A Foundation for Modern Theorem Proving in Lean 4

Where Proofs Meet Programs

By

Maia Traforti

School of Mathematics and Statistics

University of Canterbury

Submitted on: February 5th, 2025
Under the supervision of Robert Culling

A research report submitted in partial fulfillment
of the requirements for the degree of

Bachelor of Science (Mathematics)

Contents

Part I

Logical Foundations

CHAPTER 1

History and Motivation

The development of formal logic and the foundations of computability have evolved hand-in-hand over the last century, with important ideas bubbling up 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 foundation 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 (Thiel, 1982). 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 early development of the systems we use today (Newen et al., 2001). As Frege's work filtered through the mathematical world in the early 20th century, Bertrand 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 (Peckhaus, 1997). In Russell's type theory, every term has a type, and types are arranged hierarchically (Boffa, 1984). This stratification of objects into levelled 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 (Martin, 1943).

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 (Peckhaus, 1997). 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 (Pelletier and Hazen, 2012). 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) (Irwin, 2008). 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 (Emerich, 2016).

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 (Aschieri and Zorzi, 2016). 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 (Hofmann, 1994; Ireland, 1993). 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 (Dowek, 2012). 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 (Barthe and Elbers, 1996). 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 (Scott, 1980). 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 (Martin, 2008). The story of propositions-as-types is that 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 (Nowak, 1977). 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 (Fairtlough and Mendler, 2000). 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 (Mella, 2012).

CHAPTER 2 Preliminaries

2.1 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 (Crossley, 2011). 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 (Markov, 1968; Troelstra, 1977a). 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 (Besnard, 1989).

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 expressions 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 (Besnard, 1989). 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 (Troelstra, 1977a; Díez, 2000).

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 (Bjesse, 2005). Abstract mathematics often uses first-order logic with classical principles to balance expressiveness with available proof techniques. The frameworks provide the *language* for expressing mathematical concepts, while principles establish the *methods for deriving truths* within that language (Díez, 2000; Troelstra, 1977b).

2.1.1 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 (Troelstra, 1977a; Seisenberger, 2003). 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 (Longo, 2011; Garofalo et al., 2015).

Constructive Semantics with BHK 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 (Díez, 2000). 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 (Troelstra, 1977b; Sato, 1997). Existential quantification $\exists x P(x)$ requires both a specific element a and a proof that $P(a)$ holds. The proposition $\perp$ representing falsehood has no proof, while negation $\neg P$ represents a construction transforming any proof of P into a proof of $\perp$. 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 (Restall, 2004).

2.2 Models of Computation

Computation, in its most fundamental sense, is the word we use for the mechanical manipulation of symbols according to well-defined rules to transform inputs into outputs. This concept is grounded mathematics, dating back to ancient algorithms like the Euclidean method for finding greatest common divisors, but it gained new significance through the work of mathematicians and logicians in the early 20th century. Figures like Alonzo Church, Alan Turing, and Kurt Gödel developed formal models of computation that proved certain mathematical functions were 'effectively calculable' – meaning they could be evaluated through a finite sequence of precise mechanical steps (Copeland, 1996). These theoretical foundations directly influenced the development of modern programming languages and computer science. Programming languages evolved as tools for expressing computational processes in increasingly sophisticated and abstract ways. Early languages like assembly code directly reflected the step-by-step operations of computer hardware, while modern languages provide high-level abstractions that more closely match mathematical and logical concepts. This evolution mirrors the development of mathematical notation itself – from concrete calculations with specific numbers to abstract symbolic manipulation (O'Regan, 2012). Type theory, which developed from mathematical logic, now forms the theoretical foundation for many modern programming languages, enabling them to express and verify elaborate mathematical properties about programs. The development of dependently typed programming languages like Lean represents a convergence of computation, mathematical logic, and formal verification, where programs can express both calculations and mathematical proofs within a single framework. The connection between computation and mathematics continues to drive innovation in both fields. Computational methods have transformed many areas of mathematics, enabling new kinds of proofs and discoveries, while mathematical insights lead to more powerful and reliable programming languages (de Moura et al., 2015) (Li, 1989).

Type Theory Type theory developed from investigations in mathematics during the early 20th century, particularly through the work of Bertrand Russell and Alonzo Church (Scott, 1980; Barendregt, 1991). Type theory provides a formal system for classifying mathematical objects based on their nature or 'type', helping to avoid logical paradoxes that arise when treating all mathematical objects uniformly. Types and sets serve similar purposes in mathematical foundations, but they differ in their basic construction. In set theory, everything is ultimately built from sets – numbers, functions, and other mathematical objects are all constructed as particular sets. The membership relation is primitive, and a set is defined by its elements. In contrast, type theory takes types as primitive, where a type represents a collection of objects that share common structural properties and behaviours. *Rather than being defined by their elements, types are defined* *by how their elements can be constructed and used* (Hindley, 1997). The moment when types become useful as a perspective on foundational mathematics is with the introduction of a hierarchy of types, where each mathematical object belongs to a specific type, and operations are only meaningful when applied to objects of appropriate types. This system evolved through several stages, from Russell's simple theory of types to Church's simply typed lambda calculus, and eventually to Martin-Löf's constructive type theory which forms the basis for modern proof assistants like Lean (de Moura et al., 2015; Scott, 1980). Dependent type theory was a particularly strong leap, allowing types themselves to depend on values. This extension allowed for the expression of very subtle mathematical expressions directly within the type system, which sealed the gap between programs and proofs (Constable, 1991). Dependent types can, for example, express that a sorting function not only returns a list, but specifically that it returns a sorted permutation of its own input. The evolution of type theory is a reflection of a bigger shift in mathematical foundations, moving from set theory toward more computationally-oriented frameworks that naturally support both mathematical reasoning and program verification. A similar fundamental shift occured with the Four Colour Theorem in 1976, when Kenneth Appel and Wolfgang Haken announced that a proof had been made not by a human computer, but a mechanical one. Modern type theories, such as the Calculus of Constructions implemented in Lean, incorporate features like universe polymorphism, inductive types, and cumulative hierarchies, which express a rich language for expressing mathematical concepts while remaining computationally consistent. The question remains on what the next transition even deeper into mechanically-verified proofs will look like – here we will explore it's foundations and become informed wonderers (Nederpelt and Geuvers, 2014; de Moura et al., 2015).

2.3 Formal Proofs

A formal proof is a complete and rigorous mathematical argument that follows the rules of a formal logical system, where each step is explicitly justified using precisely defined inference rules and axioms (Bjesse, 2005). In a formal proof system like Lean, every logical deduction must be mechanically verifiable, with no hidden assumptions or intuitive leaps. The proof assistant checks that each step follows validly from previous steps according to the formal rules of the system, ensuring absolute mathematical certainty (Dixon and Fleuriot, 2006). This contrasts with informal proofs, which are written in natural language and rely on shared mathematical understanding and intuition among human mathematicians (Aberdein, 2007). While informal proofs can communicate mathematical ideas more naturally and concisely, they often skip 'obvious' steps and may contain subtle gaps or imprecisions that would need to be filled in to create a fully formal version (Dietrich and Buckley, 2007). The relationship between formal and informal proofs can be understood through an analogy with programming languages: informal proofs are like high-level pseudocode that communicates the key algorithmic ideas, while formal proofs are like machine code that spells out every computational step in complete detail (Culik, 1983). Just as a high-level program must be compiled down to machine instructions, an informal proof can be elaborated into a formal proof by making all assumptions explicit and filling in all logical gaps. This process of formalisation often reveals subtle issues that were glossed over in the informal version. However, formal proofs tend to be much longer and more difficult to read than their informal counterparts, which is why mathematicians typically work with informal proofs for human consumption while using formal proofs and proof assistants for mechanical verification of especially important or complex results (Dixon and Fleuriot, 2006). Proof assistants like Lean aim to bridge this gap by providing high-level control that allow users to write proofs in a more natural style while maintaining full formal rigor behind the scenes. The proof assistant elaborates these high-level proof steps into fully formal proofs under the hood, combining the reliability of formal proof with some of the readability and intuitive understanding of informal mathematical argument. This is a kind of 'best of both worlds' approach, though there remains an art to writing formal proofs that are both machine-checkable and humanly comprehensible.

CHAPTER 3

Natural Deduction

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 (Pelletier and Hazen, 2012). 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 (Li, 1992). 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 $A \wedge B$ (Yaqub, 2013). 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 $A \rightarrow B$ (Kurokawa and Kushida, 2013).

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 (Álvez and Lucio, 2005). 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 (Ihlemann et al., 2008). 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 (Merz, 1997).

3.2 Rules of Inference

The logical framework of natural deduction can be described by its 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 (Plato, 2001), 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 (Wadler, 2015). 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 (Humberstone and Makinson, 2011). 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 $\exists$) (Schroeder-Heister, 2014).

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 $A \rightarrow B$, 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 3.1 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 (Indrzejczak, 2010). 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 (McCawley, 1991).

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

Figure 3.1: Proof of $\vdash A \to (B \to A)$ (Weakening)

Extending minimal logic to intuitionistic logic requires an additional structure, namely that of Ex 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 (Steinberger, 2016). Figure 3.2 below demonstrates the proposition A leading to $\bot$ and the resulting derivation of any proposition (in this case B):

$$\begin{array}{c} [A]^1 \\ \vdots \\ \hline \bot \\ \hline B \end{array}\ \mathrm{XF}$$

Figure 3.2: 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 (Steinberger, 2016; Aschieri and Zorzi, 2016). 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 3.3 below:

$$\dfrac{\dfrac{\dfrac{\begin{matrix}[A]^1\\ \vdots\end{matrix}}{B} \qquad \dfrac{\begin{matrix}[\neg A]^2\\ \vdots\end{matrix}}{\neg B}}{\bot}}{A}\ RAA, 2$$

Figure 3.3: 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 3.3 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.

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

Figure 3.4: Connection between DNE and RAA

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 preceding inference rules. The root of the tree represents the conclusion, while the leaves represent either initial hypotheses or assumptions introduced in subproofs (Li, 1992; Pelletier and Hazen, 2012). Take for example, a minimal logic proof of the sequent $P \to Q,\ Q \to R \vdash P \to R$ shown in Figure 3.5 below:

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

Figure 3.5: 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 3.6 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.

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

Figure 3.6: Proof of $P \lor Q, \quad P \to R, \quad Q \to D \vdash R \lor D$

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 (Li, 1992; Pelletier and Hazen, 2012). 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 (Plato, 2001; Schroeder-Heister, 2014). 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:

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

The converse also holds through modus ponens:

$$\dfrac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\,\to E$$

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 (Wadler, 2015). 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 3.7 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.

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

Figure 3.7: 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 3.8 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.

