1 History and Motivation

The development of formal logic and the foundations of computation theory trace a long historical arc, with key ideas emerging in stages to form the modern conception of propositions as types. We will examine several important developments in this history, focusing on how evolving approaches to logic and computability gradually laid a foudation for the correspondence between logical propositions and type theory. Formal logic saw its modern origins in the work of Gottlob Frege in the late 19th century. In his Begriffsschrift, published in 1879, Frege introduced the first fully symbolic system of logic, including quantified variables and a logical notation that would be recognisable to modern logicians. Importantly, Frege's work shifted logic from the study of the laws of thought to the study of abstract, formal systems. Logical propositions were now represented by formal symbolic expressions, manipulated according to explicit rules. This step began the distillation of logic to symbolic form, a proper leap in the ealy devlopment of the systems we use today. As Frege's work filtered through the mathematical world in the early 20th century, Bertrand Russell recognised its significance. Russell went on to develop his type theory as an attempt to resolve certain paradoxes that arose from Frege's naive set theory, such as Russell's paradox. In Russell's type theory, every term has a type, and types are arranged hierarchically. This stratification of objects into leveled types resolved the known paradoxes, at the cost of making the system much more complicated. Very importantly though, in Russell's system, propositions were considered to be of a logical type –the first step towards drawing together notions of logical proposition and type.

A major contribution to formal logic and the foundations of mathematics came in the 1930s through the work of Gerhard Gentzen. Gentzen introduced the key idea of analytic proof, where the structure of a proof mirrors and breaks down the structure of what is being proved. He developed the technique of natural deduction, where proofs are constructed through the successive application of inference rules that introduce or eliminate logical connectives. The resulting proofs have a clear structure that reflects the shape of the propositions being proved. This introduced a new degree of structural correspondence between proofs and propositions. In the 1960s and 70s, a cross-pollination occurred between formal logic and the budding field of computer science through the Curry-Howard correspondence (also known as the proofs-as-programs correspondence). Logician Haskell Curry and computer scientist William Howard independently observed a near-poetic analogy between natural deduction proofs and certain programming languages. Specifically, the inference rules of natural deduction were seen to correspond exactly to type inference rules for the simply typed lambda calculus. This incited a curiosity around the relation between logical proofs and computer programs, as well as between logical propositions and types in programming languages.

The Curry-Howard correspondence came to life in the work of Per Martin-Löf in the 1970s and 80s as he investigated intuitionistic type theory. Martin-Löf's work brought together the threads of Russell's type theory, Gentzen's natural deduction, Church's lambda calculus, and the Curry-Howard correspondence into a single unified framework. In Martin-Löf's system, propositions are directly equated with types. Any logical proposition can be seen as specifying a type, and a proof of that proposition corresponds to a term of that type. The structure of a proof matches the structure of the corresponding lambda term. The concept of propositions-as-types was now explicitly formulated and became known as the Curry-Howard correspondence. In logic, it constitutes a continuation and refinement of the structuralist approach initiated by Gentzen, where the prominent features

of logic arise from the structure of proofs. In computer science, it has led to a bloom of work on functional programming languages and proof assistants where programs and proofs exist on the same continuum. Languages like Lean, Coq, Agda, and Idris use highly expressive type systems to enforce program correctness and allow proving mathematical theorems. The correspondence between proofs and programs also reveals an exciting new perspective on the nature of computation itself. Through the lens of this correspondence, computational models can be seen as a dynamic expression of the structure of mathematical proofs – no longer are these concepts theoretically distinct or even partially related. The lambda calculus can be viewed not just as a system of computation, but as a way of capturing the essential structure of logical reasoning. The clean semantic foundation this offers for programming languages has enabled the development of whole new paradigms of coding, including the current growth of languages that incorporate dependent types. The story of propositions-as-types is the story of the progressive refinement and structuralisation of logic, and the parallel development of models of computation. As logic became more symbolic and structural through the work of early modern logicians such as Frege, Russell, and Gentzen, and as the foundations of computation were laid by figures like Church, Gödel, and Turing, the deep currents carrying both fields slowly brought them together. The tree they grafted was the Curry-Howard correspondence, and the fruit was a remarkable unification of logic and computation through the common language of type theory. Today, the ideas that began with Russell's and Gentzen's efforts to refine logic, and with Church's and Turing's models of computation, have developed into a profound paradigm for the structure of formal systems, with implications for mathematics, computer science, and our understanding of the nature of reasoning and computation.

2 Logical Foundations

2.1 Preliminaries

2.1.1 Formal Logic

Formal logic is the study of valid reasoning, developed through centuries of philosophical and mathematical inquiry. Beginning with Aristotle's syllogistic reasoning and advancing through contributions from mathematicians such as Frege, Russell, and Gödel, formal logic evolved from qualitative philosophical arguments into precise mathematical systems. This transformation was motivated by the need to establish secure foundations for mathematical reasoning, particularly as mathematicians encountered paradoxes in set theory and questions about the consistency of arithmetic in the late 19th and early 20th centuries. Formal logic provides a mathematical framework for analysing the structure of mathematical statements and the relationships between them, enabling rigorous verification of mathematical proofs. Through formalisation, logical arguments become mathematical objects that can be studied with mathematical precision, leading to important results about the capabilities and limitations of mathematical reasoning itself. We will consider a systematic approach to formal logic that breaks the study into two distinct components: elementary frameworks and logical principles. These components fulfill different roles while working together to create complete systems of mathematical reasoning.

Elementary frameworks establish what we can express in a logical language. The most basic framework, propositional logic, works with atomic statements and logical connectives ($\wedge$, $\vee$, $\rightarrow$, $\neg$). First-order predicate logic (FOL) adds quantification over individual variables, allowing ex-

pressions such as $\forall x(P(x) \rightarrow Q(x))$. Some statements that we can make with FOL include the commutativity of addition: $\forall x \forall y(x + y = y + x)$, existence of inverses: $\forall x \exists y(x + y = 0)$, and transitivity of order: $\forall x \forall y \forall z((x < y \wedge y < z) \rightarrow x < z)$. Second-order logic (SOL) introduces quantification over predicates, extending this expression to statements such as mathematical induction: $\forall P((P(0) \wedge \forall n(P(n) \rightarrow P(n+1))) \rightarrow \forall n P(n))$, completeness of real numbers: $\forall S((S \neq \emptyset \wedge S$ is bounded above$) \rightarrow S$ has a least upper bound), and finite sets: $\forall S(\exists n \exists f(f$ is a bijection from S to $1, \ldots, n))$. Higher-order logic (HOL) extends this to include quantification over functions of functions and properties of properties such as the continuity of functions: $\forall f \forall x \forall \epsilon > 0 \exists \delta > 0 \forall y(|x - y| < \delta \rightarrow |f(x) - f(y)| < \epsilon)$, compactness: $\forall \mathcal{F}(\forall \mathcal{G} \subseteq \mathcal{F}($if $\mathcal{G}$ is finite then $\bigcap \mathcal{G} \neq \emptyset) \rightarrow \bigcap \mathcal{F} \neq \emptyset)$, and functionals (a function that takes other functions as its input and returns an element of some field as its output): $\forall F((F$ is a linear functional on vector space $V) \rightarrow$ certain properties hold). So we see a distinct hierarchy of expressive power with FOL expressing properties of individual elements and their relationships, SOL adding the ability to quantify over sets and binary relations, and HOL enabling the expression of properties of arbitrary functions and properties. Logical principles, on the other hand, determine how we reason within any given framework. Minimal logic provides foundational rules for constructive implication. Intuitionistic logic builds on these rules while maintaining constructivity: each proof must demonstrate explicitly how to construct any object claimed to exist (hence also being known as constructive logic). Classical logic adds principles such as the law of excluded middle ($p \vee \neg p$), which enables non-constructive proof methods.

An important observation is that any framework can operate under any set of principles. This independence means that first-order logic works equally well with minimal, intuitionistic, or classical principles, though with different proof-theoretic strengths. Each combination creates a logical system with specific properties. This independence has various applications in mathematics and other tasks where reasoning is involved. Program verification often uses higher-order logic for expressiveness while applying intuitionistic principles to ensure constructive proofs. Abstract mathematics often uses first-order logic with classical principles to balance expressiveness with available proof techniques. The fameworks provide the *language* for expressing mathematical concepts, while principles establish the *methods for deriving truths* within that language.

2.2 Constructive Logic

Constructive logic provides avenues of reasoning where truth requires explicit demonstration through step-by-step proof. In constructive mathematics for example, proving that something exists requires actually showing how to construct it. This approach coincides with computer programming, where algorithms must provide explicit computational paths. When a constructive mathematical proof shows that $\exists x P(x)$, it provides an actual method to compute such an x, just as a program must compute specific values rather than merely assert their existence. The canonical example of the difference between the constructive vs non-constructive approach to a proof is that of the irrationality of $\sqrt{2}$. A constructive proof expresses this through direct algebraic reasoning. We begin by assuming $\sqrt{2}$ can be written as a fraction p/q where p and q are integers in lowest terms. From this assumption, we derive that $2q^2 = p^2$. Therefore p^2 is even, which means p must be even. We can write $p = 2k$ for some integer k. Substituting this back, we obtain $2q^2 = 4k^2$, hence $q^2 = 2k^2$. This shows q must also be even. However, this contradicts our assumption that p and q were in lowest terms. Through this sequence of explicit algebraic steps, we have *directly shown* that $\sqrt{2}$ cannot be rational.

In contrast, consider this non-constructive proof that there exist irrational numbers x and y such that x^y is rational. Consider the number $\sqrt{2}^{\sqrt{2}}$. This number must be either rational or irrational. If it is rational, then we have found our example: let $x = y = \sqrt{2}$. If instead $\sqrt{2}^{\sqrt{2}}$ is irrational, then let $x = \sqrt{2}^{\sqrt{2}}$ and $y = \sqrt{2}$. In this case, $x^y = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^2 = 2$, which is rational. This proof establishes existence by using the law of excluded middle, yet provides no way to determine which case actually holds. The constructive proof provides explicit calculations and a clear sequence of logical steps that demonstrate why the statement must be true. The non-constructive proof establishes existence through indirect reasoning using the law of excluded middle, without providing a method to identify specific examples. Such non-constructive methods, while mathematically valid under classical logic, do not provide algorithmic paths to finding solutions, and hence are rejected by intuitionistic logic.

Constructive Semantics The Brouwer-Heyting-Kolmogorov (BHK) interpretation establishes semantic meaning for intuitionistic logic by connecting logical formulas with constructive mathematical proofs. The interpretation originated from L.E.J. Brouwer's early 20th century development of intuitionism, which viewed mathematics as a creation of the human mind rather than a discovery of abstract truths. Arend Heyting subsequently formalised Brouwer's ideas by providing precise proof-theoretic semantics for intuitionistic logic in the 1930s. Andrei Kolmogorov independently developed similar ideas on the interpretation of logical connectives through problem-solving operations, leading to what we now recognize as the BHK interpretation's unified approach to constructive mathematics and logic. This interpretation defines truth through the presence of constructive proof: a proposition is considered true precisely when we possess a constructive demonstration of its validity. The interpretation provides specific meaning to each propositional logic connective through proof requirements. For conjunction $P \wedge Q$, truth requires both a proof of P and a proof of Q. A disjunction $P \vee Q$ demands either a proof of P or a proof of Q, accompanied by an explicit indication of which has been proven. An implication $P \to Q$ requires a construction that transforms any proof of P into a proof of Q. Universal quantification $\forall x P(x)$ necessitates a construction yielding a proof of $P(a)$ for any given element a. Existential quantification $\exists x P(x)$ requires both a specific element a and a proof that $P(a)$ holds. The proposition $\bot$ representing falsehood has no proof, while negation $\neg P$ represents a construction transforming any proof of P into a proof of $\bot$. This interpretation explains why certain classical principles fail in intuitionistic logic. Consider the law of excluded middle, $P \vee \neg P$. Under the BHK interpretation, proving this would require either a direct proof of P or a proof that P leads to contradiction. Since we cannot guarantee having either for arbitrary propositions P, this principle does not hold generally in intuitionistic logic. Similarly, double negation elimination ($\neg\neg P \to P$) fails because proving $\neg\neg P$ shows only that assuming P has no proof leads to contradiction, without constructing a direct proof of P.

