aligns with mathematical induction: we prove/provide a term for the base case/type, and then assuming we can prove $D(x)$, if we can then prove $D(\text{s}\ x)$, then we can provide a proof for any type in the type family $D(x)$. And again to formalize judgmental equality, we have the following $\mathbb{N}$-**computation** rules:

$$\frac{\begin{array}{c}\Gamma, x : \mathbb{N} \vdash D(x)\ \text{type} \\ \Gamma \vdash a : D(0) \\ \Gamma, x : \mathbb{N}, y : D(x) \vdash b : D(\text{s}\ x)\end{array}}{\begin{array}{c}\Gamma \vdash \text{ind}_{\mathbb{N}}(a, b, 0) \doteq a : D(x) \\ \Gamma, x : \mathbb{N} \vdash \text{ind}_{\mathbb{N}}(a, b,\ \text{s}\ x) \doteq b[{}^{\text{ind}_{\mathbb{N}}(a,\, b,\, x)}/_{y}] : D(\text{s}\ x)\end{array}}$$

Here we have written things in shorthand again, packaging two rules into one, the important point being they have the same initial conditions. The last line may be confusing however knowing how to "read it" helps quite a lot with

understanding. First off, it might be confusing why the last "argument" to the induction term is "s x" and not "x". Remember we can read the turn-style as "derives", so we can think of it as implicitly using the second introduction rule of the natural numbers to get a term "s x" before providing it as an "argument" to the induction term. Now semantically, what this judgement is really doing is relating the elimination rule with the third clause above the inference. It is saying that the proof $\text{ind}_{\mathbb{N}}(a, b,\ \text{s}\ x)$ is exactly the same as the one provided by the inductive step, $\Gamma, x : \mathbb{N}, y : D(x) \vdash b : D(\text{s}\ x)$, where b depends on y and we substitute y with $\text{ind}_{\mathbb{N}}(a, b, x)$.

To solidify understanding, here is an example of using our type formers to define/derive an "add" function.

$$\dfrac{\dfrac{\dfrac{x : \mathbb{N} \vdash 0 : \mathbb{N} \qquad x : \mathbb{N}, y : \mathbb{N}, z : \mathbb{N} \vdash \text{s}\ z : \mathbb{N}}{x : \mathbb{N}, y : \mathbb{N} \vdash \text{ind}_{\mathbb{N}}(0, \text{s}\ z, y) : \mathbb{N}}}{x : \mathbb{N} \vdash \lambda y.\text{ind}_{\mathbb{N}}(0, \text{s}\ z, y) : \mathbb{N} \to \mathbb{N}}}{\vdash \lambda x.\lambda y.\text{ind}_{\mathbb{N}}(0, \text{s}\ z, y) : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})}$$

Note that we are being looser with notation than previously. For example we are not annotating the natural number type to be depending on terms, nor are we annotating the function arguments x and y, although we certainly could. Another more crucial example of this is that we omitted what would be the first assumption of our $\mathbb{N}$-**elimination** rule, which would be "$y : \mathbb{N} \vdash \mathbb{N}$ type". If we wanted to formally pattern match, we could see that adding this clause would allow for the left and right clauses at the top of the proof tree to match with the second and third clauses of the $\mathbb{N}$-**elimination** rule respectively. Because of this and the previous comments on ordering when discussing context, the ordering of x, y, z does indeed matter. We couldn't formally pattern match and then proceed with inference otherwise.

It may seem like an strange to leave this clause out for convenience and then explain it textually anyways, however it illustrates the idea that most formal systems, be it any sort of type theory or classical logic, are mostly used in