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

Figure 3.8: Proof of the sequent $\neg(P \to Q) \to P$

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 (Barthe and Elbers, 1996; Dietrich and Buckley, 2007). 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 natural deduction system is at the heart of this interwoven web of goals, with many smaller proofs combining to create a larger one.

3.4 Computational Properties

The natural deduction 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 (Pelletier and Hazen, 2012). Soundness guarantees that any conclusion derived through the rules of natural deduction is logically valid. This property tells us that if a conclusion is provable from certain premises using natural deduction 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 (Indrzejczak, 2010). Completeness, on the other hand, establishes that all valid logical consequences can be proved within the system (Troelstra, 1977b). This means that if a conclusion is a logical consequence of certain premises, then there exists a natural deduction proof of that conclusion from those premises. The completeness property ensures that the system is powerful enough to capture all valid logical relationships (Pelletier and Hazen, 2012). Natural deduction 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 (Sieg and Byrnes, 2005).

CHAPTER 4

The Untyped Lambda Calculus

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 (Grue, 2001). 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 idea of 'terms' – which can be variables, function abstractions, or function applications (Machado, 2013). 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:

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

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 (Fischer, 1972).

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 (Barendregt, 2012).

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 (Berline, 2000).

4.2 Term Construction

Construction of terms in the λ-calculus follows a set of inductive rules 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 (Kazmierczak, 1991). 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:

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

4.3 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:

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

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 (Kazmierczak, 1991).

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 $\rightarrow_\beta$, 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 (Barendregt, 1991).

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 (Scott, 1980).

4.4 Computational Properties

The untyped lambda calculus is Turing-complete, and reduction is confluent (satisfying the Church-Rosser property) (Barendregt, 1991). 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 (Scott, 1980). 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 (Kazmierczak, 1991).

4.5 Examples

Below we introduce two very cool examples of data structures that can be made with only the variables, abstractions and applications given in the untyped lambda calculus. We see in action here how possible it is to truly encode data structures with such a simple language.

4.5.1 Church Booleans

To see a concrete implementation of the untyped lambda calculus, we will consider the Church Booleans, which are a representation of Boolean values 'true' and 'false' within the untyped lambda calculus. They are defined as lambda terms that act as selectors. We define 'true' (denoted as `true` or T) as a lambda term that takes two arguments and returns the first one. In lambda notation, this is:

$$\texttt{true} := \lambda x.\lambda y.x$$

Similarly, we define 'false' (`false` or F) as a lambda term that takes two arguments and returns the second one:

$$\texttt{false} := \lambda x.\lambda y.y$$

To understand how these terms represent Boolean values, consider their behaviour when applied to two choices. If we have a conditional scenario where we want to choose between two expressions, say A and B, based on a Boolean condition C, we can write a conditional expression in lambda calculus as:

$$C\ A\ B$$

If C is `true`, then applying `true` to A and B will result in:

$$\texttt{true}\ A\ B = (\lambda x.\lambda y.x)\ A\ B \to_\beta (\lambda y.A)\ B \to_\beta A$$

Thus, when the condition is `true`, the expression evaluates to A, the first choice. If C is `false`, then applying `false` to A and B will result in:

$$\texttt{false}\ A\ B = (\lambda x.\lambda y.y)\ A\ B \to_\beta (\lambda y.y)\ B \to_\beta B$$

Thus, when the condition is `false`, the expression evaluates to B, the second choice. This effectively implements a conditional 'if-then-else' structure using Church Booleans. We can also define logical operations using these Church Boolean representations. For example, the logical 'AND' operation can be defined as follows:

$$\texttt{and} := \lambda p.\lambda q.p\ q\ \texttt{false}$$

We now consider how `and` works. If p is `true`, then $p\ q$ `false` becomes `true` q `false`, which reduces to q. So, if the first argument p is true, the result of `and` $p\ q$ is just the second argument q. If p is `false`, then $p\ q$ `false` becomes `false` q `false`, which reduces to `false`. Therefore, `and` $p\ q$ is `true` only if both p and q are `true`, and `false` otherwise, which is the correct behaviour for logical 'AND'. Similarly, the logical 'OR' operation (denoted as `or`) can be defined as:

$$\texttt{or} := \lambda p.\lambda q.p\ \texttt{true}\ q$$

If p is `true`, then p `true` q becomes `true true` q, which reduces to `true`. If p is `false`, then p `true` q becomes `false true` q, which reduces to q. So, `or` $p\ q$ is `true` if either p is `true` or q is `true` (or both), and `false` only if both are `false`, which is the behaviour of logical 'OR'. Finally, the logical 'NOT' operation can be defined as:

$$\texttt{not} := \lambda p.p\ \texttt{false}\ \texttt{true}$$

If p is `true`, then p `false true` becomes `true false true`, which reduces to `false`. If p is `false`, then p `false true` becomes `false false true`, which reduces to `true`. Thus, `not` p correctly inverts the Boolean value of p.

4.5.2 Church Numerals

Church Numerals are a representation of natural numbers within the untyped lambda calculus. A Church numeral for a number n is a higher-order function that takes two arguments: a function f and a value x. It applies the function f to the value x exactly n times. This representation allows arithmetic operations to be performed purely through function composition and application, without requiring primitive numeric types whatsoever, which is quite impressive. This is just one example of an entire system that can be delivered without a single data-structure other than functions and their composition. The fact that Church numerals can represent all computable functions on the natural numbers provides a theoretical foundation for understanding computation in terms of pure functions. Here, we can begin to see how functional programming languages began to take shape, and we haven't even introduced types yet! Now for some demonstrations.

The Church numeral for `zero` (or $\overline{0}$) is defined as a function that applies the function f zero times to x, which is simply returning x itself. In lambda notation:

$$\texttt{zero} := \lambda f.\lambda x.x$$

The Church numeral for `one` ($\overline{1}$) is a function that applies f once to x:

$$\texttt{one} := \lambda f.\lambda x.f\ x$$

The Church numeral for two `two` ($\overline{2}$) applies f twice to x:

$$\texttt{two} := \lambda f.\lambda x.f\ (f\ x)$$

In general, the Church numeral for a natural number n (denoted as $\overline{n}$) is given by:

$$\overline{n} := \lambda f.\lambda x.\underbrace{f\ (f\ (\cdots\ (f}_{n \text{ times}}\ x)\cdots))$$

This can be understood as representing the number n by the action of applying a function n times. We can define arithmetic operations on Church numerals too. The successor function (`succ`), which takes a Church numeral $\overline{n}$ and returns the Church numeral $\overline{n+1}$, can be defined as:

$$\texttt{succ} := \lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$$

To see how this works, consider applying `succ` to a Church numeral $\overline{n}$. Here, ($\texttt{succ}\ \overline{n}$) becomes $(\lambda f.\lambda x.f\ (\overline{n}\ f\ x))$. When applied to f and x, it becomes $f\ (\overline{n}\ f\ x)$. Since $\overline{n}$ applies f to x n times, $(\overline{n}\ f\ x)$ is equivalent to $f^n(x)$. Therefore, $f\ (\overline{n}\ f\ x)$ is $f(f^n(x)) = f^{n+1}(x)$, which is the Church numeral for $n+1$.

Addition of Church Numerals Addition of two Church numerals $\overline{m}$ and $\overline{n}$ can be defined as:

$$\texttt{plus} := \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x)$$

When we apply `plus` to $\overline{m}$ and $\overline{n}$, we get $(\lambda f.\lambda x.\overline{m}\ f\ (\overline{n}\ f\ x))$. Applying this to f and x gives us $\overline{m}\ f\ (\overline{n}\ f\ x)$. Then $\overline{n}\ f\ x$ applies f to x n times, resulting in $f^n(x)$. Then $\overline{m}\ f$ is applied to this result, applying f another m times. Thus, in total, f is applied $m+n$ times to x, which is the Church numeral for $m+n$.

Multiplication of Church Numerals Multiplication of two Church numerals $\overline{m}$ and $\overline{n}$ can be defined as:

$$\texttt{mult} := \lambda m.\lambda n.\lambda f.m\ (n\ f)$$

When we apply `mult` to $\overline{m}$ and $\overline{n}$, we get $(\lambda f.\overline{m}\ (\overline{n}\ f))$. Applying this to f gives us $\overline{m}\ (\overline{n}\ f)$. The term $(\overline{n}\ f)$ represents applying f n times. Let $g = (\overline{n}\ f)$. Then $\overline{m}\ (\overline{n}\ f) = \overline{m}\ g$. $\overline{m}\ g$ applies g to some value x, m times. Applying g once means applying f n times. Applying g m times means applying f $n \times m$ times. Therefore, this gives the Church numeral for $m \times n$.

CHAPTER 5

The Simply Typed Lambda Calculus

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 $\mathbb{N}$. 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 $\tau_1 \to (\tau_2 \to \tau_3)$ (Barendregt, 2013).

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 (Aczel, 1999). 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.

5.2 Typing Judgements

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 familiar Gentzen-style natural deduction form (Adams, 2006). First, we introduce the variable rule:

$$\frac{}{\Gamma, x : T \vdash x : T}\text{Var}$$

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:

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

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:

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

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 (Pierce, 2002).

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 (Pierce, 2002).

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.

5.3 Rules of Inference

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 (Geuvers and Nederpelt, 1994). 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. The typing judgments above, along with the inference rules below, combine to make the set of allowable 'moves' in the STLC natural deduction game (Duggan and Bent, 1996). Product Type introduction (×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 ⟨M,N⟩ 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:

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

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

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:

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

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), also known as 'cases' proceeds as follows:

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

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 ⟨⟩ 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:

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

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 (Pierce, 2002; Pitts, 2019).