The BHK interpretation connects logical propositions directly to computational processes by defining truth through constructive proof methods. Under this interpretation, logical connectives map naturally to programming constructs: a proof of $P \wedge Q$ requires both a proof of P and Q, similar to how a program must compute both components of a pair; a proof of $P \to Q$ provides a method to transform proofs of P into proofs of Q, reflecting how functions transform inputs to outputs; a proof of $\exists x P(x)$ must provide both a specific value and evidence that it satisfies P, just as programs must compute explicit values rather than merely assert their existence

2.2.1 Models of Computation

The idea of computability as we know it today dawned around the early twentieth century, when mathematicians sought to formalise the notion of what was then descrbed as 'effective calculability'. David Hilbert's program, which aimed to mechanise mathematical reasoning, motivated this investigation. Hilbert proposed finding a systematic method to determine the truth of any mathematical statement through purely mechanical means. Three models of what came to be known as 'computation' emerged during the 1930s, each offering distinct perspectives on mechanical calculation. Alan Turing introduced his automatic machines, now known as Turing machines, which model computation through the actions of a hypothetical device manipulating symbols on an infinite tape. This model focuses on the sequential nature of computation and the role of memory in calculation. A Turing machine operates by reading symbols, changing its internal state, and writing new symbols according to a fixed set of rules. This model provided an intuitive foundation for understanding computational processes. Alonzo Church simultaneously developed the lambda calculus, which offered an alternative characterisation of computation based on function application and variable binding. The lambda calculus represents computation through the systematic transformation of symbolic expressions, demonstrating how complex calculations can arise from simple substitution rules. This model was more abstract and has been since particularly relevant to the development of functional programming languages and type theory, as it presents computation in terms of mathematical functions rather than machine operations. The theory of recursive functions, developed in large part by Kurt Gödel and Stephen Kleene, defines computation through mathematical function composition, recursion, and minimisation. These three models—Turing machines, lambda calculus, and recursive functions—were subsequently proven equivalent in computational power, leading to the Church-Turing thesis by Kleene who was one of Church's students. This thesis proposes that these formal systems capture the intuitive notion of what it means for something to be 'effectively calculable' –any function that can be calculated through a step-by-step procedure can be computed by a Turing machine, expressed in the lambda calculus, or general recursive functions. Computational models that fit into this family of expression have come to be known as 'universal models of computation'.

The investigation of these computational models revealed important limitations of mechanical calculation. Turing's proof of the halting problem's undecidability demonstrated that no algorithm can determine whether an arbitrary program will terminate. This result, together with Gödel's incompleteness theorems, established fundamental boundaries on what mechanical procedures can achieve. These limitations carry significant implications for automated theorem proving, as they identify classes of mathematical problems that resist purely mechanical solution. The connection between computation and logic gained additional depth through the Curry-Howard correspondence, which reveals a one-to-one relationship between computer programs and mathematical proofs. This correspondence shows that types correspond to propositions, while programs correspond to proofs. Modern theorem provers build on this insight, using type theory to verify mathematical arguments mechanically. The lambda calculus plays an important part in this system, serving as a bridge between computational and logical perspectives.

Type Theory The development of type theory originated from a crisis point in the foundations of mathematics at the turn of the 20th century. It centred on the discovery of paradoxes in naive set theory, particularly Russell's paradox, which exposed inconsistencies in the formal systems math-

ematicians had relied on. Bertrand Russell introduced the first type theory in 1908 as a response to these paradoxes. His theory proposed a hierarchical organisation of mathematical objects, where sets were assigned different 'types' based on their complexity. This hierarchical structure prevented the self-reference that led to paradoxes by ensuring that a set could only contain elements of strictly lower types. While Russell's solution proved effective at avoiding paradoxes, its complexity made it impractical for everyday mathematical work. The next significant advancement came through Alonzo Church's work in the 1930s. Church's development of the simply typed lambda calculus combined Russell's insights about types with his own work on computational functions. This system provided a formal foundation for studying functions and their properties while maintaining logical consistency. Church's work demonstrated that types could serve not only to prevent paradoxes but also to structure mathematical reasoning about computation. Per Martin-Löf contributed to this in the 1970s with his intuitionistic type theory, motivated to unify constructive mathematics, computation, and formal logic. Martin-Löf's system introduced dependent types, allowing types to depend on terms, which significantly increased the expressive power of type theory. This advancement enabled precise specifications of mathematical structures and properties within the type system itself.

2.2.2 Formal Proofs

1. The difference between a pen and paper proof and a formally verified proof. Thinking about what it means to be effectively calculable (or computable) and then the means by which we evaluate that thing (?).

2.3 Natural Deduction

2.3.1 Fundamental Principles

Natural Deduction is a formal system in mathematical logic that models the process of reasoning through proofs, developed independently by Gerhard Gentzen and Stanisław Jaśkowski in the 1930s. We will refer to the Gentzen-style Natural Deduction system, though the Jaśkowski approach retains a familiar air. The system is designed to reflect the intuitive steps one might take when reasoning informally, making it a 'natural' framework for logical deduction. From a technical mathematics perspective, the fundamental principles of Natural Deduction can be understood through its structure, rules, and goals. Natural Deduction operates on the principle that every proof begins with a finite set of hypotheses and aims to derive a conclusion. The system is constructed around inference rules that govern how logical connectives (such as $\wedge$, $\vee$, $\rightarrow$, $\neg$) and quantifiers ($\forall$, $\exists$) can be introduced or eliminated in proofs. These rules are divided into two categories: introduction rules, which define how to construct a statement involving a specific connective or quantifier, and elimination rules, which describe how to deduce consequences from such statements. For example, the introduction rule for conjunction ($\wedge$) allows one to infer $A \wedge B$ if both A and B are true, while the elimination rule for conjunction allows one to infer either A or B from $A \wedge B$. Subproofs can be made, by temporarily assuming additional premises (hypotheses) to explore their logical consequences. Once the desired conclusion is reached within the subproof, the temporary assumption is discharged, and the conclusion can be used in the broader proof without relying on the temporary premise. This mechanism is particularly evident in rules like implication introduction ($\rightarrow$ Intro), where assuming A and deriving B within a subproof allows one to conclude $A \rightarrow B$.

Local soundness in the Natural Deduction system ensures that if an introduction rule is immediately followed by its corresponding elimination rule, the result is logically equivalent to bypassing the intermediate step entirely. This guarantees that no invalid conclusions can be derived. Local completeness ensures that elimination rules retain enough information to reconstruct the original statement using an introduction rule, ensuring that all valid conclusions can be derived. One could almost think of this as the introduction and elimination rule for a given expression as being inverse actions to each other. The rules are all orthogonality defined, in that each action available to a given logical connective or quantifier is defined independently of others. Direct proofs proceed in Natural Deduction by deriving conclusions step-by-step from premises using inference rules, while indirect proofs often involve assuming the negation of a statement and deriving a contradiction (reductio ad absurdum). One has the flexibility to proceed with proofs from both propositional and predicate logic, as well as freedom to choose logics from minimal, intuitionistic, and classical frameworks.

2.3.2 Rules of Inference

The logical framework of Natural Deduction can be described by it's rules of inference, which provide the mechanisms through which valid conclusions can be derived from premises. The rules are categorised into two main types: introduction rules and elimination rules, and are written in the form:

$$\frac{\text{Premises}}{\text{Conclusion}} \quad \text{(Rule-Name)}$$

Gentzen described the introduction rules as the 'definitions' of the symbols they represent, in that they specify the grounds under which we may interact with them. Introduction rules allow for the construction of arbitrarily complex logical statements from simpler ones. For instance, conjunction introduction enables two separately true statements to be combined into a single conjoined statement. These rules essentially 'build up', or 'introduce' logical complexity by establishing new connections between already proven (or assumed) statements. Elimination rules, conversely, permit the extraction of simpler statements from arbitrarily complex ones. Gentzen described these as 'consequences' of the given definitions (introduction rules). These rules enable the deconstruction of compound logical statements into their constituent parts, allowing for the derivation of new conclusions from established premises. For example, conjunction elimination allows the derivation of individual conjuncts from a conjunction. Natural Deduction includes rules for handling the logical operators from both propositional and predicate logic, including implication ($\rightarrow$), disjunction ($\vee$), negation ($\neg$), and quantification ($\forall$ and $\exists$).

The introduction rule for conjunction allows the formation of a compound statement $A \wedge B$ from two individual statements A and B. This rule reflects the intuitive notion that if both A and B are true, then their conjunction must also be true.

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge \mathrm{I}$$

Conversely, the elimination rules for conjunction permit the extraction of either component A or B from the compound statement $A \wedge B$.

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge \mathrm{E}_1 \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge \mathrm{E}_2$$

The introduction rule for disjunction allows one to infer $A \vee B$ from either A or B alone, holding the idea that if at least one of the statements is true, the disjunction holds.

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee \mathrm{I}_1 \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee \mathrm{I}_2$$

The elimination rule for disjunction, often referred to as disjunctive syllogism, is more intricate. If we have that $A \vee B$ is true, then we can split our proof into two cases: One where A is true and another where B is true, and both of these propositions individually conclude to some propoisiton C. From this, we can confirm that C must be true, since both A and B are given and lead to the same conclusion of C.

$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} \vee \mathrm{E}$$

The introduction rule for implication is based on the principle of conditional proof. To introduce $A \to B$, one assumes A as a temporary hypothesis and demonstrates that B necessarily follows from this assumption. This method captures the essence of "if-then" statements, ensuring that the implication holds under the assumption of its antecedent.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to \mathrm{I}$$

The elimination rule for implication, commonly known as modus ponens, allows one to derive B directly from A and $A \to B$.

$$\frac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B} \mathrm{MP}$$

The introduction rule for negation involves assuming a proposition A and deriving a contradiction from this assumption, thereby inferring $\neg A$ (which is actually $A \to \bot$ in disguise!). This resonates with the idea that if assuming A leads to an impossible outcome, then A must be false.

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A} \neg \mathrm{I}$$

The elimination rule for negation permits the derivation of any statement C from both A and $\neg A$, giving us the principle of explosion, meaning that contradictory premises can lead to any conclusion in both intuitionistic and classical logic.

$$\frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot} \neg \mathrm{E}$$

The structural rules of the Natural Deduction system describe how the proof derivation must be set up, analogous to the regulations around timing and seating placement for a chess match. These rules include weakening (as shown in Figure 1 below), which allows the introduction of irrelevant hypotheses without affecting the validity of the proof; contraction, which permits the duplication or reuse of hypotheses; and exchange, which ensures that the order of hypotheses does not influence the derivation process. Natural Deduction as a formal system of logic is sound, which guarantees that all derivable statements are logically valid –so they hold true under all interpretations. It is also complete, which ensures that if a statement is logically valid, it can be derived within the system.
Extending minimal logic to intuitionistic logic requires an additional structure, namely that of Ex

$$\dfrac{\dfrac{[A]^1}{B \to A} \to I, 2}{A \to (B \to A)} \to I, 1$$

Figure 1: Proof of $\vdash A \to (B \to A)$ (Weakening)

Falso Quodlibet (the principle of explosion). From this law, the derivation of a contradiction bears particularly nasty consequences –the lines of truth and absurdity are blurred and from this context we may draw *any* conclusion. Figure 2 below demonstrates the proposition A leading to $\bot$ and the resulting derivation of any proposition (in this case B):

$$\dfrac{\dfrac{[A]^1}{\vdots}}{\dfrac{\bot}{B}} \text{XF}$$

Figure 2: Ex Falso Quodlibet (XF)

Extending further still from intuitionistic logic to classical logic, we introduce the argument of Reductio Ad Absurdum (RAA). The structure introduced by this assertion gives us that if assuming the negation of a claim leads to a contradiction with what can be derived from the claim itself, then the original claim must hold. This derives from the classical reasoning that a proposition must be true if its negation is impossible. We see a general derivation of RAA in Figure 3 below:

$$\dfrac{\dfrac{[A]^1}{\vdots}}{\dfrac{B}{\dfrac{\bot}{A}}} \quad \dfrac{[\neg A]^2}{\dfrac{\vdots}{\neg B}} \; RAA, 2$$

Figure 3: Reductio Ad Absurdum (RAA)

The prooftree above demonstrates RAA by showing that assuming A produces some claim B, while assuming $\neg A$ leads to $\neg B$. The contradiction between B and $\neg B$ establishes $\bot$, allowing us to conclude A through RAA. Using RAA, we can also prove Double Negation Elimination (DNE) as shown in Figure 3 below. With $\neg\neg A$ we assume $\neg A$ to arrive at a contradition, then by RAA we get A. Conversely, DNE allows us to prove RAA. if $\neg A$ leads to $\bot$, then we have $\neg\neg A$, and by DNE we get A.