mathematics rigorously and informally. The amount of formality and level of detail varies greatly in what kind of work is being done and the audience it is being presented to. Likewise the notion of rigor is quite subjective as well. Thus it is somewhat independent of the formality of a proof, although a fully formal proof is certainly rigorous. But then what is a formal proof? Most type theory and related communities mean when they are talking about formal proofs is a machine checked proof or something of very similar nature. It is when the steps of inference are done so precisely, so carefully and laid in a specific way so that a machine could do them. Usually this is done in hand with a proof assistant/theorem prover, where one types in inference rules and each step is checked by the computer. It turns out that because of the underlying semantics and nature of type theory checking steps of a proof is done in a manner analogous to the type checking done in programming languages. So it while it is extremely useful and applicable to have a fully formal mechanization of type theory, it does not necessarily mean we have to talk about it fully formally or that it has to be practiced fully formally when we are doing it in "pen and paper". It is good to do so for rudimentary examples to get used to the ideas, however in a similar way we take arithmetic and algebra for granted as we move on from secondary school, as we advance in our understanding of type theory, we can omit details to be less formal, but we won't necessarily be less rigorous.

3.2 Coproducts

In this section and the next we will introduce two more inductive types to round out our set theoretic and propositional interpretation of types. First up are coproduct types, with the $+$-**formation** rule as follows:

$$\frac{\Gamma \vdash P \text{ type} \qquad \Gamma \vdash Q \text{ type}}{\Gamma \vdash P + Q \text{ type}}$$

The $+$-**introduction** rules:

$$\frac{\Gamma \vdash p : P \qquad \Gamma \vdash Q \text{ type}}{\Gamma \vdash \text{inl}(p) : P + Q}$$

$$\frac{\Gamma \vdash P \text{ type} \qquad \Gamma \vdash q : Q}{\Gamma \vdash \text{inr}(q) : P + Q}$$

Notice that we have mixed kinds of judgments as clauses, one which has a term p of type P, and another existence of a type Q and vicea versa. Again falling back on the idea of propositions as types, this makes it clear that coproducts are internalizing the notion of logical disjunction: an "or" connective . "inl" and "inr"" can be read as "in left" and "in right", respectively. For the set theoretic interpretation we can think of this coproduct type as a disjoint union. The reason for a disjoint union specifically, and not simply a union, is because of what we mentioned earlier about there being no notions of subtypes, in the way that there are subsets.

The $+$-**elimination** rule is as follows:

$$\frac{\begin{array}{c}\Gamma, x : P + Q \vdash D(x) \text{ type} \\ \Gamma, p : P \vdash a : D(\text{inl}(p)) \\ \Gamma, q : Q \vdash b : D(\text{inr}(q))\end{array}}{\Gamma, x : P + Q \vdash \text{ind}_{P+Q}(a, b, x) : D(x)}$$

Note that thinking this as exhaustive pattern matching is a also a good schema here: to provide a term for a type family, one must be able to provide a term for all types in the family. In this case, we need to provide proofs that we can derive $D(x)$ from both kinds of canonical terms of $P + Q$, which is why we have the middle two judgments. Unlike the natural numbers however, there is no recursive definition. This gives rise to the $+$-**computation** rule:

$$\frac{\begin{array}{c}\Gamma, x : P + Q \vdash D(x) \text{ type} \\ \Gamma, p : P \vdash a : D(\text{inl}(p)) \\ \Gamma, q : Q \vdash b : D(\text{inr}(q))\end{array}}{\begin{array}{c}\Gamma, p : P \vdash \text{ind}_{P+Q}(a, b, \text{inl}(p)) \doteq a : D(\text{inl}(p)) \\ \Gamma, q : Q \vdash \text{ind}_{P+Q}(a, b, \text{inr}(q) \doteq b : D(\text{inr}(q))\end{array}}$$

3.3 Dependent Pair Types

One of the motivations for product types is that we can currently encode a dependent type family of vectors, indexed by length, and other things of that nature and provide terms for it. However we cannot construct a term for a pi type over a dependent type family "IsPrime(n)" because it is simply no true for all numbers. What we need is something which corresponds to existential quantification, as opposed to universal quantification. Thus to do so we will have the following Σ-**formation** rule:

$$\frac{\Gamma, x : P \vdash Q(x) \text{ type}}{\Gamma \vdash \Sigma_{x:P} Q(x) \text{ type}}$$