Example Derivatives

We will now proceed through some examples of natural deduction sequent proofs using the STLC, explaining each step along the way to demonstrate the theory behind it. Figure 5.1 starts us off with a proof of $(P \to Q) \to (\neg Q \to \neg P)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[f : P \to Q]^1 \qquad [p : P]^2}{fp : Q}\,\text{App} \qquad [g : \neg Q]^3}{(fp)g : \bot}\,\text{App}}{\lambda z.g(fz) : \neg P}\,\lambda 2}{\lambda y.\lambda z.g(fz)y : \neg Q \to \neg P}\,\lambda\ 1,3}{\lambda x.\lambda y.g(xy)z : (P \to Q) \to (\neg Q \to \neg P)}\,\lambda 1$$

Figure 5.1: Proof of $(P \to Q) \to (\neg Q \to \neg P)$

Starting from the top of the tree, we begin with three assumptions, given by square brackets with superscript numbers for discharge tracking: $[f : P \to Q]^1$, $[p : P]^2$, and $[g : \neg Q]^3$. These represent our working hypotheses that we'll eventually discharge through lambda abstraction. The first application rule `(App)` combines $f : P \to Q$ with $p : P$ to produce $fp : Q$. This represents the function application typing judgement we discussed earlier, where a function of type $P \to Q$ is applied to an argument of type P, producing a term of type Q. The second application rule combines this result $fp : Q$ with $g : \neg Q$. Since $\neg Q$ is equivalent to $Q \to \bot$ in intuitionistic logic, this application produces $(fp)g : \bot$, resulting in a contradiction. Now we begin discharging assumptions through lambda abstraction. The first abstraction ($\lambda 2$) discharges the assumption $[p : P]^2$, producing $\lambda z.g(fz) : \neg P$. This step transforms our term into a function that takes a proof of P and produces a contradiction, which is precisely what $\neg P$ means. The next abstraction ($\lambda 1, 3$) simultaneously discharges $[g : \neg Q]^3$, giving $\lambda y.\lambda z.g(fz)y : \neg Q \to \neg P$. This represents a function that takes a proof of $\neg Q$ and produces a proof of $\neg P$. The last abstraction ($\lambda 1$) discharges our initial assumption $[f : P \to Q]^1$, giving us our final term $\lambda x.\lambda y.g(xy)z : (P \to Q) \to (\neg Q \to \neg P)$. This represents a function that takes a proof of $P \to Q$ and produces a proof of $\neg Q \to \neg P$, completing our proof. Figure 5.2 below demonstrates currying in the STLC. The proof tree shows how we can transform a function that takes a pair of arguments $(P \times Q) \to R$ into a curried function $P \to (Q \to R)$ that takes these arguments one at a time.

$$\dfrac{\dfrac{\dfrac{f : (P \times Q) \to R \qquad \dfrac{[p : P]^1 \qquad [q : Q]^2}{(p, q) : P \times Q}\,(\times\text{I})}{f(p, q) : R}\,\text{App}}{\lambda q.f(p, q) : Q \to R}\,\lambda 2}{\lambda p.\lambda q.f(p, q) : P \to (Q \to R)}\,\lambda 1$$

Figure 5.2: Proof of $f : (P \times Q) \to R \vdash P \to (Q \to R)$ (Currying)

Here, we start with a given assumption $f : (P \times Q) \to R$ and two additional assumptions that we'll eventually discharge: $[p : P]^1$ and $[q : Q]^2$. The superscripts 1 and 2 again track which lambda abstractions will discharge these assumptions. The first inference rule used is $(\times\text{I})$, the introduction rule for products. This rule combines our assumptions $p : P$ and $q : Q$ to form a pair $(p, q) : P \times Q$. This product introduction is essential as it constructs the appropriate input type for our function f. Next, we apply the application rule `(App)` to combine our function $f : (P \times Q) \to R$ with the pair we just constructed $(p, q) : P \times Q$. This gives us $f(p, q) : R$, the result of applying f to our pair of arguments. We then begin the currying process through lambda abstractions. The first abstraction $(\lambda 2)$ discharges the assumption $[q : Q]^2$, producing $\lambda q.f(p, q) : Q \to R$. This creates a function that takes a Q and produces an R, while p remains free. The second abstraction $(\lambda 1)$ discharges $[p : P]^1$, producing our target term $\lambda p.\lambda q.f(p, q) : P \to (Q \to R)$. This completes our fully curried function that takes arguments of type P and Q one at a time to produce a result of type R. This proof shows us the relationship between product types and function types in typed lambda calculus, specifically how any function that takes a pair of arguments can be transformed into an equivalent curried function that takes these arguments sequentially. This turns out to be a very useful feature down the track. We now turn to the sequent given in Figure 5.3 below, which demonstrates uncurrying in the STLC. This proof shows the inverse of the previous derivation –how to transform a curried function of type $P \to (Q \to R)$ into one that takes a pair of arguments $(P \times Q) \to R$:

$$\dfrac{\dfrac{\dfrac{\dfrac{g : P \times Q}{\pi_1(g) : P}\,\pi_1 \qquad f : P \to (Q \to R)}{f(\pi_1(g)) : Q \to R}\,\textsc{App} \qquad \dfrac{g : P \times Q}{\pi_2(g) : Q}\,\pi_2}{f(\pi_1(g))(\pi_2(g)) : R}\,\textsc{App}}{\lambda g.f(\pi_1(g))(\pi_2(g)) : (P \times Q) \to R}\,\lambda 1$$

Figure 5.3: Proof of $f : P \to (Q \to R) \vdash (P \times Q) \to R$ (Uncurrying)

We begin with two premises: a variable $g : P \times Q$ (which appears twice in the proof tree) and our curried function $f : P \to (Q \to R)$. The two appearances of g allow us to extract both components of the pair using projection operations. On the left branch, we apply the first projection rule (π_1) to $g : P \times Q$ to obtain $\pi_1(g) : P$. This extracts the first component of the pair. We then apply this to our function f using the application rule `(App)`, producing $f(\pi_1(g)) : Q \to R$. This intermediate result is a function expecting an argument of type Q. On the right branch, we apply the second projection rule (π_2) to $g : P \times Q$ to obtain $\pi_2(g) : Q$. This extracts the second component of the pair. We then use another application rule to apply our intermediate function $f(\pi_1(g))$ to this second component, yielding $f(\pi_1(g))(\pi_2(g)) : R$. We then use lambda abstraction $(\lambda 1)$ to bind the free variable g, producing our target term $\lambda g.f(\pi_1(g))(\pi_2(g)) : (P \times Q) \to R$. This results in our uncurried function that takes a pair as input and produces a result of type R. From this proof, we can see the mechanics behind how product types and their projection operations allow us to transform a curried function into an uncurried one, completing the bidirectional relationship between these two ways of handling multiple arguments in the STLC. Rule-abiding typing derivations do not lie, after all. We now proceed to demonstrate the slightly more cumbersome relationship between negation, products, and sums in the STLC. The proof in Figure 5.4 shows that given $\neg P + \neg Q$ (the sum of negations), we can derive $\neg(P \times Q)$ (the negation of a product).

$$\dfrac{\dfrac{\dfrac{\dfrac{g : [P \times Q]^1}{\pi_1 g : P}\,\pi_1 \qquad [h : \neg P]^2}{h(\pi_1 g) : \bot}\,\textsc{App}}{\lambda x.x(\pi_1 g) : \neg P \to \bot}\,\lambda 2 \qquad \dfrac{\dfrac{\dfrac{g : [P \times Q]^1}{\pi_2 g : Q}\,\pi_1 \qquad [j : \neg Q]^3}{j(\pi_2 g) : \bot}\,\textsc{App}}{\lambda y.y(\pi_2 g) : \neg Q \to \bot}\,\lambda 3 \qquad t : \neg P + \neg Q}{\dfrac{\text{cases } t\ (\lambda x.x(\pi_1 g))\ (\lambda y.y(\pi_2 g)) : \bot}{\lambda z.\text{cases } z\ (\lambda x.x(\pi_1 z))\ (\lambda y.y(\pi_2 z)) : P \times Q \to \bot}\,\lambda 1}\,\textsc{Cases}$$

Figure 5.4: Proof of $t : \neg P + \neg Q \vdash \neg(P \times Q)$

We begin with three main branches that come together in a cases (sum elimination) rule. The two left branches prepare the handling of each possible case from our sum type $\neg P + \neg Q$, while the right branch provides the sum itself. In the leftmost branch, we start with an assumption $g : P \times Q$ (marked for discharge with [1]) and apply the first projection to get $\pi_1 g : P$. We then assume $h : \neg P$ (marked [2]) and apply it to our projected value, giving $h(\pi_1 g) : \bot$. This branch is then abstracted over h to produce $\lambda x.x(\pi_1 g) : \neg P \to \bot$, showing how we handle the $\neg P$ case. The middle branch follows a similar pattern but works with the second component. From the same assumption g, we project $\pi_2 g : Q$, assume $j : \neg Q$ (marked [3]), and apply it to get $j(\pi_2 g) : \bot$. This is then abstracted to $\lambda y.y(\pi_2 g) : \neg Q \to \bot$, handling the $\neg Q$ case. The rightmost branch simply provides our given sum type $t : \neg P + \neg Q$. These three branches come together in the cases rule, which eliminates the sum type by showing how to handle each possible case. The resulting term cases $t\ (\lambda x.x(\pi_1 g))\ (\lambda y.y(\pi_2 g)) : \bot$ demonstrates that either case leads to a contradiction. We then abstract over our initial assumption g with $\lambda 1$, producing $\lambda z.\text{cases } z\ (\lambda x.x(\pi_1 z))\ (\lambda y.y(\pi_2 z)) : P \times Q \to \bot$. This final type is equivalent to $\neg(P \times Q)$, completing our proof. This derivation in particular corroborates a deep connection in constructive logic: if either component of a product can be refuted ($\neg P + \neg Q$), then the product itself can be refuted ($\neg(P \times Q)$) (think back to the BHK interpretation of conjunction). This proof term explicitly shows how to construct such a refutation by using case analysis on the sum type and projection operations on the product type. In the final blow of our demonstrations, we will derive how disjunctions of negations interact with function types in the STLC. The proof in Figure 5.5 shows that given functions $f : P \to R$ and $g : Q \to S$, along with a disjunction $\neg R \vee \neg S$, we can derive $\neg P \vee \neg Q$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[p : P]^1 \qquad f : P \to R}{fp : R}\,\textsc{App} \qquad i : [\neg R]^2}{i(fp) : \bot}\,\textsc{App}}{\lambda w.i(fw) : \neg P}\,\lambda 1}{\text{inl}(\lambda w.i(fw)) : \neg P \vee \neg Q}\,\textsc{Inl}}{\lambda x.\text{inl}(\lambda w.x(fw)) : \neg P \to \neg P \vee \neg Q}\,\lambda 2 \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[q : Q]^3 \qquad g : Q \to S}{gq : S}\,\textsc{App} \qquad j : [\neg S]^4}{j(gq) : \bot}\,\textsc{App}}{\lambda y.j(gy) : \neg Q}\,\lambda 3}{\text{inr}(\lambda y.j(gy)) : \neg P \vee \neg Q}\,\textsc{Inr}}{\lambda z.\text{inr}(\lambda y.z(gy)) : \neg Q \to \neg P \vee \neg Q}\,\lambda 4 \qquad t : \neg R \vee \neg S}{\text{cases } t\ (\lambda x.\text{inl}(\lambda w.x(fw)))\ (\lambda z.\text{inr}(\lambda y.z(gy))) : \neg P \vee \neg Q}\,\textsc{Cases}$$