2.3.3 Proof Construction

Proofs in Natural Deduction are carried out in the style of a proof tree. In these representations, each node corresponds to a formula derived from its parent node(s) according to preceeding inference rules. The root of the tree represents the conclusion, while the leaves represent either initial

$$\dfrac{\dfrac{\neg\neg A \qquad [\neg A]^1}{\bot}\,\bot E}{A}\,RAA, 1$$

Figure 4: Connection between DNE and RAA

hypotheses or assumptions introduced in subproofs. Take for example, a minimal logic proof of the sequent $P \to Q,\ Q \to R \vdash P \to R$ shown in Figure 5 below:

$$\dfrac{\dfrac{[P]^1 \qquad P \to Q}{Q}\,\to E \qquad Q \to R}{R}\,\text{MP}$$
$$\dfrac{R}{P \to R}\,\to I, 1$$

Figure 5: Proof of $P \to Q,\ Q \to R \vdash P \to R$. (Function Composition)

We see that the proof establishes the trasitivity of implication through a combination of implication elimination and introduction rules that follow in a tree-like structure. We see the ability to make temporary assumptions (here shown as $[P]^1$), and discharge them to prove implications. The proof proceeds linearly from the premises and assumption to the conclusion, using only the rules of inference allowed by minimal logic. The proof in Figure 6 below demonstrates the use of case analsis in a Natural Deduction proof through the interaction between disjunction elimination and implication rules in minimal logic. Here the proof must handle a premise by considering each case separately, deriving the same conclusion in each case, and then using an elimination rule to establish that conclusion independently of the particular disjunct.

$$\dfrac{P \vee Q \qquad \dfrac{\dfrac{\dfrac{[P]^1 \qquad P \to R}{R}\,\text{MP}}{R \vee D}\,\vee I_1}{P \to (R \vee D)}\,\to I, 1 \qquad \dfrac{\dfrac{\dfrac{[Q]^2 \qquad Q \to D}{D}\,\text{MP}}{R \vee D}\,\vee I_2}{Q \to (R \vee D)}\,\to I, 2}{R \vee D}\,\vee E$$

Figure 6: Proof of $P \vee Q,\ \ P \to R,\ \ Q \to D \vdash R \vee D$

The process of constructing proofs in the Natural Deduction system combines both forward and backward reasoning strategies. Proof construction typically begins with an analysis of both the given premises and the desired conclusion, working to bridge the gap between them through valid logical steps. When constructing proofs, one typically employs a strategy of working backward from the desired conclusion while simultaneously working forward from the given premises. This bidirectional approach helps identify the necessary intermediate steps and guides the selection of appropriate inference rules. The process often involves creating sub-proofs, managing assumptions, and carefully tracking logical dependencies.

An important aspect of proof construction in this system is to be mindful of which assumptions have been made and which are yet to be discharged. Temporary assumptions can be introduced when

needed, but must be properly discharged when their use is complete in order to preserve consistency. The deduction theorem establishes that if a proposition B is derivable from a context γ together with proposition A, then $A \to B$ is derivable from γ alone such that:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to I$$

The converse also holds through modus ponens:

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B} \to E$$

Together, these rules establish that $\Gamma \cup \{A\} \vdash B$ if and only if $\Gamma \vdash A \to B$. In practice, the deduction theorem tells us that we may make assumptions of the antecedent of an implication on the right hand side of a turnstile if we proceed to discharge it through implication introduction throughout the proof. It requires the logician to maintain proper tracking over the scope of assumptions and ensure that all dependencies are properly taken care of throughout the proof. The deduction theorem in action can be demonstrated in the sequent proof $\vdash (P \to Q) \to (\neg Q \to \neg P)$ shown in Figure 7 below. By the deduction theorem, we may manipulate the sequent so that as many hypotheses as possible are on the left of the turnstile. The proof of this new sequent, which we write as $P \to Q, \neg Q, P \vdash \bot$, will be a proof of the original sequent after extending it with implication introductions to justify each temporary hypothesis.

$$\dfrac{\dfrac{\dfrac{\dfrac{[P \to Q]^1 \qquad [Q]^2}{Q}\,\text{MP} \qquad [\neg Q]^3}{\bot}\,\text{MP}}{\neg P} \to I, 2}{\neg Q \to \neg P} \to I, 3}{(P \to Q) \to (\neg Q \to \neg P)} \to I, 1$$

Figure 7: Proof of the sequent $\vdash (P \to Q) \to (\neg Q \to \neg P)$

Classical reasoning through the law of excluded middle is demonstrated in Figure 8 below. Here, we assume $\neg P$ and derive a contradiction to prove P. The Ex Falso principle gives us two structural components to the proof –constructing $P \to Q$ from contradictory premises, and deriving the final contradiction needed for negation elimination.

Various techniques exist for handling different types of logical statements, not so dissimilar to sequences of moves in chess. For example, in conjunctive conclusions, one must prove each conjunct separately before combining them. Recognition of this encourages the logician to visualise the playing field with the steps necessary to perform this action, and any required temporary hypotheses in the overall map. For conditional statements, one assumes the antecedent and works to prove the consequent. Disjunctive reasoning involves case analysis, where different possibilities are explored separately. As we will see in the next chapter, Lean can be thought of as 'gamifying' a problem — requiring resolution in a series of 'goals'. Proof construction in the ND system is at the heart of this interwoven web of goals, with many smaller proofs combining to create a larger one.

$$\dfrac{\dfrac{\dfrac{[\neg(P \to Q)]^1 \qquad \dfrac{\dfrac{[\neg P]^2 \qquad [P]^3}{\bot}\,\text{XF}}{Q}\,\neg E,2}{P \to Q}\,\to I,3}{P}\,\text{XF}}{\neg(P \to Q) \to P}\,\to I,1$$

Figure 8: Proof of the sequent $\neg(P \to Q) \to P$

2.3.4 Computational Properties

The ND system possesses several important theoretical properties that establish its validity and utility as a logical system. The most fundamental of these properties are soundness and completeness, which together ensure the system's reliability and comprehensive nature. Soundness guarantees that any conclusion derived through the rules of ND is logically valid. This property tells us that if a conclusion is provable from certain premises using ND rules, then that conclusion is indeed a logical consequence of those premises. Soundness is essential for maintaining the system's integrity and ensuring that proofs constructed within it are trustworthy. Completeness, on the other hand, establishes that all valid logical consequences can be proved within the system. This means that if a conclusion is a logical consequence of certain premises, then there exists a ND proof of that conclusion from those premises. The completeness property ensures that the system is powerful enough to capture all valid logical relationships. ND supports normalisation, which allows for the elimination of detours in proofs, leading to more direct and elegant derivations. Furthermore, the sub-formula property states that any formula appearing in a normal proof is a sub-formula of either the conclusion or one of the premises. This property is particularly valuable for proof search and automated theorem proving.

2.4 The Untyped Lambda Calculus

2.4.1 Definition and Syntax

The Untyped Lambda Calculus, often written λ-calculus, is a formal system developed by Alonzo Church in 1936 as a model of computation based on function abstraction and application. The λ-calculus consists of a single transformation rule, variable substitution, and function definition scheme. The λ-calculus is a universal model of computation, capable of expressing anything that a Turing machine can express. The basic definition of the λ-calculus revolves around the notion of terms –which can be variables, function abstractions, or function applications. Let $\mathcal{V}$ be a countably infinite set of variables, denoted by x, y, z, etc.. The set of λ-terms, Λ, can be defined inductively:

$$\begin{array}{lll}
\Lambda ::= x & \text{where } x \in \mathcal{V} & \text{(variables are terms)} \\
\quad \mid (\lambda x.M) & \text{where } x \in \mathcal{V}, M \in \Lambda & \text{(abstractions are terms)} \\
\quad \mid (M\ N) & \text{where } M, N \in \Lambda & \text{(applications are terms)}
\end{array}$$

Variables Variables in the λ-calculus are atomic entities used to represent arbitrary values or placeholders within a term. They are the simplest form of λ-terms and can be freely used within

abstractions and applications. The main role of variables is to allow the construction of generic functions that can accept arguments and enable substitution.

Abstractions Also known as anonymous functions, abstractions are used to define functions in the λ-calculus and consists of two parts: a bound variable, and a term (the function body). In the abstraction $(\lambda x.M)$, the variable x is the parameter bound in the term M. The λ symbol is used to denote the creation of an anonymous function, and the dot (.) separates the bound variable from the function body. Abstractions allow the definition of functions without explicitly naming them, hence the label of 'anonymous functions', and enable the creation of higher-order functions as they can take other functions as arguments or return functions as results.

Appplications Also known as function applications, applications represent the act of applying a function to an argument. The application $(M\ N)$ represents the application of the function M to the argument N. When an application is evaluated (reduced), the bound variable in the abstraction M is substituted with the argument N in the function body. Applications allow the execution of functions and the propagation of values throughout the term and multiple applications can be nested to represent the sequential application of functions to their respective arguments.

2.4.2 Term Construction

Construction of terms in the λ-calculus follows a set of inductive rules based on the syntax of the formal system that define how variables, abstractions, and applications can be combined to form valid λ-terms. The set of free variables in a term, denoted as $FV(M)$, consists of all variables that are not bound by an enclosing abstraction. In the case of a variable term x, the set of free variables is simply x. For an abstraction $(\lambda x.M)$, the set of free variables is determined by removing the bound variable x from the free variables of the subterm M. In an application $(M\ N)$, the free variables are the union of the free variables of both the function term M and the argument term N. The set of bound variables, denoted as $BV(M)$, follows a similar pattern: for a variable term, the set is empty; for an abstraction, the bound variable is added to the set of bound variables in the subterm; and for an application, the bound variables are the union of those in the function and argument terms. We have then, that the set of free variables of a term M can be configured in the following ways:

$$\begin{aligned} FV(x) &::= \{x\}, \text{where } x \in \mathcal{V} \\ FV(\lambda x.M) &::= FV(M) \setminus \{x\} \\ FV((M\ N)) &::= FV(M) \cup FV(N) \end{aligned}$$

and the set of bound variables of a term M, is defined as:

$$\begin{aligned} BV(x) &::= \emptyset, \text{where } x \in \mathcal{V} \\ BV(\lambda x.M) &::= BV(M) \cup \{x\} \\ BV((M\ N)) &::= BV(M) \cup BV(N) \end{aligned}$$

2.4.3 Variable Binding

A variable x is bound in a term M if it occurs within the scope of an abstraction $\lambda x.N$, where N is a subterm of M. A variable that is not bound is considered free. The capture-avoiding substitution

of a term N for a variable x in a term M, denoted $M[N/x]$, is defined inductively:

$$
\begin{aligned}
M[N/x] ::= {} & N && \text{where the term is } x \\
| {} & y && \text{where } y \neq x \text{ and the term is } y \\
| {} & \lambda x.M && \text{where the term is } \lambda x.M \\
| {} & \lambda y.(M[N/x]) && \text{where } y \neq x \text{ and } y \notin FV(N) \text{ and the term is } \lambda y.M \\
| {} & (M_1[N/x]\ M_2[N/x]) && \text{where the term is } (M_1\ M_2)
\end{aligned}
$$

Variable binding connects variables to their scopes through λ abstractions. In $\lambda x.M$, x becomes bound in M, while variables outside any binding λ remain free. The key operation is capture-avoiding substitution $M[N/x]$, which replaces bound variables with terms while preventing free variables in N from becoming accidentally bound in M.

2.4.4 Operations

Alpha Conversion Alpha conversion allows renaming bound variables in a term consistently. Two terms that differ only in the names of bound variables are considered α-equivalent. Formally, α-equivalence is defined as the smallest congruence relation $=_\alpha$ on Λ such that:

$$
\begin{aligned}
\alpha\text{-conversion} ::= {} & \lambda x.M =_\alpha \lambda y.M[y/x] && \text{where } y \notin FV(M) \\
& M_1 =_\alpha M_2 \implies M_2 =_\alpha M_1 && \text{(symmetry)} \\
& M_1 =_\alpha M_2, M_2 =_\alpha M_3 \implies M_1 =_\alpha M_3 && \text{(transitivity)} \\
& M_1 =_\alpha M_2 \implies (M_1\ N) =_\alpha (M_2\ N) && \text{(compatibility)}
\end{aligned}
$$

Alpha conversion defines how bound variables can be systematically renamed while preserving a term's meaning. The relation $=_\alpha$ establishes that two terms are equivalent if they differ only in their bound variable names, formalised through a congruence relation that is symmetric, transitive, and compatible with term construction. We have that $\lambda x.M$ is alpha-equivalent to $\lambda y.M[y/x]$ when y is not free in M, ensuring that renaming maintains the original binding structure without creating unintended variable captures. Alpha conversion preserves the meaning of a term while avoiding name clashes.