Note that here the hypothesises for the formation rule here are the exact same for the pi types. This is because we can always postulate the existence of the type, but we can't necessarily prove it by providing a term. And of course to construct a term, we need a Σ-**introduction** rule:

$$\frac{\Gamma \vdash p : P \qquad \Gamma \vdash q : Q(p) \text{ type}}{\Gamma \vdash \text{pair}(p, q) : \Sigma_{x:P} Q(p)}$$

Notice here that the hypothesises differ for the introduction rule of pi types. Instead of indexing an entire type family, we can only construct a term for one type family, hence why the hypothesises are separate. To prove a type indexed by a Σ type, we of course need an Σ-**elimination** rule:

$$\frac{\begin{array}{c}\Gamma, x : \Sigma_{p:P} Q(y) \vdash D(x) \text{ type} \\ \Gamma, p : P, q : Q(p) \vdash a : D(\text{pair}(p, q))\end{array}}{\Gamma, x : \Sigma_{p:P} Q(y) \vdash \text{ind}_\Sigma(a, x) : D(x)}$$

Note p in the index of Σ is distinct from the p used in the second judgement. Aside from this inconvenience, this rule reads quite nicely. If we can prove $D(\text{pair}(p, q))$, for all p and q, then we have a proof of $D(x)$. And as always, we have a corresponding Σ-**computation** rule:

$$\frac{\begin{array}{c}\Gamma, x : \Sigma_{p:P} Q(y) \vdash D(x) \text{ type} \\ \Gamma, p : P, q : Q(p) \vdash a : D(\text{pair}(p, q))\end{array}}{\Gamma, p : P, q : Q(p) \vdash \text{ind}_{\Sigma}(a, \text{pair}(p, q)) \doteq a : D(x)}$$

4 Concluding Remarks

This concludes our brief introduction to the technical basics of type theory. There is a lot we haven't covered, as mentioned, and we want to summarize and motivate some of the ideas some more here. For example, we haven't covered how to encode logical negation (although the idea is relatively straightforward). A less trivial example of something we omitted is that if you think of propositions as questions, then by extension so are types. And a very natural question to ask is when are two proofs/terms of a type equal? Perhaps the most important inductive type is the identity type, which "asks the question" of whether or not two terms are equal. The elimination rule for which is often called path induction, which again allude to the idea that we can interpret types as spaces, and terms as points in space. This is one of the main high level ideas of homotopy type theory.

More foundationally, we briefly mentioned universes, by claiming we cannot yet prove $0 \neq 1$ in $\mathbb{N}$. This statement can be stated more precisely in the context of the identity type: we cannot yet derive a term for the proposition/type $0 \neq_{\mathbb{N}} 1$. we have avoided talking about universes. The formal details of which are quite technical, however the main idea can be summarized as: types with types as terms. Note that this does not contradict our earlier claim that there is no notion of subtypes in type theory, as one finds subsets in set theory. A subset is not the same as having sets as elements in a set. Despite this however, the

notion of universes take us away from the set theoretic interpretation of types. Of course this is not to say the intuitions we built are invalid, just that they are not exact correspondences once we are made aware of these new ideas.

More broadly, because we are working with foundations it also a very natural question to ask how this relates to areas such as computability theory or constructive, intuitionistic logic. It turns out the framework of type theory, although it does have a constructive flavor, can still consistently admit "truncated" forms of the law of excluded middle [*Awodey] and the axiom of choice [*Awodey]. Thus exploring the area further can also be a great entry point into learning the nuances and history of different areas of logic, and even if these terms are unfamiliar to the reader, we hope that mentioning these buzzwords can help one start the journey if they so desire.

As a follow up, if interested in pursuing more of type theory and its applications, I would encourage the reader to take a look at some of the sources I used for this semester, and will continue to use here, as advertised in the beginning. It's certainly a broad subject, and there are many ways one can take it. One can go into the more theoretical side exploring homotopy type theory, and related areas, or go into something more applied be it the use or design of proof assistants and programming languages. And I hope that however one may continue their journey with learning about this, has found this work useful as some sort of starting point or frame of reference.