Figure 5.5: Proof of $t : \neg R \vee \neg S \vdash \neg P \vee \neg Q$

The proof tree has three main branches that come together in a cases rule, which is intuitively where we begin our reasoning process for this proof once we see the need to eliminate a disjunction on the left side of the turnstile. The left and middle branches construct the possible cases of our target disjunction, while the rightmost branch provides our given disjunction $t : \neg R \vee \neg S$. In the left branch, we start with an assumption $[p : P]^1$ and our given function $f : P \to R$. Applying f to p yields $fp : R$. We then assume $[i : \neg R]^2$ and apply it to fp to get $i(fp) : \bot$. This is abstracted over p to produce $\lambda w.i(fw) : \neg P$, which is then injected into the sum type using inl to get $\text{inl}(\lambda w.i(fw)) : \neg P \vee \neg Q$. The middle branch follows a similar pattern but for Q and S. From assumptions $[q : Q]^3$ and our given $g : Q \to S$, we get $gq : S$. With the assumption $[j : \neg S]^4$, we obtain $j(gq) : \bot$. This is abstracted over q to give $\lambda y.j(gy) : \neg Q$, which is injected using inr to get $\text{inr}(\lambda y.j(gy)) : \neg P \vee \neg Q$. The cases rule then combines these branches with our given $t : \neg R \vee \neg S$. The resulting term cases $t\ (\lambda x.\text{inl}(\lambda w.x(fw)))\ (\lambda z.\text{inr}(\lambda y.z(gy))) : \neg P \vee \neg Q$ shows how to transform either a proof of $\neg R$ or a proof of $\neg S$ into either a proof of $\neg P$ or a proof of $\neg Q$. This is yet again another display of the relationship between constructive logic and the BHK: negations can be 'pulled back', so to speak, through functions. If we have functions from P to R and Q to S, and we know that either R or S is refutable ($\neg R \vee \neg S$), then we can conclude that either P or Q must be refutable ($\neg P \vee \neg Q$). The proof term here explicitly constructs this refutation using case analysis and function composition. By now, we are beginning to get our bearings on constructive logic and how their proofs are derived.

5.4 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 (Barendregt, 1991). 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) (Pierce, 2002). Like the untyped calculus, STLC maintains confluence through the Church-Rosser property, ensuring unique normal forms up to α-equivalence (Barendregt, 1991). The typing constraints provide a static guarantee against type errors and ensures subject reduction (preservation of types under reduction) (Geuvers and Nederpelt, 1994). The STLC supports both call-by-value and call-by-name evaluation strategies, with both strategies guaranteed to terminate for well-typed terms (Geuvers and Nederpelt, 1994). 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 (Pierce, 2002).

CHAPTER 6

Dependent Type Theory

In the simply typed lambda calculus, we work with a strict separation between terms and types. Terms are the objects we compute with, while types classify these terms and ensure they are used consistently. The typing relation $\Gamma \vdash t : A$ asserts that term t has type A in context Γ. In simple types, when we form a type like $A \to B$ or $\text{List}(A)$, the type constructors $\to$ and List take only types as arguments, never terms. Dependent type theory dissolves this rigid boundary between terms and types. The most notable difference is that we can allow types themselves to be indexed by terms, creating families of types that vary based on term values. This is achieved by introducing dependent function types, written as $\Pi(x : A).B(x)$, where $B(x)$ is a type that can refer to the term variable x. When we inhabit such a type with a function f, applying f to an argument $a : A$ gives us a result of type $B(a)$, where we substitute the actual argument a into the type expression $B(x)$ (Barthe and Coquand, 2000). Consider the formation rules. In simply typed lambda calculus, we have rules such as:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma \vdash B : \text{Type}}{\Gamma \vdash (A \to B) : \text{Type}}$$

In dependent type theory, this generalises to:

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

Notice how B can now depend on x, a term variable. We can now express properties about terms within the type system itself. Consider a function that concatenates vectors. In simple types, we might write this as $\text{Vector} \to \text{Vector} \to \text{Vector}$, but this loses information about the lengths. With dependent types, we can write $\Pi(n : \text{Nat}).\Pi(m : \text{Nat}).\text{Vector}(n) \to \text{Vector}(m) \to \text{Vector}(n + m)$, getting the precise information of how the output length relates to the input lengths. We can also define dependent pair types $\Sigma(x : A).B(x)$, representing pairs where the type of the second component $B(x)$ depends on the value of the first component $x : A$ (McKinna, 2006). This allows us to bundle together data with proofs about that data. For example, we could define a type of 'perfect squares' as $\Sigma(n : \text{Nat}).\Sigma(m : \text{Nat}).\text{Id}(n, m * m)$, where Id is the identity type expressing equality. The price we pay for this expressivity is that type checking becomes more complex. To determine if two types $\Pi(x : A).B(x)$ and $\Pi(x : A).C(x)$ are equal, we must check if $B(x)$ and $C(x)$ are equal types for all possible values of $x : A$ (Siles, 2010). This may require evaluating terms, making type checking undecidable in general. Practical implementations often restrict the form of dependencies to maintain decidability while preserving much of the expressive power. Dependently typed systems give a more expressive basis for formal verification –far beyond what's possible in simply typed systems (Ou et al., 2004). Properties that would require separate proof terms in simple types can be encoded directly in the types themselves, allowing us to express and verify more elaborate mathematical properties within the type system itself. In dependent type theory, both sum and product types generalise their simply-typed counterparts (Garner, 2009).

Dependent Product Types ($\prod$-types) A dependent product type, denoted $\Pi(x : A).B(x)$, represents the type of functions where the output type can depend on the input value. The key is understanding that for each value x of type A, we get a potentially different return type $B(x)$. The formation rule establishes when we can form a valid $\prod$-type:

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

This rule states that to form $\Pi(x : A).B(x)$, we require that A must be a valid type in context Γ, and that $B(x)$ must be a valid type in the extended context $\Gamma, x : A$. To construct terms of $\prod$-type, we use lambda abstraction:

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

This rule shows that a function of type $\Pi(x : A).B(x)$ is constructed by a term that, when given an x of type A, produces a term of type $B(x)$. To use a $\prod$-type term, we apply it to an argument:

$$\frac{\Gamma \vdash f : \Pi(x : A).B(x) \quad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B(a)}$$

Note the substitution in the conclusion: the type of the result is $B(a)$, obtained by substituting the actual argument a into $B(x)$.

Dependent Sum Types (Σ-types) A dependent sum type, written $\Sigma(x : A).B(x)$, represents pairs where the type of the second component depends on the value of the first component. They generalise cartesian products by allowing this dependency. The formation rule mirrors that of $\prod$-types:

$$\frac{\Gamma \vdash A : \text{Type} \quad \Gamma, x : A \vdash B(x) : \text{Type}}{\Gamma \vdash (\Sigma(x : A).B(x)) : \text{Type}}$$

To construct a σ-type term, we form pairs while respecting the dependency:

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B(a)}{\Gamma \vdash (a, b) : \Sigma(x : A).B(x)}$$

The elimination rules for σ-types involve projections. The first projection is given by:

$$\frac{\Gamma \vdash p : \Sigma(x : A).B(x)}{\Gamma \vdash \pi_1(p) : A}$$

and the second projection:

$$\frac{\Gamma \vdash p : \Sigma(x : A).B(x)}{\Gamma \vdash \pi_2(p) : B(\pi_1(p))}$$

Note the dependency in the second projection: the type of $\pi_2(p)$ depends on $\pi_1(p)$.

6.1 The Calculus of Constructions

In dependent type theory, we move beyond the limitations of the STLC by allowing types to depend on terms. This means we have terms, types, and the base judgment $t : A$. Furthermore, we have type constructors such as function types, given by $A \to B$, and importantly, dependent product types, which we have denoted by $\Pi x : A.B(x)$. In the dependent product type $\Pi x : A.B(x)$, the type $B(x)$ is not fixed but can vary depending on the term x of type A. This construction generalises the familiar function space, allowing for a collection of functions where the output type is not constant but can be determined by the specific input term. Extending our reach further still, we consider the Calculus of Constructions (CoC), which is an even more expressive type theory that builds on dependent type theory by introducing polymorphism – not only over terms but also over types themselves, and even over special kinds of organisational types called 'sorts'. In type theory, a `sort` (or universe) is a classifier for types that can contain other types. Sorts form a hierarchy where each `sort` can type the elements of lower sorts, preventing paradoxical circular typing (Stefanova and Geuvers, 1995) (bye bye Russell's paradox!). In the CoC, `Prop` and `Type_0`, `Type_1`, etc. are sorts, where each `Type_i` can contain `Type_j` for $j < i$, and `Prop` sits at the bottom of this hierarchy. The word 'sort' in CoC can be thought more generally to mean 'category of types' in the sense that it will only ever be referring to either the type `Prop` or `Type_i` for some $i \in \mathbb{N}$. The CoC is essentially a generalisation of dependent type theories, incorporating features that allow for a more unified and expressive system suitable for formalising mathematics and programming (Coquand and Huet, 1988). CoC builds upon the concept of dependent product types from dependent type theory, but makes the idea of 'type' more flexible. CoC is stratified into a hierarchy of sorts, the two fundamental sorts being denoted as `Prop` and `Type`. `Prop` is the sort of propositions, and it holds logical statements. `Type` is the sort of types, and it can hold collections of data or mathematical structures (Seldin, 1997).

The relationship between Prop and Type While appearing similar at a distance, it is important to note that `Type_0` and `Prop` are not identical —their relationship is a subtle but important one. In CoC, it is given that `Prop : Type`, or more specifically, `Prop : Type_0`. This means that `Prop` itself is considered to be a type, inhabiting the `sort Type_0` at the very bottom of the `Type` hierarchy (Geuvers, 1992). One could consider `Prop` as a kind of 'base case' for the `Type` hierarchy. `Prop` is the `sort` where propositions reside. It's the foundation upon which the purely logical part of CoC is built. Thus, while `Prop` is an element of `Type_0`, and therefore is a part of the classification of the `Type` hierarchy, it also plays a unique role as the `sort` of propositions with special impredicative properties, which is distinct from the predicative nature of `Type_0` and the broader `Type` hierarchy (Assaf, 2014). This means that propositions can be treated as types, but not vice versa. So we have that Prop is at the base of the Type hierarchy, ihabiting the `sort Type_0`, which inhabits the `sort Type_1`, which inhabits the `sort Type_2`, and so on, such that `Type_i : Type_i+1` for each $i \in \mathbb{N}$. This hierarchy of sorts is designed to dodge paradoxes when dealing with type polymorphism and quantification at higher levels (Blanqui et al., 1999).