Beta Reduction Beta reduction is the process of applying a function to an argument, replacing the bound variable in the function's body with the argument. The β-reduction relation, denoted $\to_\beta$, is defined as the smallest relation on Λ satisfying:

$$
\begin{aligned}
\beta\text{-reduction} ::= {} & (\lambda x.M)\ N \to_\beta M[N/x] && \text{(application)} \\
& M \to_\beta M' \implies (M\ N) \to_\beta (M'\ N) && \text{(left reduction)} \\
& N \to_\beta N' \implies (M\ N) \to_\beta (M\ N') && \text{(right reduction)} \\
& M \to_\beta M' \implies \lambda x.M \to_\beta \lambda x.M' && \text{(abstraction)}
\end{aligned}
$$

The process of β-reduction substitutes an argument N for the bound variable x in a function body M when applying $(\lambda x.M)$ to N. Additional rules ensure reduction can occur within subterms: both the function and argument parts of an application can be reduced independently, and reduction can proceed under lambda abstractions.

Normal Forms A term is in normal form if it cannot be further reduced using β-reduction. The process of repeatedly applying β-reduction until no more reductions are possible is called normalisation. A term may have multiple normal forms or no normal form at all. The Church-Rosser theorem states that if a term can be reduced to two different terms, then there exists a term to which both can be reduced. This theorem implies the uniqueness of normal forms, when they exist. The existence of normal forms for all terms is not guaranteed in the Untyped Lambda Calculus. Some terms, such as $(\lambda x.x\ x)\ (\lambda x.x\ x)$, known as the ω-combinator, do not have a normal form and lead to infinite reductions.

2.4.5 Examples

INSERT EXAMPLES OF PROOF TREES HERE

2.4.6 Computational Properties

The untyped lambda calculus is Turing-complete, and reduction is confluent (satisfying the Church-Rosser property). It supports both call-by-value and call-by-name evaluation strategies, with call-by-name being normalising for more terms but potentially less efficient in practical implementations. The calculus can encode structures such as the natural numbers, booleans, pairs, and recursion through pure lambda terms (Church encodings), despite lacking primitive data types or explicit recursion mechanisms.

2.5 The Simply Typed Lambda Calculus

2.5.1 Definition and Syntax

The Simply Typed Lambda Calculus extends the untyped lambda calculus by introducing a formal type system alongside terms. The syntax consists of two inductively defined expressions –types and terms:

$$\text{Types} \quad \tau ::= \mathsf{unit} \mid \tau_1 \to \tau_2$$

$$\text{Terms} \quad e ::= () \mid x \mid e_1\ e_2 \mid \lambda x : \tau.e$$

Types Type construction in the STLC starts from a base type unit and builds up with function types. For example, a base type $\mathbb{Z}$ may extend inductively to produce a function type $\mathbb{Z} \to \mathbb{N}$, a function which take inputs of type $\mathbb{Z}$ and produces outputs of type $mathbbN$. Additionally, we have that if τ_1 and τ_2 are valid types, then the function type $\tau_1 \to \tau_2$ is also a valild type. This recursive definition allows for the construction of arbitrarily complex types from simpler ones. For example, Bool $\to$ (Bool $\to$ Bool) is a valid type that represents a function taking a Boolean input and returning another function from Booleans to Booleans. Function types associate to the right, meaning $\tau_1 \to \tau_2 \to \tau_3$ is interpreted as $\tau_1 \to (\tau_2 \to \tau_3)$.

Terms Terms in STLC include the unit value (), variables x, function applications $e_1\ e_2$, and lambda abstractions $\lambda x : \tau.e$. The unit value () serves as a primitive term of type unit. Variables represent placeholders that can be bound by lambda abstractions. Function application $e_1\ e_2$ applies the function e_1 to argument e_2. Lambda abstractions $\lambda x : \tau.e$ represent functions where x is a variable of type τ, and e is the function body. Function application remains left-associative,

complementing the right-associativity of function types. Given a function f of type $\tau_1 \to (\tau_2 \to \tau_3)$ and values v_1, v_2 of types τ_1, τ_2 respectively, the expression $f\ v_1\ v_2$ is parsed as $(f\ v_1)\ v_2$ and yields a result of type τ_3.

2.5.2 Operational Semantics

The operational semantics of STLC defines how terms evaluate through reduction rules. We first define values, which represent fully evaluated terms:

$$\text{Values} \quad v ::= () \mid \lambda x : \tau.e$$

Small-Step Semantics These operational semantics define program behaviour through atomic reduction steps. They specify transition between program configurations under three primary reduction relations and specify how terms are evaluated during computation:

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \quad \text{(E-App1)}$$

$$\frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'} \quad \text{(E-App2)}$$

$$(\lambda x : \tau.e_1)\ v_2 \to [x \mapsto v_2]e_1 \quad \text{(E-AppAbs)}$$

Here, E-App1 handles the reduction of function expressions in applications and E-App2 manages argument reduction when the function is a value. The rule E-AppAbs performs β-reduction by substituting values for variables in λ-abstractions with capture-avoiding substitution of value v for variable x in term e. These rules operate within evaluation contexts that establish a strict left-to-right evaluation order, ensuring both determinism (each term reduces to at most one other term) and compositionality (reductions can be applied consistently within larger expressions):

$$\text{Determinism:} \quad e \to e_1 \land e \to e_2 \implies e_1 = e_2$$

$$\text{Compositionality:} \quad \frac{e \to e'}{E[e] \to E[e']}$$

Call-by-value reduction strategy evaluates the function before its argument, and arguments are evaluated to values before substitution occurs. We define the multi-step reduction relation $\to^*$ as the reflexive, transitive closure of $\to$:

$$e \to^* e \quad \text{(reflexivity)}$$

$$\frac{e_1 \to e_2 \quad e_2 \to^* e_3}{e_1 \to^* e_3} \quad \text{(transitivity)}$$

Typing Judgements Big-step operational semantics (also called natural semantics) directly relate terms to their final values using evaluation judgments. A typing judgment is an assertion that determines how types are assigned to values. These rules are derived using inference rules which can be written in the familar Gentzen-style Natural Deduction form. First, we introduce the variable

rule:

$$\frac{}{\Gamma, x : T \vdash x : T} \text{ (T-VAR)}$$

This rule establishes that a variable x has type T when it appears in the typing context Γ with that same type. Building on the variable rule, the abstraction rule formalises how function types are constructed:

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash (\lambda x : T_1.t) : T_1 \to T_2} \text{ (T-ABS)}$$

This rule states that if a term t has type T_2 under the assumption that x has type T_1, then the lambda abstraction $\lambda x : T_1.t$ has the function type $T_1 \to T_2$. This typing rule corresponds directly to the intuitive notion of function types, where the type reflects both the input and output types of the function. The application rule tells us how function application works under the typing system:

$$\frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash (t_1\ t_2) : T_2} \text{ (T-APP)}$$

This rule says that when applying a function t_1 of type $T_1 \to T_2$ to an argument t_2 of type T_1, the resulting application has type T_2. The application rule ensures type safety by verifying that functions are only applied to arguments of the appropriate type.

2.5.3 Reduction Rules

The reduction rules of the STLC define how terms evaluate. The primary reduction rule is β-reduction:

$$(\lambda x : T.t_1)t_2 \quad \to_\beta \quad [x := t_2]t_1$$

This rule specifies that applying a function to an argument results in substituting the argument for all free occurrences of the bound variable in the function body. Complementing β-reduction is α-conversion:

$$\lambda x : T.t \quad \equiv_\alpha \quad \lambda y : T.[x := y]t \quad \text{where } y \notin \text{FV}(t)$$

Alpha conversion establishes that bound variables can be renamed without changing the meaning of the term, provided the new variable name doesn't conflict with existing free variables.

2.5.4 Type System Properties

Subject reduction, or type preservation, is expressed as:

$$\frac{\Gamma \vdash t_1 : T, \quad t_1 \to_\beta t_2}{\Gamma \vdash t_2 : T} \text{ (Subject Reduction)}$$

This property guarantees that reduction steps preserve typing, ensuring that well-typed terms remain well-typed throughout computation. The progress theorem is formalised as:

$$\frac{\vdash t : T}{\text{Value}(t) \ \vee \ \exists t'.(t \to_\beta t')} \text{ (Progress)}$$

This establishes that well-typed terms are either already values or can take a reduction step, preventing terms from becoming stuck in invalid states. Perhaps the most significant property of STLC

is strong normalisation:

$$\forall t, T.(\vdash t : T \implies \exists v.\text{Value}(v) \wedge t \twoheadrightarrow_\beta v)$$

This property ensures that all reduction sequences starting from a well-typed term eventually terminate in a value.

2.5.5 Rules of Inference

Transitioning from the inference rules of the untyped lambda calculus to the STLC is an augmentation of the original rules with additional type information. In Natural Deduction, the introduction and elimination rules for propositional connectives specify how to construct and deconstruct logical formulas. Similarly, in the untyped lambda calculus, rules decide term construction and reduction. The STLC unifies both of these approaches by incorporating types as syntactic objects that track the behaviour of terms. The original inference rules become typing judgments, where the turnstile now indicates not just derivability but also type assignment. The contexts evolve from tracking assumptions about propositions to maintaining type assignments for variables. Each introduction rule becomes a typing rule for constructing terms of a specific type, while elimination rules become typing rules for term elimination that preserve type safety. Product Type introduction ($\times$I) is formalised in the form of:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B}(\times\text{I})$$

This rule states that if we have a term M of type A and a term N of type B in context γ, we can construct a pair $\langle$M,N$\rangle$ of type $A \times B$. This corresponds to forming an ordered pair of two terms. Product Type elimination ($\times E_1$ and $\times E_2$) takes the form of:

$$\frac{\Gamma \vdash P : A \times B}{\Gamma \vdash \pi_1(P) : A}(\times\text{E}_1)$$

$$\frac{\Gamma \vdash P : A \times B}{\Gamma \vdash \pi_2(P) : B}(\times\text{E}_2)$$

These rules define projection operations. Given a pair P of type $A \times B$, π_1(P) extracts the first component of type A, and π_2(P) extracts the second component of type B. Sum Type introduction ($+I_1$ and $+I_2$) is given as:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \texttt{inl}(M) : A + B}(+\text{I}_1) \quad \frac{\Gamma \vdash N : B}{\Gamma \vdash \texttt{inr}(N) : A + B}(+\text{I}_2)$$

These rules define injection into a sum type. The notation `inl(M)` injects a term M of type A into the left side of $A + B$, while `inr(N)` injects a term N of type B into the right side of $A + B$.
Sum Type Elimination (+E):

$$\frac{\Gamma \vdash L : A + B \quad \Gamma, x : A \vdash M : C \quad \Gamma, y : B \vdash N : C}{\Gamma \vdash \text{case } L \texttt{ of inl}(x) \Rightarrow M \mid \texttt{inr}(y) \Rightarrow N : C}(+\text{E})$$

This rule implements case analysis on sum types. Given a term L of type $A + B$ and two branches (M handling the A case and N handling the B case), both producing a result of type C, we can construct a case expression that produces a C. Unit Type introduction (1I) is defined as:

$$\frac{}{\Gamma \vdash \langle\rangle : 1}(1\text{I})$$

This rule states that we can always construct the unit value $\langle\rangle$ of type 1. The empty premises indicate no preconditions are needed. Unit Type elimination (1E) is handled by:

$$\frac{\Gamma \vdash M : 1 \quad \Gamma \vdash N : A}{\Gamma \vdash N : A}(1\text{E})$$

This rule states that given a term M of type 1 and a term N of type A, we can derive N of type A. This captures that no information is carried by terms of type 1. Empty Type elimination (0E) outside of a minimal context is given by:

$$\frac{\Gamma \vdash M : 0}{\Gamma \vdash \text{abort}(M) : A}(0\text{E})$$

This rule states that if we have a term $M$of type 0 (which is impossible in a consistent context), we can derive a term of any type A using abort. This captures the principle of ex falso quodlibet.

2.5.6 Example Derivatios

PUT EXAMPLE DERIVATIONS HERE

2.5.7 Computational Properties

The STLC holds some distinct properties that differentiate it from its untyped counterpart. Most notably, STLC guarantees strong normalisation, meaning every well-typed term reduces to a normal form in a finite number of steps, making it terminating but not Turing-complete. The type system enforces constraints that prevent the encoding of general recursion and self-application, making it impossible to represent certain terms that would lead to non-termination in the untyped calculus (such as the ω-combinator). Like the untyped calculus, STLC maintains confluence through the Church-Rosser property, ensuring unique normal forms up to α-equivalence. The typing constraints provide a static guarantee against type errors and ensures subject reduction (preservation of types under reduction). The STLC supports both call-by-value and call-by-name evaluation strategies, with both strategies guaranteed to terminate for well-typed terms. This restricted computational power makes it unsuitable for general-purpose programming as a standalone system, but the STLC is very capable at expressing mathematics which make it a nice foundation for studying type systems, program verification, and the relationship between logic and computation. Any percieved limitation might also be seen as a feature, because a consequence of this is well-behaved and terminating programs.