Syntax and Semantics The terms in CoC are constructed using familiar operations from lambda calculus and dependent type theory. These include variables, lambda abstractions, and applications. CoC incorporates dependent product types, given as `(x : A) -> B`. In dependent type theory, we are accustomed to B being a type that can depend on a term x of type A. In CoC, this dependency is extended in two significant ways. First, A and B themselves can be types, not just terms (Bunder and Seldin, 2004). This allows for the formation of types that are parametrised by other types. Second, the result `(x : A) -> B` can itself be a type or a proposition. Consider the sorts of A and B. If A is of `sort` s_1 and B is of `sort` s_2, then the sort of `(x : A) -> B` is determined by a set of rules (McBride, 2000). We have that if `A : Prop` and `B : Prop`, then `(x : A) -> B : Prop`. This corresponds to logical implication and universal quantification within propositions. We also have that if `A : Sort_i` and `B : Sort_j` (where `Sort` can be either `Prop` or `Type_k` for some k), then `(x : A) -> B : Sort_{max(i, j)}`. Specifically, if s_1 and s_2 are sorts, then `(x : s_1) -> s_2` is a sort, so a dependent product type formed from sorts is itself a sort (Coquand and Huet, 1988).

The rules for forming product types and the interaction between `Prop` and `Type` lend to the expressive power of CoC as a language. Consider the rule that if `(A : Type)` and `(B : Prop)`, then `(x : A) -> B : Prop`. This allows us to express universal quantification over elements of a type within propositions (Coquand and Huet, 1988). For example, if `Nat : Type` represents the type of natural numbers and `P : Nat -> Prop` is a predicate on natural numbers, then `(x : Nat) -> P x : Prop` represents the proposition 'for all natural numbers x, $P(x)$ holds'. Furthermore, since `Prop : Type`, we can also form types that depend on propositions. For instance, we might consider a type that is defined only when a certain proposition is true. The rule that if `(A : Type)` and `(B : Type)`, then `(x : A) -> B : Type` allows for the creation of dependent function types, similar to those found in dependent type theory. However, in CoC, we can also have product types where the domain is a sort itself. For example, `(X : Type) -> List X : Type` represents the type of polymorphic list constructors, where for any type X, `List X` is the type of lists with elements of type X (Arbiser et al., 2006). The CoC is the most expressive of the typed lambda calculi before the final extension we see in the next section. Appendix gives a short exposition on the ways that these calculi can be visualised as a family, and provides a systematic way to understand how the various typing dependencies provide specific levels of expression—ultimately culminating in this system.

6.2 The Calculus of Inductive Constructions

The Calculus of Inductive Constructions (CIC) developed in the late 1980s through the work of Christine Paulin-Mohring and Thierry Coquand at INRIA (Paulin-Mohring, 1993; Coquand and Huet, 1988). The CoC was powerful enough to express ellaborate mathematical propositions through the correspondence discussed in Chapter 7, but it lacked native support for inductive definitions, requiring them to be encoded through complex impredicative encodings. The extension was thus motivated by practical needs in formal verification — the impredicative encodings of inductive types in pure CoC were unwieldy for real-world theorem proving, and so they were added in as a primitive type. The CIC has the in-built capability to define types and propositions through induction, which dramatically expands its expressiveness and practical utility for formal verification (Paulin-Mohring, 1993; Blanqui, 2003).

6.2.1 Inductive Definitions

In CoC, types and propositions are constructed using products, abstractions, and applications, along with the sort hierarchy. While powerful, this system lacks a direct way to define types or predicates based on their construction rules in a recursive or inductive manner. Inductive definitions fill this gap by allowing the specification of types and propositions by explicitly listing their constructors. A constructor is essentially a way to build elements of the type or to establish the truth of the proposition Găină et al. (2013); Paulin-Mohring (1993). Consider the natural numbers — in a system without inductive definitions, representing natural numbers and reasoning about them is cumbersome and often requires encoding them using other structures. The CIC, however, allows us to define natural numbers directly through their inductive nature. We can say that a natural number is either zero, or it is the successor of another natural number Blanqui (2003). We might structure the definition as:

```
Inductive Nat : Type :=
| zero : Nat
| succ : Nat -> Nat.
```

Here, `Inductive Nat : Type :=` declares a new inductive type named `Nat` which is a `Type`. We use the lines following `:=` to list the constructors. Here, `| zero : Nat` states that `zero` is a constructor that builds an element of type `Nat`. `| succ : Nat -> Nat` indicates that `succ` is a constructor that takes a `Nat` as input and produces another `Nat`. Intuitively, `zero` represents the number zero, and `succ` represents the successor function. Using these constructors, we can build elements of `Nat`, such as `zero`, `succ zero`, `succ (succ zero)`, and so on, representing $0, 1, 2$, etc. Beyond simple types, inductive definitions in CIC are equally powerful for defining propositions. For example, we can define the proposition that a natural number is even. This is an inductive predicate, which is a proposition parameterised by a value (in this case, a natural number). The inductive definition of `even` could be:

```
Inductive even : Nat -> Prop :=
| even_zero : even zero
| even_succ_succ : forall n : Nat, even n -> even (succ (succ n)).
```

Here, `Inductive even : Nat -> Prop :=` declares an inductive predicate `even` which takes a `Nat` and returns a `Prop`. `| even_zero : even zero` is a constructor stating that `even zero` is true (base case: zero is even). `| even_succ_succ : forall n : Nat, even n -> even (succ (succ n))` is a constructor that provides a rule: for any natural number `n`, if `even n` is true, then `even (succ (succ n))` is also true (inductive step: if `n` is even, then `n+2` is even). These constructors define how we can prove that a number is even. For instance, `even (succ (succ zero))` (i.e., even 2) can be proven using `even_succ_succ` applied to `zero` and the fact `even zero` (which we know by `even_zero`). Inductive definitions are not limited to simple examples like `Nat` and `even`. They can be parametrised and mutually recursive, allowing for the definition of complex data structures and propositions. Consider the example of polymorphic lists, which are lists that can contain elements of any type. In CIC, we can define polymorphic lists inductively, parametrised by the type of elements they contain:

```
Inductive List (A : Type) : Type :=
| nil : List A
| cons : A -> List A -> List A.
```

Here, `Inductive List (A : Type) : Type :=` defines an inductive type `List` which is parametrised by a type `A`. `| nil : List A` is a constructor representing the empty list for any type `A`. `| cons : A -> List A -> List A` is a constructor that takes an element of type `A` and a list of type `List A`, and constructs a new list of type `List A` by adding the element to the front of the list. These inductive structures are defined by the essence of their form in a generic way, applicable to be used for all different types of input. The list definition here could be used for lists of natural numbers, lists of booleans, lists of other lists, and so on.

6.2.2 Induction Principles

An important feature of inductive definitions in CIC is that they automatically come with associated induction principles. For each inductive definition, the CIC generates an induction principle that reflects the structure of the definition Paulin-Mohring (1993). These principles are useful for proving properties about inductively defined types and predicates. For the `Nat` type, the induction principle is essentially the standard mathematical induction: to prove a property $P(n)$ for all natural numbers n, we need to prove $P(\text{zero})$ (base case) and for any n, if $P(n)$ holds, then $P(\text{succ } n)$ holds (inductive step) Blanqui (2003). For the `even` predicate, the induction principle allows us to prove properties about even numbers by considering the base case (`even zero`) and the inductive step (`even_succ_succ`). Similarly, for lists, structural induction is derived from the definition of `List`. The extension from CoC to CIC is significant because it provides a way to introduce recursion and induction directly into the type theory. Without inductive definitions, CoC would be far less practical for formalising mathematics and verifying software Paulin-Mohring (1993); Găină et al. (2013).

CHAPTER 7

The Curry-Howard Correspondence

The 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 provides a formal foundation for the verification of proofs and software (Sørensen and Urzyczyn, 2006). 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. It is worth mentioning that beyond the scope of this paper, the correspondence extends to category theory, physics, and circuit design amongst others, and is a fascinating warren with many branching paths to explore. We will focus on logic and computation here (Paquette, 2009).

Propositions as Types

At the very base of this correspondence, we see that logical propositions can be interpreted as types (Wadler, 2015). Consider the basic logical connectives and their corresponding type constructors. An implication between two propositions, say $A \to B$, is understood as a function type $A \to B$. The proposition $A \to B$ is true if and only if, given a proof of A, we can construct a proof of B. Similarly, a function of type $A \to B$ is a program that, given an input of type A, produces an output of type B. The input to the function corresponds to the premise of the implication, and the output corresponds to the conclusion. Similarly, conjunction $A \wedge B$ corresponds to the product type, $A \times B$. A proposition $A \wedge B$ is true if and only if both A is true and B is true. Correspondingly, a value of product type $A \times B$ is a pair consisting of a value of type A and a value of type B. To prove $A \wedge B$, we must provide a proof of A and a proof of B, just as to construct a value of type $A \times B$, we need to provide a value of type A and a value of type B. Disjunction, $A \vee B$, corresponds to the sum type, $A + B$ (or $A \oplus B$, or variant type). The proposition $A \vee B$ is true if and only if either A is true or B is true (or both). A value of sum type $A + B$ is either a value of type A or a value of type B, along with an indication of which of the two it is. Proving $A \vee B$ requires providing a proof of A or providing a proof of B, along with a justification for the choice, analogous to constructing a value of type $A + B$. The logical constant 'true' $\top$ corresponds to the unit type, which we give as $\mathbf{1}$ or `Unit`. The proposition $\top$ is always true, and its proof is trivial (it requires no premises). The unit type $\mathbf{1}$ has exactly one value, which we denote as `()` or `unit`. This single value serves as the trivial program that inhabits the type $\mathbf{1}$, corresponding to the trivial proof of $\top$. The logical constant 'false', denoted $\bot$, corresponds to the empty type, which we give as $\mathbf{0}$ or `Void`. The proposition $\bot$ is always false, and there is no proof of $\bot$. Correspondingly, the empty type $\mathbf{0}$ has no values. This reflects the fact that there is no program that can have the empty type as its output type in a meaningful way, mirroring the impossibility of proving falsehood (Sørensen and Urzyczyn, 2006).

This initial set of correspondences forms the basis. Propositions like $\forall x \in A, P(x)$ and $\exists x \in A, P(x)$ also have type counterparts, which will be discussed later in the context of dependent types. Table 7.1 below summarises the correspondence between propositional logic and type theory:

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

Table 7.1: Correspondence between Propositional Logic and Type Theory

Proofs as Programs

Exploring the correspondence further, we see that linking propositions with types gives us that proofs of propositions can be interpreted as programs of the corresponding types (Irwin, 2008). Table 7.2 below summarises the correspondence in comparison to how one may consider the connection to set theory. Note that while similar, the set theoretical analogue is not exactly the same, but it can be helpful when building an idea of how the correspondence translates to mathematical reasoning.

Logic	**Type Theory**	**Set Theory**
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 $\perp$ (falsehood)	Empty type	Empty set

Table 7.2: Correspondence between Logic, Type Theory, and Set Theory

We inspect this connection deeper still by examining the inference rules of natural deduction and their counterparts in typed lambda calculus. For each logical inference rule, there is a corresponding operation on programs that mirrors the proof construction. The reader may note that this correspondence smells of the BHK interpretation explored in Section 2.1.1, which provides a constructive understanding of logical connectives (Sato, 1997). Consider the inference rule for implication introduction in natural deduction:

$$\begin{array}{c}[A]\\ \vdots\\ \dfrac{B}{A \rightarrow B}\quad (\rightarrow I)\end{array}$$

This rule states that if we can derive proposition A under the assumption of proposition A, then we can conclude the implication $A \to B$. In the typed lambda calculus, the corresponding rule is lambda abstraction:

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B} \quad (\to I)$$

Here, Γ represents a typing context, $x : A$ signifies that we are assuming a variable x of type A, and $M : B$ means term M has type B. This rule states that if in a context Γ extended with the assumption that x is of type A, we can derive a term M of type B, then in the original context Γ, we can form a lambda abstraction $\lambda x : A.M$ which has type $A \to B$. According to the BHK interpretation, a proof of $A \to B$ is a method that transforms any proof of A into a proof of B. Lambda abstraction precisely embodies this idea. The term M is a 'proof of B' that may depend on the assumption of A, represented by the variable x of type A. By forming the lambda abstraction $\lambda x : A.M$, we are constructing a function that takes any input of type A (representing a proof of A) and produces an output of type B (representing a proof of B). Thus, the lambda abstraction is the program-level construction that corresponds to the logical operation of implication introduction, and it directly reflects the BHK interpretation of implication (Sato, 1997).

$$\frac{A \to B \quad A}{B} \quad (\to E)$$

This rule says if we have a proof of $A \to B$ and a proof of A, we can derive a proof of B. The corresponding rule in typed lambda calculus is function application:

$$\frac{\Gamma \vdash F : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash F\,N : B} \quad (\to E)$$

This rule states: if we have a term F of type $A \to B$ and a term N of type A, then we can apply F to N to obtain a term $F\,N$ of type B. The BHK interpretation states that if we have a proof of $A \to B$ (a method to transform proofs of A to proofs of B) and a proof of A, we can obtain a proof of B by applying the method to the proof of A. Function application is the direct computational counterpart of this. The term F represents the 'proof transformation' from A to B, and N represents a proof of A. Applying F to N, denoted $F\,N$, performs the transformation and produces a term of type B, which is a proof of B. Function application, therefore, is the program operation corresponding to logical implication elimination, mirroring the BHK interpretation's application of a proof method (Sørensen and Urzyczyn, 2006). For conjunction introduction:

$$\frac{A \quad B}{A \wedge B} \quad (\wedge I)$$

This rule states that if we have a proof of A and a proof of B, we can construct a proof of $A \wedge B$. The typed rule is pairing:

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

This rule states that if we have a term M of type A and a term N of type B, we can form a pair $\langle M, N \rangle$ which has type $A \times B$. According to the BHK interpretation, a proof of $A \wedge B$ is given by providing a proof of A and a proof of B. Pairing in typed lambda calculus directly implements this. The term M is a proof of A, and N is a proof of B. Forming the pair $\langle M, N \rangle$ combines these two proofs into a single entity that represents a proof of $A \wedge B$. Thus, pairing is the program construction corresponding to logical conjunction introduction, and it directly reflects the BHK definition of conjunction proof (Sørensen and Urzyczyn, 2006). For conjunction elimination, we have two rules:

$$\frac{A \wedge B}{A} \quad (\wedge E_1) \qquad \frac{A \wedge B}{B} \quad (\wedge E_2)$$

These rules state that if we have a proof of $A \wedge B$, we can extract a proof of A or a proof of B. The corresponding typed rules are projections:

$$\frac{\Gamma \vdash P : A \times B}{\Gamma \vdash \pi_1(P) : A}\ (\pi_1) \qquad \frac{\Gamma \vdash P : A \times B}{\Gamma \vdash \pi_2(P) : B}\ (\pi_2)$$

These rules state: if we have a term P of type $A \times B$, we can extract its first component $\pi_1(P)$ of type A, and its second component $\pi_2(P)$ of type B. The BHK interpretation of conjunction states that a proof of $A \wedge B$ contains within it both a proof of A and a proof of B. Projections are the program operations that extract these component proofs from a pair. If P is a term of type $A \times B$ (representing a proof of $A \wedge B$), then $\pi_1(P)$ extracts the first component, which is a term of type A (a proof of A), and $\pi_2(P)$ extracts the second component, a term of type B (a proof of B). Projections are therefore the program operations corresponding to logical conjunction elimination, aligning with the BHK view that a proof of a conjunction inherently includes proofs of its conjuncts (Sørensen and Urzyczyn, 2006). For disjunction introduction, we have two rules:

$$\frac{A}{A \vee B}\ (\vee I_1) \qquad \frac{B}{A \vee B}\ (\vee I_2)$$

These rules state that if we have a proof of A, we can prove $A \vee B$, and similarly if we have a proof of B, we can prove $A \vee B$. The typed rules are injections, or disjunction introduction for types:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \texttt{Inl}(M) : A + B}\ (+I_1)(\texttt{Inl}) \qquad \frac{\Gamma \vdash N : B}{\Gamma \vdash \texttt{Inr}(N) : A + B}\ (+I_2)(\texttt{Inr})$$

Here, `Inl` and `Inr` are injection tags to distinguish whether we are injecting from the left type A or the right type B into the sum type $A + B$. These rules state: if we have a term M of type A, we can inject it as the left case $\texttt{Inl}(M)$ into the sum type $A + B$; if we have a term N of type B, we can inject it as the right case $\texttt{Inr}(N)$ into $A + B$. The BHK interpretation of disjunction states that a proof of $A \vee B$ is given by presenting either a proof of A or a proof of B, and indicating which one is being presented. Injections in typed lambda calculus directly represent this. If we have a proof of A (term M), then $\texttt{Inl}(M)$ is a term of type $A + B$ which represents the proof of $A \vee B$ because we have provided a proof of A and indicated (using `Inl`) that it is the proof of the left disjunct. Similarly for $\texttt{Inr}(N)$ from a proof N of B (Sato, 1997). Injections are the program operations corresponding to logical disjunction introduction, reflecting the BHK requirement to explicitly provide a proof of one of the disjuncts. For disjunction elimination:

$$\frac{A \vee B \quad \begin{matrix}[A]\\ \vdots \\ C\end{matrix} \quad \begin{matrix}[B]\\ \vdots \\ C\end{matrix}}{C}\ (\vee E)$$

This rule is more complex. It states that if we have a proof of $A \vee B$, and we know that assuming A leads to a proof of C, and assuming B also leads to a proof of C, then we can conclude C. The corresponding typed rule is case analysis, or disjunction elimination for types:

$$\frac{\Gamma \vdash L : A + B \quad \Gamma, x : A \vdash M : C \quad \Gamma, y : B \vdash N : C}{\Gamma \vdash \texttt{cases}\ \ L \text{ of } \texttt{Inl}(x) \Rightarrow M \mid \texttt{Inr}(y) \Rightarrow N : C}\ (+E)(\texttt{Cases})$$

This rule states: if we have a term L of type $A + B$, and if under the assumption $x : A$ we can derive a term $M : C$, and under the assumption $y : B$ we can derive a term $N : C$, then we can construct a case expression cases L of $\texttt{Inl}(x) \Rightarrow M \mid \texttt{Inr}(y) \Rightarrow N$ of type C. The BHK interpretation for disjunction elimination tells us that given a proof of $A \vee B$, and knowing how to obtain a proof of C from a proof of A, and also how to obtain a proof of C from a proof of B, we can conclude C. Case analysis in typed lambda calculus exactly captures this. The term L of type $A+B$ is the proof of $A \vee B$. The case expression examines which disjunct was proven. If L is of the form $\texttt{Inl}(x)$ (meaning A was proven, with proof x), we use the term M (which is a proof of C assuming A). If L is of the form $\texttt{Inr}(y)$ (meaning B was proven, with proof y), we use the term N (which is a proof of C assuming B). Since we have a way to obtain a term of type C in either case, the case expression as a whole produces a term of type C. Case analysis is thus the program operation corresponding to logical disjunction elimination, mirroring the BHK approach to handling disjunctive premises (Wadler, 2015). Finally, consider the rule for absurdity elimination (ex falso quodlibet):

$$\frac{\perp}{A} \quad (\texttt{XF})$$

If we have derived falsehood $\perp$, we can derive any proposition A. The corresponding typed rule is empty type elimination:

$$\frac{\Gamma \vdash Z : \mathbf{0}}{\Gamma \vdash \text{abort}_A(Z) : A} \quad (\mathbf{0}E)$$

This rule states: if we have a term Z of type $\mathbf{0}$ (the empty type), we can bully it into becoming any type A using an 'abort' operation, which we denote as $\text{abort}_A(Z)$. The BHK interpretation states that falsehood $\perp$ has no proof. If we were to somehow derive falsehood, it represents an impossible situation, from which anything can be derived. The abort operation in typed lambda calculus reflects this. A term of type $\mathbf{0}$ ideally should never exist in a well-typed program, as it represents a contradiction at the type level. However, if such a term Z were available, then $\text{abort}_A(Z)$ allows us to treat it as a term of any type A. This is because in a constructive system, deriving falsehood is catastrophic, and from it, any proposition can be trivially 'proven' in a vacuous sense. The `abort` operation is the program operation that corresponds to logical absurdity elimination, reflecting the BHK understanding that falsehood has no proof and implies everything. So we see that for each inference rule of natural deduction, we have identified a corresponding operation in simply typed lambda calculus. These program operations are not arbitrary; they directly mirror the proof constructions, and they precisely implement the constructive meaning of logical connectives as defined by the BHK interpretation (Sørensen and Urzyczyn, 2006). Hence we derive the 'proofs as programs' aspect of the Curry-Howard Correspondence and take a moment to admire the view before moving forward. Table 7.3 on the following page provides an overview of the connection between natural deduction and the STLC as given by the Curry-Howard Correspondence.