2.6 Dependent Type Theory

Predicative calculus with dependent types naturally aligns with first-order intuitionistic logic because both systems maintain a strict hierarchical structure where definitions and quantifications can only reference previously established concepts. In contrast, impredicative calculus with dependent types corresponds to higher-order intuitionistic logic because both systems allow for more complex forms of quantification and self-reference, including the ability to quantify over predicates and functions themselves. This correspondence explains why modern proof assistants like Lean implement both approaches - predicative reasoning through universe levels for more foundationally secure constructions, and impredicative reasoning through its `Prop` universe for greater expressive power.

Dependent type theory is a formal system for specifying and reasoning about mathematical objects and proofs. It forms an alternative foundation to mathematics compared to set theory. As with it's non-dependent counterpart, every expression in this theory has an associated type. For example, $x + 0$ may denote a natural number while f may denote a function on the natural numbers. The type of an expression provides semantic information about what kind of object it represents.

The distinguishing feature that makes a type theory 'dependent', is when types can depend on terms. The canonical example is the type of lists, `List` : α. Here, α is a type parameter and `List` α denotes the type of lists whose elements have type α. So `List` : `Nat` is the type of lists of natural numbers while `List` : `Bool` is the type of lists of boolean values. The type argument α determines the type of the list. Specifically, types themselves are also terms and every type has a type called a universe. There is an infinite hierarchy of universes `Type` 0, `Type` 1, `Type` 2, etc. (also denoted `Set, Type, Type` 1, etc.) where `Type` n : `Type` $(n + 1)$. Viewing types as terms allows one to quantify over types, construct types from other types, and create type synonyms.

Dependent type theories typically include dependent function types ($\prod$-types) and dependent pair types ($\sum$-types). If α is a type and β is a type family indexed by α (meaning that for each term $a : \alpha$, βa is a type), then $(x : \alpha) \to \beta x$ denotes the type of functions that map any term $a : \alpha$ to a term of type βa. So the type of the result can depend on the value of the input. Non-dependent function types can be thought of as a special case where β does not depend on x, given as $\alpha \to \beta$. The polymorphic `cons` function is an example of a dependent type which adds an element to the front of a list, creating a new list. This function has the type $(\alpha : \text{Type}) \to \alpha \to \text{List } \alpha \to \text{List } \alpha$. It takes a type α, an element of type α, a list of elements of type α, and returns a new list of type List α. Notice how the type argument α determines the type of the subsequent arguments. Dependent pair types generalise the familiar Cartesian product type. If α is a type and β is a type family indexed by α, then $(\text{x} : \alpha) \times \beta x$ denotes the type of pairs (a, b) where $\text{a} : \alpha$ and $\text{b} : \beta$ a. The second component's type can depend on the first component's value. It is also written as $\sum(x : \alpha, \beta x)$. Ordinary binary products are a non-dependent special case, $\alpha \times \beta$.

2.6.1 The Calcuus of Constructions

The Calculus of Constructions is the most expressive component of the Lambda Cube. It incorporates the expressiveness of all preceeding calculi, accumulating in a highly expressive system that is the foundation of many proof assistants (such as Isabell, Agda, Haskell, and Lean). In this system, it is possible to define functios from terms to terms, as well as terms to types, and types to types. It is strongly normalising, and hence consistent. This system can be considered as an extension of the Curry Howard Isomorphism, which associates a term in the STLC with each Natural Deduction proof in the intuitionistic propositional logic. The CoC extends this isommorphism to proofs in the full intuitionistic predicate calculus, which includes proofs of quantified statements (which we also call propositions)

2.7 Martin-Löf Type Theory

Martin-Löf Type Theory (MLTT) represents a significant advancement in our understanding of types, logic, and computation. Developed by Per Martin-Löf in the 1970s, this theory builds upon the STLC by introducing two important concepts: dependent types and identity types. To understand

MLTT's significance, we must first understand how it extends simpler type systems. In STLC, we work primarily with typing judgments that tell us when a term has a particular type, written as $\Gamma \vdash t : A$. MLTT extends this framework by introducing several new forms of judgment, each serving a distinct purpose in the theory. The judgment A type establishes that A is a well-formed type, while $A = B$ type indicates that two types are equal. Term equality is captured by $t = s : A$, and Γ ctx ensures our contexts are well-formed. These judgments work together to create a rich system where we can reason about both terms and their types. One of MLTT's most notable contributions is the introduction of dependent types. In simpler type systems, types are independent of terms –they exist in separate worlds. Dependent types break down this barrier by allowing types to depend on terms. Consider how this generalises function types: in the STLC, we have simple function types $A \to B$ where A and B are fixed types. With dependent types, we can write $\Pi x{:}A.B(x)$, where the output type B can vary based on the input value x. This dependency is formalised by the formation rule:

$$\frac{\Gamma \vdash A \text{ type} \quad \Gamma, x{:}A \vdash B(x) \text{ type}}{\Gamma \vdash \Pi x{:}A.B(x) \text{ type}} \quad (\Pi\text{-Form})$$

This rule tells us that to form a dependent function type, we need two things: first, A must be a valid type, and second, $B(x)$ must be a valid type whenever x has type A. Perhaps the most conceptually innovative aspect of MLTT is its treatment of equality through identity types. In traditional mathematics and classical logic, equality is a simple yes-or-no proposition –two things are either equal or they aren't. MLTT takes a more nuanced approach. For any type A and terms a and b of type A, we can form a new type $\mathrm{Id}_A(a, b)$. Think of this type as containing 'evidence' or 'proofs' of equality between a and b. This type might be empty (if a and b are not equal), or it might contain proofs of their equality. The formation rule for identity types captures this idea formally:

$$\frac{\Gamma \vdash A \text{ type} \quad \Gamma \vdash a : A \quad \Gamma \vdash b : A}{\Gamma \vdash \mathrm{Id}_A(a, b) \text{ type}} \quad (\text{Id-Form})$$

This rule states that we can form an identity type whenever we have two terms of the same type. We can express these rules using product notation, which emphasises how identity types work across all types and terms. The formation and introduction rules work together as a pair:

$$\mathrm{Id} : \prod_{A:\mathrm{Type}} \prod_{a,b:A} \mathrm{Type} \qquad \mathrm{refl} : \prod_{A:\mathrm{Type}} \prod_{a:A} \mathrm{Id}_A(a, a)$$

Here, the left equation shows how identity types can be formed for any type and any pair of terms, while the right equation shows how reflexivity provides the basic building block of equality for any term with itself. The introduction rule for identity types captures an important message: every term is equal to itself. This is represented by the reflexivity constructor:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathrm{refl}_a : \mathrm{Id}_A(a, a)} \quad (\text{Id-Intro})$$

The real power of identity types comes from their elimination principle, known as the J-rule. This rule tells us how to use identity proofs. Recall from the discussion on Natural Deduction that this is exactly how Gentzen himself describes such rules. We have that

$$\frac{C : \prod_{x,y:A} \mathrm{Id}_A(x, y) \to \mathrm{Type} \quad d : \prod_{x:A} C(x, x, \mathrm{refl}_x) \quad p : \mathrm{Id}_A(a, b)}{J_{C,d}(p) : C(a, b, p)}$$

While this rule appears involved (to say the least), its intuition is straightforward: if we want to prove something about two equal terms, it suffices to prove it in the case where the terms are actually the same and their equality comes from reflexivity. The computation rule then tells us how this works in practice:

$$J_{C,d}(\text{refl}_a) \equiv d(a)$$

MLTT includes several other type constructors that work together with dependent and identity types. These include dependent sum types ($\Sigma x{:}A.B(x)$), which let us pair a term with a term of a dependent type, and disjoint unions ($A + B$), which combine types in a way that keeps track of their origin. The theory also includes `empty` and `unit` types, which serve as the extreme cases in the type system. An important characteristic of MLTT is how it balances expressiveness with computational tractability. The system maintains decidable type checking, meaning we can always determine whether a term has a given type. However, type inference –automatically determining the type of a term –becomes undecidable due to dependent types. This trade-off makes MLTT suitable both as a programming language and as a foundation for constructive mathematics. MLTT uses what we call *intensional equality*, which differs from *extensional equality* in an important way. Two terms might compute to the same result (be definitionally equal) without necessarily having a proof term of type $\text{Id}_A(a, b)$. This distinction helps maintain decidable type checking while still providing a rich theory of equality. To understand this in practice –, consider two functions: $f(x) = x + 1$ and $g(x) = 1 + x$. We know these functions give the same output for every input - they 'compute' to the same result. This is extensional equality –they behave the same way from the outside. However, in MLTT, a finer distinction is made. Even though f and g *compute* to the same results (they are definitionally equal), the system treats them as potentially distinct unless we explicitly provide a proof that they are equal –that is, a term of type $\text{Id}_A(f, g)$. This is intensional equality –it cares about how things are built, not just what they compute to (think: construstive and computable vs. the classical proof methods). By requiring explicit proofs of equality rather than automatically identifying all computationally equivalent terms, MLTT maintains decidable type checking, since we can always mechanically determine whether two terms are definitionally equal by computing them to normal form.

2.8 The Lambda Cube

The lambda cube, introduced by Henk Barendregt in 1991, represents eight variations of typed lambda calculi as vertices of a cube, with each dimension corresponding to a specific type dependency: terms depending on terms (λ), types depending on types ($\lambda 2$), terms depending on types ($\lambda\omega$), and types depending on terms (λP). The base system, STLC ($\lambda \rightarrow$), sits at the origin, while the other seven systems are obtained by combining these dependencies. The cube's most expressive vertex contains the Calculus of Constructions ($\lambda P\omega$ or λC), developed by Thierry Coquand and Gérard Huet, which incorporates all possible dependencies and serves as the theoretical foundation for proof assistants like Lean.

The lambda cube provides a systematic framework for understanding the relationships between different typed lambda calculi, organising them according to three fundamental kinds of abstraction. To understand the cube's structure, we must first recognise that it represents a space of type systems, where each point corresponds to a specific calculus with distinct capabilities for expressing relationships between terms and types. The origin of the cube, at coordinate $(0, 0, 0)$, contains the

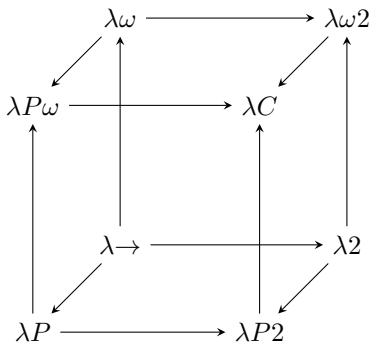

Figure 9: The Lambda Cube (arrows indicate the addition of new dependencies between terms and types)

STLC. The STLC serves as the foundation because it represents the most basic typed lambda calculus that maintains both type safety and strong normalisation. In the STLC, we have term-level abstractions $\lambda x : A.t$ and applications $(t\ s)$, where terms can only depend on other terms. The type system includes basic constructions such as function types $A \to B$, product types $A \times B$, sum types $A + B$, and the unit type 1 and empty type 0, each with their corresponding introduction and elimination rules inherited from natural deduction. From this foundation, the cube extends along three orthogonal axes, each representing a distinct form of dependency between terms and types. We can think of these axes as 'switches' that can be turned on or off, leading to eight possible combinations represented by the cube's vertices. Each axis corresponds to a fundamental extension of the type system:

The first axis extending to coordinate $(1, 0, 0)$, introduces polymorphism through System F. When we move along this axis, we add the ability for terms to depend on types. This means we can write functions that operate uniformly over all types. Mathematically, this is given as type abstraction $\Lambda X.t$ and type application $t[A]$. For example, the polymorphic identity function $\Lambda X.\lambda x : X.x$ has type $\forall X.X \to X$, where the universal quantifier $\forall$ indicates that the function works for any type X. The typing judgment must now track type variables in the context, written as $\Gamma \vdash t : A$ where Γ may contain both term and type variables.

The second axis reaching to $(0, 1, 0)$, introduces type operators through the system $\lambda\omega$. This extension allows types to depend on other types, enabling us to construct type-level functions. The canonical example is the list type operator, defined as $\text{List} = \lambda X.1 + (X \times \text{List}\ X)$. This definition shows how we can build recursive types using type-level abstraction. To accommodate type operators, we need a more sophisticated kind system. While STLC uses only the base kind $*$ for types, $\lambda\omega$ introduces function kinds $\kappa_1 \to \kappa_2$ to classify type operators. For instance, List has kind $* \to *$, indicating it takes a type and returns a type.

The third axis extending to $(0, 0, 1)$, introduces dependent types through the system λP. This powerful extension allows types to depend on terms, enabling us to express precise specifications in types themselves. The key constructions are dependent function types $\Pi x : A.B(x)$ and dependent pair types $\Sigma x : A.B(x)$. In $\Pi x : A.B(x)$, the type B may contain the term variable x, allowing the return type to vary based on the function's input. For example, we can define a type $\text{Vector}(A, n)$

representing lists of length n, where n is a term of type Nat. The typing rules become more complex, as we must handle substitution in types when terms are applied.

The remaining vertices of the cube represent systems that combine these fundamental dependencies. At $(0, 1, 1)$, the system $\lambda P\omega$ combines dependent types with type operators, allowing both term dependencies in types and type-level functions. The system $\lambda\omega 2$ at $(1, 1, 0)$ merges polymorphism with type operators, enabling higher-kinded polymorphism where we can quantify over type operators. At $(1, 0, 1)$, $\lambda P2$ joins dependent types with polymorphism. Finally, at $(1, 1, 1)$ we find the Calculus of Constructions (CoC), which unifies all three dimensions into a single, powerful type theory. Throughout the cube, several critical properties remain invariant. Each system is based on intuitionistic logic, meaning they reject the law of excluded middle and maintain a constructive interpretation of proofs. All systems in the cube exhibit strong normalisation: every well-typed term reduces to a normal form in a finite number of steps. The Church-Rosser property holds throughout, ensuring that if a term can reduce to two different terms, there exists a common term to which both can further reduce. Perhaps most remarkably, type checking remains decidable in all systems, though the algorithms become progressively more sophisticated as we add dependencies. The visual organisation of type systems within the lambda cube make it highly beneficial to learn about the correspondence between computation and logic. As we move from STLC to CoC, there is a parallel in the development from propositional logic to higher-order predicate logic, with each dimension adding expressive power while maintaining constructive content. The vertices of the cube form a lattice under the subtyping relation, with STLC at the bottom and CoC at the top. This structure sets the stage for understanding the Curry-Howard isomorphism, which we will explore in the next chapter.

2.9 The Curry-Howard Correspondence

The Curry-Howard Correspondence is to logic and computation what Maxwell's Equations are to electricity and magnetism. Also known as the proofs-as-programs interpretation, this correspondence establishes a fundamental isomorphism between the act of constructing proofs in a formal logic system and the process of writing well-typed programs in a typed calculus. This observation is a complete game-changer for the verification of proofs and software. At it's core, this correspondence states that every logical implication behaves like a computable function type, and conversely, for each valid propositional argument there exists a typed function. Once considered distributaries, logic and computation reveal themselves to be an anabranch, intimately interconnected as two ribbons of the same bow. As it turns out,

> The simplification of a proof is, in fact, no different to the evaluation of a computer program.

It is possible to look at this proposition through various lenses, some we will cover here and others we wont (category theory, physics, circuit design).

Proposition 2.1 (The Curry-Howard Isomorphism).

(i) If $\Gamma \vdash M : \varphi$ *then* $|\Gamma| \models \varphi$*, where* $|\Gamma|$ *denotes the range of* Γ*.*

(ii) If $\Gamma \vdash \varphi$ *then there exists* $M \in \Lambda_\Pi$ *such that* $\Delta \vdash M : \varphi$*, where* $\Delta = \{(x_\varphi : \varphi) \mid \varphi \in \Gamma\}$*.*

Proof Outline:

(i) By induction on the derivation of $\Gamma \vdash M : \varphi$.

(i) By induction on the derivation of $\Gamma \vdash \varphi$. Let $\Delta = \{(x_\varphi : \varphi) \mid \varphi \in \Gamma\}$.

1. The derivation is
$$\Gamma, \varphi \vdash \varphi$$
We consider two subcases:
   (a) $\varphi \in \Gamma$. Then $\Delta \vdash x_\varphi : \varphi$.
   (b) $\varphi \notin \Gamma$. Then $\Delta, x_\varphi : \varphi \vdash x_\varphi : \varphi$.

2. The derivation ends in
$$\frac{\Gamma \vdash \varphi \to \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi}$$
By the induction hypothesis $\Delta \vdash M : \varphi \to \psi$ and $\Delta \vdash N : \varphi$, and then also $\Delta \vdash MN : \psi$.

3. The derivation ends in
$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi}$$
We consider two subcases:
   (a) $\varphi \in \Gamma$. Then by the induction hypothesis $\Delta \vdash M : \psi$. (By Weakening) $\Delta, x : \varphi \vdash M : \psi$, where $x \notin \mathrm{dom}(\Delta)$. Then also $\Delta \vdash \lambda x : \varphi.M : \varphi \to \psi$.
   (b) $\varphi \notin \Gamma$. Then by the induction hypothesis $\Delta, x_\varphi : \varphi \vdash M : \psi$ and then also $\Delta \vdash \lambda x_\varphi : \varphi.M : \varphi \to \psi$.

The Curry Howard Correspondence consists of two fundamental claims: every well-typed term represents a valid proof, and every proof can be represented as a well-typed term. This can be stated formally as a proposition with two parts: if $\Gamma \vdash M : \varphi$ then $|\Gamma| \models \varphi$ (see footnote [1]), and conversely, if $\Gamma \vdash \varphi$ then there exists a term M in the typed lambda calculus such that $\Delta \vdash M : \varphi$ (where Δ contains typed variables corresponding to assumptions in Γ). The proof of this correspondence proceeds by structural induction on derivations, demonstrating that the connection between proofs and programs follows necessarily from their structures. An examination of the key cases of this proof reveals the structural alignment between logical and computational systems.

The base case considers the minimal proof: using an assumption present in the context. In logic, this appears as the rule $\Gamma, \varphi \vdash \varphi$. In the type system, this corresponds to using a variable of the appropriate type. When φ exists in Γ, the variable x_φ can be used directly. When it does not exist, the context must be extended with a new variable. This part of the proof tells us that assumptions in logic correspond exactly to variables in programming, as both represent basic units of hypothetical reasoning. The inductive case for modus ponens demonstrates how composite proofs correspond to program composition. A proof of $\varphi \to \psi$ combined with a proof of φ yields a deduction of ψ. In the type system, this corresponds to function application: a term M of type $\varphi \to \psi$ combined with a term N of type φ forms the application MN of type ψ. The inductive hypothesis ensures that subproofs correspond to well-typed terms, and the typing rules ensure their composition preserves this

[1] The notation $|\Gamma|$ extracts just the types from context Γ, and $|\Gamma| \models \varphi$ indicates that φ follows semantically from these types.

correspondence. The case for implication introduction provides particular insight. In logic, proving $\varphi \to \psi$ requires assuming φ and proving ψ. The proof addresses two cases: when φ exists among the assumptions and when it does not. This corresponds precisely to function abstraction in the type system, where a function is constructed by assuming an input variable and building a term of the output type. The two subcases correspond to whether the typing context requires extension with a new variable. This parallel structure reflects the fundamental nature of hypothetical reasoning in both systems.

The correspondence maintains consistency because both systems implement substitution identically. An assumption in a proof later discharged by substituting a specific proof corresponds to substituting a term for a variable in programming. This substitution property appears formally in the weakening lemma demonstrated in Section 2.3, Figure 1, which establishes that new assumptions/variables can be added without invalidating existing proofs/terms. The inductive structure of the proof explains the natural extension of the correspondence to more complex logical systems and type systems. New logical connectives or type constructors that follow the pattern of introduction and elimination rules automatically maintain the correspondence. This explains why dependent types correspond to quantifiers, sum types to disjunction, and product types to conjunction—these constructions follow the fundamental patterns of proof theory. The proof also establishes the correspondence between proof normalisation and program evaluation. Each proof normalisation step, which simplifies a proof by removing an introduction rule followed by an elimination rule, corresponds to β-reduction in the lambda calculus:

$$(\lambda x : \varphi.M)N \rightsquigarrow M[N/x]$$

In proof theory, this reduction represents the simplification of a proof structure. When a general proof rule (represented by $\lambda x : \varphi.M$) is applied to a specific proof (represented by N), the result is a direct proof ($M[N/x]$) where the specific case has been substituted into the general scheme. In computation, this same reduction represents function evaluation. A function ($\lambda x : \varphi.M$) applied to an argument (N) reduces to the function body (M) with the argument substituted for the parameter ($[N/x]$). We see then that the above represents both proof simplification (using a general proof scheme with a specific instance) and program evaluation (applying a function to an argument). This structural correspondence extends to proof validity and program correctness. The first part of the correspondence establishes that well-typed terms represent valid proofs, explaining why type checking serves as proof verification. The second part establishes that every proof has a program representation, explaining why proof assistants can extract computational content from formal proofs. The structural correspondence provides the theoretical foundation for proof assistants based on type theory. The construction of a proof simultaneously builds a program whose type expresses the theorem being proven. Type checking verifies proofs by ensuring term well-typedness, while the execution of proofs as programs provides computational content for theorems.

Propositional Logic		**Type Theory**	
Component	**Formula**	**Component**	**Formula**
Conjunction	$\wedge$	Product	$\times$
Disjunction	$\vee$	Sum	$+$
Implication	$\rightarrow$	Function	$\rightarrow$
Negation	$\neg$	False	$\bot$

Table 1: Correspondence between Propositional Logic and Type Theory

Predicate Logic		**Dependent Type Theory**	
Component	**Formula**	**Component**	**Formula**
Universal quantification	$\forall x : A.B(x)$	Dependent Product Type	$\prod_{x:A} B(x)$
Existential quantification	$\exists x : A.B(x)$	Dependent Sum Type	$\sum_{x:A} B(x)$

Table 2: Correspondence between Predicate Logic and Dependent Type Theory

Rule Type	Natural Deduction (Propositional Logic)		The Simply Typed Lambda Calculus ($\lambda \to$)	
	Logical Connective	Formula	Type	Formula
Introduction	Conjunction	$\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge\mathrm{I}$	Product	$\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B}\ \times\mathrm{I}$
Elimination	Conjunction	$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge\mathrm{E}_1\ \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge\mathrm{E}_2$	Product	$\dfrac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_1(M) : A}\ \times\mathrm{E}_1\ \dfrac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_2(M) : B}\ \times\mathrm{E}_2$
Introduction	Disjunction	$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee\mathrm{I}_1\ \dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee\mathrm{I}_2$	Sum	$\dfrac{\Gamma \vdash M : A}{\Gamma \vdash \mathrm{inl}(M) : A + B}\ +\mathrm{I}_1\ \dfrac{\Gamma \vdash M : B}{\Gamma \vdash \mathrm{inr}(M) : A + B}\ +\mathrm{I}_2$
Elimination	Disjunction	$\dfrac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ \vee\mathrm{E}$	Sum	$\dfrac{\Gamma \vdash M : A + B \quad \Gamma, x : A \vdash N : C \quad \Gamma, y : B \vdash P : C}{\Gamma \vdash \mathrm{case}\ M\ \mathrm{of}\ \mathrm{inl}(x) \Rightarrow N \mid \mathrm{inr}(y) \Rightarrow P : C}\ +\mathrm{E}$
Introduction	Implication	$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to\mathrm{I}$	Function	$\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B}\ \to\mathrm{I}$
Elimination	Implication	$\dfrac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \to\mathrm{E}$	Function	$\dfrac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ \to\mathrm{E}$
Introduction	Negation	$\dfrac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg\mathrm{I}$	False	$\dfrac{\Gamma, x : A \vdash M : \bot}{\Gamma \vdash \lambda x : A.M : A \to \bot}\ \bot\mathrm{I}$
Elimination	Negation	$\dfrac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot}\ \neg\mathrm{E}$	False	$\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash N : A \to \bot}{\Gamma \vdash NM : \bot}\ \bot\mathrm{E}$

Rule Type	Natural Deduction (First Order Predicate Logic)		λP System with Dependent Types	
	Component	Formula	Component	Formula
Introduction	Universal Quantification	$\dfrac{\Gamma \vdash P(x)}{\Gamma \vdash \forall x P(x)}\ \forall\text{I}$	Dependent Product	$\dfrac{\Gamma, a : A \vdash M : B(a)}{\Gamma \vdash \lambda a : A.M : \Pi a : A.B(a)}\ \Pi\text{I}$
Elimination	Universal Quantification	$\dfrac{\Gamma \vdash \forall x P(x)}{\Gamma \vdash P(t)}\ \forall\text{E}$	Dependent Product	$\dfrac{\Gamma \vdash M : \Pi x : A.B(x) \quad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B(N)}\ \Pi\text{E}$
Introduction	Existential Quantification	$\dfrac{\Gamma \vdash P(t)}{\Gamma \vdash \exists x P(x)}\ \exists\text{I}$	Dependent Sum	$\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash N : B(M)}{\Gamma \vdash \langle M, N\rangle : \Sigma x : A.B(x)}\ \Sigma\text{I}$
Elimination	Existential Quantification	$\dfrac{\Gamma \vdash \exists x P(x) \quad \Gamma, P(y) \vdash Q}{\Gamma \vdash Q}\ \exists\text{E}$	Dependent Sum	$\dfrac{\Gamma \vdash M : \Sigma x : A.B(x) \quad \Gamma, a : A, b : B(a) \vdash N : C(\langle a, b\rangle)}{\Gamma \vdash \text{let } \langle a, b\rangle := M \text{ in } N : C(\langle M\rangle)}\ \Sigma\text{E}$
Introduction	Equality	$\dfrac{t_1 =_\beta t_2}{\Gamma \vdash t_1 = t_2}\ =\text{I}$	Identity Type	$\dfrac{}{\Gamma \vdash \text{refl}_a : \text{Id}_A(a, a)}\ \text{IdI}$
Elimination	Equality	$\dfrac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash P(t_1) \to P(t_2)}\ =\text{E}$	Identity Type	$\dfrac{\Gamma \vdash p : \text{Id}_A(a, b) \quad \Gamma, x : A, q : \text{Id}_A(a, x) \vdash C(x, q)}{\Gamma \vdash J(a, b, p, \lambda x.\lambda q.C(x, q)) : C(b, p)}\ \text{IdE}$

Table 4: Correspondence between Natural Deduction for First Order Predicate Logic and the λP System with Dependent Types

Looking at it from a general proof theory perspective, the following table outlines the correspondence between logic, type theory, and mathematics.