Rule Type	Natural Deduction (Propositional Logic)		The Simply Typed Lambda Calculus ($\lambda \rightarrow$)	
	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	$\dfrac{\Gamma \vdash M : A + B \quad \Gamma, x : A \vdash N : C \quad \Gamma, y : B \vdash P : C}{\Gamma \vdash \text{case } M \text{ of } \mathrm{inl}(x) \Rightarrow N \mid \mathrm{inr}(y) \Rightarrow P : C}\ +\mathrm{E}$
Introduction	Implication	$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}\ \rightarrow\mathrm{I}$	Function	$\dfrac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \rightarrow B}\ \rightarrow\mathrm{I}$
Elimination	Implication	$\dfrac{\Gamma \vdash A \rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \rightarrow\mathrm{E}$	Function	$\dfrac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ \rightarrow\mathrm{E}$
Introduction	Negation	$\dfrac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg\mathrm{I}$	False	$\dfrac{\Gamma, x : A \vdash M : \bot}{\Gamma \vdash \lambda x : A.M : A \rightarrow \bot}\ \bot\mathrm{I}$
Elimination	Negation	$\dfrac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot}\ \neg\mathrm{E}$	False	$\dfrac{\Gamma \vdash M : A \quad \Gamma \vdash N : A \rightarrow \bot}{\Gamma \vdash NM : \bot}\ \bot\mathrm{E}$

Table 7.3: Correspondence between Natural Deduction and the Simply Typed Lambda Calculus

Formulas-as-Types

Viewed from the perspective of formulas and types, we see that the type system of a programming language can be seen as a logic. A type declaration can be viewed as a logical assertion, and type checking becomes proof verification. If a program is well-typed, it signifies that the corresponding logical statement (its type, interpreted as a proposition) is provable, and the program itself is a constructive proof of that statement. The syntax and rules for manipulating proofs directly correspond to the syntax and rules for manipulating programs (Constable, 1980). To describe this in the form of an analogy, consider logical propositions as specifications for buildings. Here, a proposition such as 'If it rains, then the roof will not leak' is a specification for a building. A proof of this proposition is like a construction plan that guarantees this specification is met. In the Curry-Howard correspondence, the proposition 'If it rains, then the roof will not leak' is a type. A proof of this proposition (the construction plan) is a program of this type – a program that takes 'rain' as input (if it happens) and ensures 'no leak' as output. The type system aims to ensure that any program of this type will indeed satisfy the specification, just as proof verification aims to ensure that a proof is logically sound (Wadler, 2015).

Constructive Foundation

The Curry-Howard Correspondence is inherently linked to intuitionistic logic and constructive mathematics. Intuitionistic logic differs from classical logic in its treatment of disjunction and existence. In intuitionistic logic, to prove $A \vee B$, one must provide either a proof of A or a proof of B. Similarly, to prove $\exists x, P(x)$, one must provide a specific value x and a proof of $P(x)$ for that value. Classical logic, by contrast, allows for non-constructive proofs, such as proofs by contradiction that might establish existence without explicitly constructing an object (Sørensen and Urzyczyn, 2006). The BHK interpretation of logical connectives provides a constructive semantics for intuitionistic logic. As we discovered above, the BHK interpretation aligns perfectly with the 'proofs as programs' aspect of the Curry-Howard Correspondence. The descriptions of proofs in BHK are essentially descriptions of computations. This is why the correspondence naturally applies to intuitionistic logic. The STLC, and many type systems derived from it, are inherently constructive. A well-typed program in such a system is guaranteed to compute a result, and this computational nature mirrors the constructive nature of intuitionistic proofs. Classical logic, with its non-constructive aspects, does not have such a direct correspondence with standard type systems. To represent classical logic within a type-theoretic framework, one often needs to explicitly add features like classical axioms (e.g., law of excluded middle) which breaks the direct, natural correspondence seen in the intuitionistic setting (Brogi, 2021).

Extending to Dependent Type Theories

The Curry-Howard Correspondence extends significantly when we move from propositional logic and the STLC to predicate logic and more expressive type theories like dependent type theory. In predicate logic, we have quantifiers $\forall$ (for all) and $\exists$ (there exists). Dependent type theory provides type constructors that correspond to these quantifiers. For universal quantification $\forall x : A.P(x)$, where A is a type and $P(x)$ is a proposition depending on x of type A, the corresponding type constructor is the dependent product type, often written as $\prod_{x:A} B(x)$ or $(x : A) \to B(x)$, where $B(x)$ is the type corresponding to $P(x)$. A term of type $\prod_{x:A} B(x)$ is a function that, for each value x of type A, produces a value of type $B(x)$. This directly mirrors the meaning of $\forall x : A.P(x)$: for every x of type A, $P(x)$ holds. For existential quantification $\exists x : A.P(x)$, the corresponding type constructor is the dependent sum type, often written as $\sum_{x:A} B(x)$ or $(x : A) \times B(x)$. A term of type $\sum_{x:A} B(x)$ is a pair consisting of a value x of type A and a value of type $B(x)$. This corresponds to the meaning of $\exists x : A.P(x)$: there exists an x of type A for which $P(x)$ holds, and to prove it, we need to provide such an x and a proof of $P(x)$ for that x (Dowek, 2012). Extending to the

CoC and again to CoIC, propositions are indeed still types, and proofs are terms of these types. The type system of CoC is designed such that type checking corresponds precisely to proof checking in a very rich logic (McBride, 2000). Table 7.4 below summarises the connection between predicate logic and dependent type theory at the level of typing judgements, and Table 7.5 on the next page extends this to their corresponding inference rules.

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 7.4: Correspondence between Predicate Logic and Dependent Type Theory

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

Table 7.5: Correspondence between Natural Deduction for First Order Predicate Logic and the λP System with Dependent Types

Programming Language Theory and Proof Assistants

The Curry-Howard Correspondence has very directly impacted programming language theory. Dependent type systems, inspired by the correspondence, allow for the specification and verification of program behaviours, moving beyond basic type safety to functional correctness. More directly, the Curry-Howard Correspondence is the foundational principle behind the development of proof assistants and interactive theorem provers like Lean, Coq, Agda, and Idris. In these systems, users write programs that are simultaneously proofs. When a user constructs a term in these systems, the type checker verifies not only the type correctness in the programming sense, but also the logical validity of the proof. The system ensures that if a term is claimed to be a proof of a proposition (type), it is indeed a valid proof (Wadler, 2015). Peeking into the next chapter and using Lean 4 as an example, to prove a theorem, one constructs a term of the type that corresponds to the theorem. Lean's type checker then verifies that the constructed term is indeed of the specified type, thus verifying the correctness of the proof. This allows mathematicians and computer scientists to formalise mathematical theories and verify the correctness of proofs with machine assistance. The Curry-Howard Correspondence thus transforms proof construction into a form of programming and proof verification into type checking, making formal verification accessible and practical (de Moura et al., 2015). The development of these tools is a direct consequence of understanding the deep connection between logic and computation revealed by the Curry-Howard Correspondence, enabling the rigorous formalisation of mathematics and the development of verified software.

Part II

The L∃∀N Proof Assistant

CHAPTER 8

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 (de Moura et al., 2015). 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 (de Moura and Ullrich, 2021). 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 (de Moura et al., 2015).

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 (de Moura and Ullrich, 2021). 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. The kernel's operation is centered around the notion of definitional equality. When Lean is asked to check if a term `t` has a type `T`, the kernel must determine if `t : T` is a valid typing judgment. This involves not only checking the syntactic form of `t` and `T` but also considering their definitional equality. Two terms are considered definitionally equal if they reduce to the same normal form through a process of computation. This computation includes beta-reduction (applying lambda abstractions), delta-reduction (unfolding definitions), and eta-reduction (simplifying functions). The kernel implements these reduction rules and uses them to determine if two types are indeed the same, even if they are presented in different syntactic forms. Furthermore, the kernel manages the environment of definitions and axioms. When a new definition is introduced, the kernel stores it and allows it to be used in subsequent type checking through delta-reduction. Similarly, when an axiom is asserted, the kernel records it as a primitive assumption. Axioms are treated with particular care because they are points of potential inconsistency. The kernel itself does not validate the consistency of axioms, as this is generally undecidable. Instead, it relies on the user to ensure that the axioms introduced are consistent with each other and with the underlying logic. The kernel's responsibility is to faithfully apply these axioms in the type checking process, ensuring that if a proof relies on an axiom, that axiom is indeed properly accounted for (Buzzard and Massot, 2021).

The Elaboration System The elaboration system in Lean 4 is the component responsible for processing and transforming user-written Lean code into the formal, kernel-checkable terms of the underlying type theory. It acts as a bridge between the more user-friendly syntax of Lean and the strict, minimal language understood by the kernel. Elaboration is essentially a form of type inference and term reconstruction. The user may provide incomplete or high-level specifications using implicit arguments, and the elaborator's task is to fill in the missing details and resolve ambiguities to produce a fully explicit term that the kernel can then verify for type correctness. This process allows users to write Lean code that is more concise and closer to mathematical notation than would be possible if they had to write everything in the fully explicit language of the kernel, which is verbose to say the least. The main purpose of the elaboration system is to enhance the usability and expressiveness of Lean. It achieves this by automating many of the tedious and repetitive tasks associated with writing formal proofs and programs. For example, Lean allows for implicit arguments, where type parameters or function arguments can be omitted and inferred by the elaborator based on the context. This is helpful for writing code that resembles standard mathematical notation (or at least closer than it could be), where many arguments are often implicitly understood. Furthermore, the elaboration system handles features like operator overloading and coercion[1], allowing users to write code that is more natural and less cluttered with explicit type annotations and conversions. The elaborator aims to make Lean code easier to write, read, and maintain, while still ensuring that everything ultimately reduces to a term that can be verified by the kernel (de Moura and Ullrich, 2021).

Type inference is the main task of the elaborator, where it deduces the types of expressions based on the context and the types of known variables and functions. Implicit arguments are resolved by the elaborator by searching for suitable values that satisfy type constraints. Overloading resolution determines which specific function or operator is intended when multiple options are available based on the types of the arguments. Coercions are automatically inserted by the elaborator to bridge type mismatches, for example, implicitly converting a natural number to an integer when required. The elaborator also interacts closely with the kernel during this process. It repeatedly queries the kernel to check the types of partially elaborated terms and uses this information to guide the elaboration process. If the elaborator encounters an error or cannot resolve ambiguities, it reports an error to the user, indicating where and why the elaboration failed. Successfully elaborated terms are then passed to the kernel for final type checking, ensuring the overall soundness of the system (Avigad et al., 2023).

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 (de Moura and Ullrich, 2021). 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 solu-

[1]Operator overloading allows the same operator symbol (such as $+$ or $*$) to have different meanings depending on the types of its operands. Coercion is the automatic conversion of a value from one type to another (such as a natural number to an integer), typically to make operations type-correct.

tions 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 (Avigad et al., 2023). 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 (Buzzard and Massot, 2021). The remainder of this section is referenced from the documentation directly, and a combination of (de Moura and Ullrich, 2021; de Moura et al., 2015; Lean Development Team, 2024; Avigad et al., 2023; Buzzard and Massot, 2021).