Logic	**Type Theory**	**Mathematical Analogue**
Formula	Type	Set
Proof	Term	Element of a set
Formula is true	Type has an element	Non-empty set
Formula is false	Type does not have an element	Empty set
Logical constant $\top$ (truth)	Unit type	Singleton set
Logical constant $\bot$ (falsehood)	Empty type	Empty set

Table 5: Correspondence between logic, Type Theory, and Mathematical Structures

3 The L∃∀N Proof Assistant

3.1 Introduction

The development of Lean began at Microsoft Research in 2013 under Leonardo de Moura. The system arose from a need for a proof assistant that could effectively combine formal mathematics with computational efficiency. De Moura's work focused on implementing dependent type theory in a way that would support both programming and mathematical proof verification. The system has progressed through four major versions. Lean 1 and 2 established the basic architecture, implementing a kernel based on dependent type theory with a preliminary elaboration system. Lean 3, released in 2017, introduced a metaprogramming framework that allowed users to write custom automation within the system itself. This version saw the beginning of mathlib, the mathematical component library that has been the bedrock for mathematical formalisation in Lean. Lean 4, released in 2021, was a complete rewrite of the system. The current version implements a new compiler architecture that improves performance while maintaining the logical foundations of its predecessors. This version also introduces new programming capabilities, enhancing Lean's utility as both a proof assistant and a general-purpose programming language. Since 2013, Lean has transitioned from a research project to an open-source system supported by an international community of mathematicians and computer scientists. The system implements the Calculus of Constructions with inductive types, building on the theoretical work of Martin-Löf, Coquand, and Huet. Development of Lean now proceeds through community contribution, with major developments coordinated through the Lean prover community.

The Kernal The system's architecture is built on a small kernel that implements the rules of dependent type theory. The kernel is responsible for verifying that every proof term satisfies the rules of the type theory behind it. The kernel's design implements the 'de Bruijn criterion', meaning that the correctness of any proof depends only on the correctness of this component. This makes sure that the more complex components of the system, such as proof automation and user interface, cannot introduce logical inconsistencies. The type-theoretical foundation of Lean builds on the Calculus of Inductive Constructions. The system implements an infinite hierarchy of cumulative

universes, beginning with the universe `Prop` at level zero. This impredicative universe `Prop` contains propositions, while above it extends a sequence of predicative universes `Type 0`, `Type 1`, and so forth. This universe structure allows Lean to represent both mathematical propositions and computational types within its framework without running into Russell or his paradox. Verification in Lean proceeds through type checking. When a user writes a proof, the system translates this input into a proof term in its internal representation. The kernel then verifies this term by checking that it has the correct type according to the rules of dependent type theory. Through the Curry-Howard correspondence, this process reduces all mathematical verification to type checking, transforming mathematical proofs into computational objects that can be mechanically verified.

The Elaboration System The elaboration system transforms high-level mathematical notation and proof scripts into explicit proof terms for kernel verification. This system handles implicit arguments, coercion insertion, overload resolution, and typeclass inference. The elaborator employs unification algorithms to resolve implicit details, enabling users to write (somewhat) natural mathematical expressions while maintaining formal precision. Lean implements memory management through reference counting and garbage collection for term management, using persistent data structures for efficient proof term manipulation. The system includes parallel processing capabilities, allowing simultaneous elaboration and proof checking of independent components. This parallel architecture maintains synchronisation to preserve logical soundness while enabling efficient processing of complex mathematical objects.

Metaprogramming and Mathlib The metaprogramming framework allows users to write programs in Lean that construct or manipulate proofs. This framework provides a monadic interface for tactic programming, enabling controlled access to the proof state and elaborator functionality. Through metaprogramming, users can implement custom automation while preserving the system's logical guarantees. The standard library, mathlib, implements definitions and theorems organised in a hierarchy that is intended to reflect mathematical structure (albeit a little tricky to navigate on first glance). This library incites hope that this system has the capacity for large-scale formalisation, having incoorporated solutions for dependency management and consistency maintenance. Lean's approach to definitions and proofs encourages conservative extension principles, which helps to contain some kind of redundency explosion across the internet. New definitions must be shown to be well-formed and consistent before being accepted into the library, while axioms remain carefully controlled to maintain logical consistency. This design means that the formalisation of mathematics can steadily grow, while preventing the introduction of contradictions. Through a combination of architecture and community support, Lean provides a foundation for mechanical verification of mathematical proofs. The system transforms mathematical proof into a concrete, verifiable process while maintaining the expressivity needed for mathematical reasoning, which makes it both useful and fun to use.

3.2 Lean Syntax

Lean's syntax follows naturally from its roots in dependent type theory and the Curry Howard Correspondence where we observe a cunning relation between propositions and types. We may begin our document by declaring a proposition, say `P`, `Q`, `R`, and `S` as types in the universe `Prop`, with each proof as a term inhabiting that type. This will declaration remain throughout the document.

```
variable (P Q R S : Prop)
```

Lean makes generous use of keywords, which are special reserved words that have specific meanings and functions, such as the `def` keyword, which is the basic syntax for definitions and introduces new objects. The general syntax for using the `def` keyword is

```
def name {universe_params} (params) : type := body
```

Where `name` is the identifier, `universe_params` gives optional implicit universe parameters (explained in Section **??**), `params` is zero or more explicit parameters with their types, `type` specifies the return type, and `body` contains the implementation. To see this in action, the definition of function composition can be expressed as:

```
def compose (g : Q → R) (f : P → Q) : P → R :=
λ x =>
g (f x)
```

The type system in Lean directly corresponds to logical propositions through the Curry-Howard correspondence. Universal quantification is expressed using the Pi-type notation, written in Lean as `(x : α] → β x`, where x is a variable of type α and β x is a type that may depend on x. This corresponds to the mathematical notation $\forall$x:α, β(x). The arrow type ($\rightarrow$) represents simple function types and logical implication when working with propositions. For example, a basic theorem about function properties can be stated as:

```
theorem comp_assoc (h : R → S) (g : Q → R) (f : P → Q) :
compose h (compose g f) = compose (compose h g) f :=
rfl
```

Existential quantification is denoted using Sigma-types, written as `Exists` or $\exists$ in Lean. These constructions allow us to express statements about the existence of mathematical objects with specific properties. Consider this fundamental example demonstrating existential quantification:

```
def has_inverse {α : Type} (f : α → α) : Prop :=
∃ g : α → α, (∀ x, g (f x) = x) ∧ (∀ x, f (g x) = x)
```

Function application in Lean follows standard mathematical notation, where `f x` denotes the application of function f to argument x. Note our first use of an implicit parameter, since α wasn't declared as a variable of type `Prop` at the beginning of the document. Lean supports both prefix and infix notation, allowing expressions to be written in the mathematicians' flavour of choice. The syntax for lambda abstractions uses the λ symbol followed by the parameter and body, corresponding to the mathematical notation $x \mapsto f(x)$. For example:

```
def square : Nat → Nat :=
λ n =>
n * n
```

Type declarations in Lean serve dual purposes: they act as both computational specifications and logical propositions. When working with propositions, the colon-equals notation (`:=`) provides definitions, while the colon notation (`:`) specifies types or propositions without providing proofs. This distinction is important for understanding how Lean separates specification from implementation:

```
def is_even (n : Nat) : Prop := ∃ k : Nat, n = 2 · k

theorem four_is_even : is_even 4 :=
⟨2, rfl⟩
```

The syntax for dependent types allows for precise specification of mathematical structures. A dependent function type `(x : α) → βx` represents both universal quantification in logic and dependent functions in mathematics, where the return type βx may depend on the value of the argument `x`.

3.3 Proof Terms

Proof terms are the Lego™ blocks of formal verification in Lean and it's typre-theoretic kernal. A proof term is an object that provides explicit evidence for the truth of a proposition, constructed within the bounds of dependent type theory. When we construct a proof term in Lean, we are simultaneously creating both a program and a formal proof whose correctness can be mechanically verified. These terms are a precise, unambiguous representation of logical reasoning, where each inference step corresponds to a specific typed construction. Proof terms are the most primitive mechanism through which Lean can interact with the user to verify expressions, and are generally too verbose to use on their own. The working mathematician should know how they work, build some for themselves to get a feel for them, and then read the next chapter for the next installment. The BHK interpretation shines here, as we see constructive logic permeate the very foundation of how we form valid expressions in propositional and predicate logic. We first consider a basic logical implication:

```
theorem modus_ponens {p q : Prop} : p → (p → q) → q :=
λ (hp : p) (hpq : p → q) =>
hpq hp
```

In this example, the proof term `λ (hp : p) (hpq : p → q) => hpq hp` directly corresponds to the BHK interpretation and natural deduction proof. The lambda abstraction `λ hp` introduces the assumption `p`, while `hpq hp` represents the application of the implication to our assumption, delivering `q`. Note the use of the deduction theorem here, where the antecedent of an implication in the conclusion is taken over to the left-hand side of the turnstile (premises) and then later discharged. This is all happening very quickly here in Lean, as oppoised to the multi-step process we performed by hand in a natural deduction proof tree. For conjunction operations, proof terms also mirror the logical structure we saw in natural deduction:

```
theorem and_comm {p q : Prop} : p ∧ q → q ∧ p :=
λ h =>
⟨h.right, h.left⟩
```

Here, `h.left` and `h.right` access the individual components of the conjunction, while `⟨h.right, h.left⟩` constructs a new conjunction with the components reversed. This corresponds to the natural deduction rules for conjunction elimination and introduction, as well as the BHK requirement that both conjuncts be constructed in order to validate their union. The correspondence extends to predicate logic too. Consider existential quantification:

```
theorem exists_and_comm {α : Type} {p q : α → Prop} :
(∃ x, p x ∧ q x) → (∃ x, q x ∧ p x) :=
```

```
λ h =>
match h with
| ⟨x, ⟨hp, hq⟩⟩ =>
⟨x, ⟨hq, hp⟩⟩
```

This proof term demonstrates pattern matching on an existential witness. The pattern `⟨x, ⟨hp, hq⟩⟩` destructures the existence proof, allowing us to construct a new witness with the conjunction components reversed. Universal quantification makes use of dependent function types:

```
theorem forall_and_comm {α : Type} {p q : α → Prop} :
(∀ x, p x ∧ q x) → (∀ x, q x ∧ p x) :=
λ h x =>
⟨(h x).right, (h x).left⟩
```

Here, the proof term `λ h x, ⟨(h x).right, (h x).left⟩` represents a function that takes a proof of the universal statement and produces a proof of the commuted statement for any given `x`. We see that the construction of proof terms follows rules that mirror natural deduction such that introduction rules correspond to lambda abstractions and pair constructions, elimination rules correspond to function applications and projections, and structural rules are implemented through variable usage and substitution. For example, consider the transitivity of implication:

```
theorem imp_trans {p q r : Prop} : (p → q) → (q → r) → (p → r) :=
λ hpq hqr hp =>
hqr (hpq hp)
```

This proof term demonstrates function composition, where `hpq hp` produces a proof of `q`, which is then used by `hqr` to produce a proof of `r`. The structure directly corresponds to the natural deduction proof using implication elimination (modus ponens) twice.

3.4 Tactic Mode

While proof terms provide an exact representation of mathematical reasoning, Lean offers a more intuitive approach to proof construction through step-by-step refinement of proof goals with mini-programs called tactics. Rather than directly constructing proof terms, tactics allow mathematicians to develop proofs incrementally by transforming proof states through a sequence of well-defined operations where the end result is always a proof term, but the route feels a bit more as though one is writing a mathematical proof rather than talking directly to the kernal. Each tactic implements a specific proof technique—such as introducing hypotheses, applying theorems, or decomposing complex statements –and automatically generates the corresponding proof terms. This abstraction layer allows mathematicians to focus on the high-level structure of their proofs while Lean manages the underlying formal details. Tactic mode bridges the gap between informal mathematical practice and formal verification by providing a more natural interface for proof development. When constructing proofs in tactic mode, mathematicians work with a goal state that displays current hypotheses and proof requirements, applying tactics that systematically reduce complex goals to simpler subgoals until the proof is complete. Tactics are a very useful tool in the mathematician's toolbelt for formalising proofs by enabling an intuitive, goal-directed approach to proof construction while minimising the verbosity incurred with proof term construction directly.

In tactic mode, proofs begin with an initial goal state that presents the proposition to be proved along with any available hypotheses. Consider a basic example:

```
theorem and_swap (h : P ∧ Q) : Q ∧ P := by
  cases h with
  | intro hp hq =>
    constructor
     exact hq
     exact hp
```

Here, the keyword `by` begins tactic mode. The `cases` tactic calls the conjunction hypothesis, while `constructor` builds the structure of the conclusion. The `exact` tactic completes atomic goals using available hypotheses. Each tactic application transforms the current state into a new state, potentially generating subgoals. Consider implication introduction:

```
theorem imp_trans : (p → q) → (q → r) → (p → r) := by
  intro hpq hqr hp
  apply hqr
  apply hpq
  exact hp
```

The `intro` tactic introduces hypotheses into the local context. Subsequently, `apply` performs backward reasoning, reducing goals through hypothesis application. Each step transforms the proof state systematically until completion.

3.4.1 Tactic Categories

Most tactics can be organised into categories, and although this is not a definite systematisation, it is worth discussing some of the main ones for ease of starting out.

Introduction tactics Introduction tactics manage how propositions and hypotheses enter the proof state. These tactics convert goals into workable forms by bringing assumptions into the local context. For example, `intro` manages implications and universal quantifiers to create named hypotheses (think: deduction theorem), while `constructor` builds frameworks for compound statements like conjunctions and disjunctions. Introduction tactics typically appear at the start of proofs, where they transform the initial goal into a state containing the necessary hypotheses and structural components for subsequent proof steps. For example, when proving an implication `P → Q`, the `intro` tactic converts the goal into a context where `P` is a hypothesis and `Q` becomes the new goal to prove. Consider the use of introduction tactics on the following sequent:

```
theorem and_imp : (P ∧ Q → R) → (P → Q → R) := by
  intro h hp hq
  apply h
  constructor
   exact hp
   exact hq
```

Elimination Tactics Elimination tactics in Lean provide methods for using and decomposing complex hypotheses into their constituent parts. The main elimination tactics, such as `cases`,

`destruct`, and `elim`, break down compound propositions like conjunctions, disjunctions, or existential statements into simpler components that can be directly used in proofs. For example, when working with a conjunction `P ∧ Q`, the `cases` tactic separates it into individual hypotheses for `P` and `Q`. Similarly, when handling a disjunction `P ∨ Q`, elimination tactics create separate proof branches for each possibility. These tactics should look familiar after learning about the elimination rules from natural deduction in Section 2.3. We see this in action through the following proof:

```
theorem or_and_distrib {p q r : Prop} : p ∨ (q ∧ r) → (p ∨ q) ∧ (p ∨ r) := by
  intro h
  constructor
   cases h with
    | inl hp => left; exact hp
    | inr hqr => right; exact hqr.left
   cases h with
    | inl hp => left; exact hp
    | inr hqr => right; exact hqr.right
```

Rewriting Tactics Rewriting tactics in Lean manipulate expressions by applying equalities and equivalences within goals or hypotheses. Here we see our first example of where seemingly similar commands have a very specific operation at a machine level that the proof-artist should be savvy to. The most commonly used rewriting tactic, `rw`, substitutes terms according to equality statements, while variants like `simp` perform a near-same action in a subtly different way. The `rfl` and `simp` tactics operate at fundamentally different levels of computation in Lean's processing engine. The `rfl` (reflexivity) tactic performs a direct check for definitional equality, examining whether two terms reduce to exactly the same normal form in the kernal through β-reduction. It succeeds only if terms are computationally identical without requiring any further logical reasoning. In contrast, `simp` implements a more computationally heavy rewriting system that applies a collection of lemmas and equations (defined by the user) from the local context as 'simp'-tagged theorems. It performs ordered term rewriting using these rules, potentially exploring multiple transformation paths to reach its goal. Where `rfl` only considers computational equality, `simp` can verify equality using its preconfigured rule set. This means that while `rfl` is faster due to its lighter computational requirements, `simp` is more powerful but computationally intensive as it searches through possible rewriting sequences. While subtle, it is these differences in 'under-the-hood' processing that the mathematician may become familiar with over time when deciding which tactic to use in what circumstance.

When provided with an equality `a = b`, these tactics both identify occurrences of `a` in the current goal and replace them with `b`, or vice versa when used with symmetry. Rewriting can be very specifically controlled through explicit locality annotations, allowing mathematicians to specify exactly where substitutions should occur. These kinds of tactics often show up in algebraic proofs where routine term manipulation is required, such as in ring calculations or when normalising expressions to match known theorems. We see a basic example in the following proof:

```
theorem add_zero (n : Nat) : n + 0 = n := by
  induction n with
  | zero => rfl
  | succ n ih =>
    rw [nat.add_succ]
```

```
6         rw [ih]
```

Automation Tactics Automation tactics in Lean combine multiple reasoning steps into single commands that attempt to solve goals automatically. The first automation tactics to learn should be `aesop`, which performs automated proof search using configurable rule sets, and `tauto`, which handles propositional tautologies. These tactics apply built-in short-cuts and decision procedures to resolve goals without requiring explicit step-by-step guidance (very helpful, and fun), though their success depends on the complexity of the goal and the available lemmas in scope (as well as the guiding hand of the artist). Mathematicians will find merit in employing automation tactics for straightforward logical deductions or when dealing with goals that would require monotonous mechanical steps to prove manually, though they should be used judiciously as their behavior can become unpredictable with complex goals. Consider the following example: Automation tactics combine multiple reasoning steps:

```
1     theorem simple_auto {p q : Prop} (h1 : p) (h2 : p → q) : q := by
2       aesop
```

Versus the (slightly) more verbose

```
1 theorem simple_explicit {p q : Prop} (h1 : P) (h2 : P → Q) : Q := by
2 apply h2
3 exact h1
```

As proofs build in complexity, the number of lines of code spared will obviously increase at a rate that justifies their use.

Working with Tactics Tactics make the Lean environment feel more flexible while still retaining it's formal rigor. Proof scripts can be formed in as many ways as a pen-and-paper proof – sometimes dealing with subcases individually, sometimes rewriting an expression repeatedly, and sometimes calling procedures that do arithmetic or logical reasoning automatically. The end product is always a proof term that Lean checks for correctness on the foundation of dependent type theory. Tactic-based proofs can seem complicated at first (especially when trying to decide which one to choose!) and can be less transparent if each step is not documented. However, they empower the user to work with much more complicated arguments without the thousands of lines of code that an equivalent proof all made from explicit proof-terms would induce, and large or branching case analyses in a structured and interactive way. Beginners will likely find themselves switching between the direct use of `exact` or `apply` for small steps and making use of more involved tactics such as `simp`, or `aesop` when more familiar with the basics. Over time, different approaches feel like an intuitive place to start, and combinations of tactics will spring to mind when looking at a theorem. The user might enjoy finding ways to mix short proofs that are more coherent with occasional blasts of automation when the details are too tedious.

The best approach to becoming confident with tactics is to experiment with them and look at how Lean displays intermediate goals in the infoview with each tactic. One approach might be to rewrite a proof with fewer lines by letting the system do more automation, or alternatively use more explicit steps for clarity. Since the statements proved by tactics reduce internally to the same proof terms no matter what path is taken to get there, the difference is a matter of taste and efficiency.

3.5 Formalising Mathematics

The transition from working with atomic propositions to formalising mathematical structures in Lean is first and foremost a process of translating mathematical concepts, definitions, and proofs into a language founded in dependent type theory –rather than the typical set theory. For mathematicians familiar with type theory, constructive logic, and hierarchical abstraction, this transition is more straightforward, but any mathematician will become familiar and well-versed with time and practice. As with pen-and-paper mathematics, the formalisation process beings with precise definitions. Lean's type system permits the expression of mathematical structures through typeclasses and structures, which takes on the form of axioms and operations. For example, a group can be defined as a typeclass bundling a carrier type, a binary operation, an inverse, and proofs of associativity, identity, and inverse laws. Such definitions mirror the standard mathematical practice of specifying structures via their constituent components and axioms. Similarly, properties like commutativity or continuity are encoded as predicates over types, often expressed using dependent function types or inductive propositions.

As with the sequent proofs from Section 3.4, mathematical proofs in Lean are generally constructed using tactic mode, which will be translated to proof terms for the kernal to verify. While tactic proofs for propositional logic rely on basic logical steps (e.g., `intro`, `apply`, `exact`), proofs of mathematical objects require domain-specific tactics and automation. For example, in abstract algebra, the `simp` tactic simplifies expressions using ring axioms, while `norm_num` verifies numerical inequalities in analysis. The `rw` tactic rewrites terms using equalities or equivalences, which is the basis for manipulating algebraic expressions or topological properties. These tactics abstract low-level proof term construction, allowing mathematicians to focus on high-level reasoning and the direction of the proof.

Transitioning from propositional logic, or even basic mathematical proofs, to more elaborate mathematics involves managing larger proof states and making use of Lean's community-driven mathematical library, `Mathlib`. From this library, one can access definitions and theorems across diverse areas –from adjacency matrices in graph theory to the epsilon-delta definition of continuity in analysis, `Mathlib` has the mathematician covered. Importing `Mathlib` delivers a huge (and growing) database of reusable components, ready to be played with.

As with a pen-and-paper proof, there is a balance between computational and propositional content that must be made. Lean distinguishes between data-carrying structures (e.g., a specific graph with explicit vertices and edges) and proof-relevant propositions (e.g., the statement that a graph is bipartite). Definitions prioritise computational relevance, while theorems focus on verifiable claims. For example, defining a metric space involves specifying a type and a distance function, while proving its completeness requires constructing a limit for every Cauchy sequence –a process that combines recursive definitions (for sequences) and existential quantifiers (for limits).

To illustrate, consider formalising the statement 'every finite group has a composition series'. In Lean, this begins by defining `Group`, `Finite`, and `CompositionSeries` as typeclasses or structures. The proof then proceeds by induction on group order, using tactics like `cases` to decompose hypotheses and `use` to construct required series. Automation via `aesop` or `library_search` might handle intermediate steps, while explicit term construction fills in the gaps with explicit proof terms.

Inference Rule	**Structure**
Conjunction Introduction	$\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge \mathrm{I}$
Conjunction Elimination	$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge \mathrm{E}_1 \quad \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge \mathrm{E}_2$
Disjunction Introduction	$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee \mathrm{I}_1 \quad \dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee \mathrm{I}_2$
Disjunction Elimination	$\dfrac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} \vee E$
Implication Introduction	$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \to \mathrm{I}$
Implication Elimination	$\dfrac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B} \to E \quad (MP)$
Negation Introduction	$\dfrac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A} \neg I$
Negation Elimination	$\dfrac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot} \neg E$

Table 6: Natural Deduction Rules for Propositional Logic

4 Appendix

Extending the CH isomorphism we have that

Logic	**STLC**	**Lean 4**
$\wedge I$	(p, q)	`And.intro p q`
$\wedge E_l$	$fst\ \ t$	`And.left t`
$\wedge E_r$	$snd\ \ t$	`And.right t`
$\rightarrow I$	$\lambda p : P.$	λ `p : P =>`
$\rightarrow E$	$(f\, t)$	`(f t)`
$\vee I_l$	$inl\ \ p$	`Or.intro_left <right-disj> p`
$\vee I_r$	$inr\ \ p$	`Or.intro_right <left-disj> p`
$\vee E$	$cases\ \ t\ \ f\ \ g$	`Or.elim t f g`

Table 7: Syntax correspondence between propositional logic, STLC, and Lean 4