CHAPTER 9

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. Extending the correspondence to Lean syntax we obtain the equivalent statements shown in Table 9.1 below:

Logic	**STLC**	**Lean 4**
$\wedge I$	(p, q)	`And.intro p q`
$\wedge E_l$	$\pi_1\ t$	`And.left t`
$\wedge E_r$	$\pi_2\ t$	`And.right t`
$\to I$	$\lambda p : P.$	`λ p :  P =>`
$\to E$	$(f\,t)$	`(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 9.1: Syntax correspondence between propositional logic, STLC, and Lean 4

We 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, so that it does not need to be declared for every statement individually.

```
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, `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`]` $\to$ 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 ∀x:α, β(x). The arrow type (→) 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 given using Sigma-types, written as `Exists` or ∃ in Lean, and `rfl` provides a proof that two definitionally equal terms are equal (more on this soon). 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`.

CHAPTER 10

Proof Terms

Proof terms are the Lego™ blocks of formal verification in Lean and it's type-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.

CHAPTER 11

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.

11.1 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_implic : (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 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. As mentioned in 10, the reflexive `rfl` tactic proves equality too, although `rfl` and `simp` operate at fundamentally different levels under the hood. The `rfl` 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 definitional 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]
    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:

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

Versus the (slightly) more verbose

```
theorem simple_explicit {p q : Prop} (h1 : P) (h2 : P → Q) : Q := by
apply h2
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.

CHAPTER 12

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. Throughout this chapter, we use the previous deocumentation references along with (Macbeth, 2021; Yan and Hanna, 2023). 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 11, 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.

12.1 The Natural Numbers

Through inductive types, we will now define the natural numbers from first principles, implement basic arithmetic operations, and prove some important properties step by step. We begin by creating an inductive type to represent the natural numbers. The definition mirrors Peano's axioms, so we see an alternative approach from a dependent type-theoretic perspective to derive the natural numbers from first principles:

```
inductive myNat where
  | zero : myNat
  | succ : myNat → myNat
  deriving Repr
```

This definition introduces two fundamental constructors: `zero`, which represents `0`, and `succ` (the successor function), which given any natural number n produces $n + 1$. The `deriving Repr` allows our type to be displayed in a readable format by enabling automatic generation of string representations for those types. To make our natural numbers compatible with Lean's built-in numerical notation, we need to create a bridge between Lean's pre-defined `Nat` type and our `myNat` type. This can be accomplished in two steps. First, we define a conversion function:

```
def natToMyNat : Nat → myNat
  | 0     => myNat.zero
  | n + 1 => myNat.succ (natToMyNat n)
```

Then, we implement the `OfNat` typeclass to allow literal numbers:

```
instance (n : Nat) : OfNat myNat n where
  ofNat := natToMyNat n
```

This allows us to write numbers such as `2`, `3` instead of `succ (succ zero)`.

Addition Addition is defined recursively based on the second argument:

```
def add (m n : myNat) : myNat :=
  match n with
  | zero   => m
  | succ n => succ (add m n)
```

This definition says that adding `zero` to `m` gives `m`, and that adding `succ n` to `m` gives `succ (m + n)`. To use the familiar + notation, we implement the `HAdd` typeclass:

```
instance : HAdd myNat myNat myNat where
  hAdd := add
```

We then prove some fundamental properties of addition. Each proof builds on previous results. First, we prove how addition interacts with `succ`:

```
theorem add_succ (m n : myNat) : m + (succ n) = succ (m + n) := by rfl
theorem succ_add (m n : myNat) : (succ m) + n = succ (m + n) := by
  induction n with
  | zero      => rfl
  | succ n ih => rw [add_succ, ih, add_succ]
```

Then, we establish properties about adding zero:

```
theorem add_zero (n : myNat) : n + zero = n := by rfl
theorem zero_add (n : myNat) : zero + n = n := by
  induction n with
  | zero      => rfl
  | succ n ih => rw [add_succ, ih]
```

Finally, we prove associativity and commutativity:

```
theorem add_assoc (a b c : myNat) : (a + b) + c = a + (b + c) := by
  induction c with
  | zero      => rw [add_zero, add_zero]
  | succ c ih => rw [add_succ, add_succ, ih, ←add_succ]

theorem add_comm (a b : myNat) : a + b = b + a := by
  induction b with
  | zero      => rw [add_zero, zero_add]
  | succ b ih => rw [add_succ, succ_add, ih]
```

Multiplication Multiplication is defined recursively in terms of addition:

```
def mul (m n : myNat) : myNat :=
  match n with
  | zero   => zero
  | succ n => add m (mul m n)

instance : HMul myNat myNat myNat where
  hMul := mul
```

We then prove essential properties of multiplication, following a similar pattern to addition:

```
theorem mul_zero (n : myNat) : n * zero = zero := by rfl
theorem mul_succ (a b : myNat) : a * (succ b) = a + (a * b) := by rfl
theorem zero_mul (n : myNat) : zero * n = zero := by
  induction n with
  | zero      => rfl
  | succ n ih => rw [mul_succ, zero_add, ih]
```

Finally, we implement exponentiation using multiplication:

```
def exp (base exponent : myNat) : myNat :=
  match exponent with
  | zero => succ zero
  | succ exponent => mul base (exp base exponent)
```

This inductive construction provides a complete foundation for natural number arithmetic, with all the essential operations and their properties formally verified. Each component builds on previous definitions and theorems, creating a mathematical structure that mirrors the way we traditionally understand natural numbers, but with the added benefit of machine-checked proofs of correctness.

Recommended Extension Some natural extensions to this construction include implementing inequalities between natural numbers, proving cancellation laws for addition and multiplication, establishing the distributive law, and proving properties of exponentiation such as $(a^m)^n = a^{(mn)}$. Next, one might practice defining division with remainder, implementing Euclidean division, and proving basic number theory results like the uniqueness of prime factorisation.

12.2 Algebraic Structures

The formalisation of abstract structures in Lean 4 can make use of both the `structure` and `type class` systems. Here we will demonstrate how to use type classes to define the algebraic hierarchy of multiplicative and addtive groups. Building on our experience with inductive types for natural numbers, we now extend to more abstract structures by defining operations and their properties as type class fields. Type classes in Lean allow us to construct a hierarchy that mirrors the mathematical progression from magmas through to groups, having the structure for both operations and axioms as computable functions and propositions respectively. The overall approach involves defining each structure as a type class that extends simpler structures, with operations and axioms expressed in dependent type theory. Lean's type class inference mechanism then automatically handles the inheritance relationships, allowing us to build ever-growing algebraic structures while maintaining consistency. We'll start with the multiplicative group structure first, as it establishes the pattern we'll follow throughout.

Multiplicative Groups First, we establish the most basic structure - a magma, which is simply a set with a binary operation:

```
class mul ( : Type u) where
  mul :  →  → 

infix : 70 "·" => mul.mul
```

This code introduces a type class `mul` parameterisd by a type α in universe level `u`. The universe level parameter allows our structure to work at any type level in Lean's hierarchy. The class contains a single field `mul` that takes two arguments of type α and returns a value of type α, representing our binary operation. The `infix` declaration creates custom notation, allowing us to write `a · b` instead of `mul.mul a b`. The precedence level `70` ensures proper parsing order in expressions with multiple operators –higher numbers bind more tightly. The unicode symbol · is used instead of `*` to avoid confusion with Lean's built-in multiplication. Next, we define a semigroup by adding the associativity property:

```
class mulSemigroup ( : Type u) extends mul  where
  mul_assoc : ∀ a b c : , (a · b) · c = a · (b · c)
```

The `extends` keyword indicates inheritance from the `mul` class. The field `mul_assoc` represents the associativity axiom as a proposition that must be proven for any instance of `mulSemigroup`. The ∀ quantifier means this must hold for all elements `a, b, c in` α. Before defining a monoid, we need a class for structures with an identity element:

```
class hasOne ( : Type u) where
  one :
```

This simple class provides a distinct element that will serve as the multiplicative identity. With this in place, we can define a monoid:

```
class mulMonoid ( : Type u) extends mulSemigroup , hasOne  where
  one_mul : ∀ a : , one · a = a
  mul_one : ∀ a : , a · one = a
```

The monoid class extends both `mulSemigroup` and `hasOne`, adding axioms that specify that the identity element behaves correctly on both left and right multiplication. Note that we need both `one_mul` and `mul_one` as separate axioms since we haven't assumed commutativity. Now we define the multiplicative group structure by extending the monoid with inverse elements:

```
class mulGroup ( : Type u) extends mulMonoid  where
  inv :  →
  mul_left_inv : ∀ a : , (inv a) · a = one
  mul_right_inv : ∀ a : , a · (inv a) = one
```

The `mulGroup` class introduces an inverse function `inv` that maps each element to its multiplicative inverse. We require both left and right inverse properties since we haven't yet assumed commutativity. The type signature `inv :` $\alpha \to \alpha$ tells Lean that `inv` is a function from α to itself. For abelian groups, we extend the group structure with commutativity:

```
class mulAbelian ( : Type u) extends mulGroup  where
  mul_comm : ∀ a b : , a · b = b · a
```

Now we enter a namespace to organise our theorems about multiplicative groups:

```
namespace mulGroup
open mul
open mulSemigroup
open hasOne
open mulMonoid
```

The `namespace` command creates a context for our theorems. The `open` commands make the operations and properties from our type classes available without qualification. This allows us to write `one` instead of `hasOne.one`, for example. Our first theorem proves that the inverse of an inverse returns the original element:

```
theorem mul_inv_of_inv ( : Type u) [mulGroup ] :
  ∀ a : , (inv (inv a)) = a := by
    intro a
    calc
    (inv (inv a)) = one · (inv (inv a)) := by
                    rw [one_mul]
    _             = (a · (inv a)) · (inv (inv a)) := by
                    rw [mul_right_inv]
    _             = (a) · ((inv a) · (inv (inv a))) := by
```

```
                    rw [mul_assoc]
    _             = a · one := by
                    rw [mul_right_inv]
    _             = a := by
                    rw [mul_one]
```

Here, the theorem statement takes a type α and requires an instance of **mulGroup** α (indicated by the square brackets). The tactic `intro a` brings the universally quantified variable into our context. The `calc` block allows us to construct an equational proof, where each step is justified by rewriting with a previously established property. The underscore indicates that intermediate expressions will be inferred. The rewrite tactic, `rw` applies equalities left-to-right. We progress with theorems about uniqueness of inverses. These theorems establish that group inverses are unique and work in both directions:

```
theorem mul_unique_inv ( : Type u) [mulGroup ] :
  ∀ a b : , (b · a = one) → b = (inv a) := by
  intros a b
  intro t
  calc
  b         = b · one := by
              rw [mul_one]
  _         = b · (a · (inv a)) := by
              rw [mul_right_inv]
  _         = (b · a) · (inv a) := by
              rw [mul_assoc]
  _         = one · (inv a) := by
              rw [t]
  _         = (inv a) := by
              rw [one_mul]
```

This theorem proves that if an element b acts as a left inverse for a, then b must equal the canonical inverse of a. The proof structure works by taking arbitrary elements a and b and a hypothesis t that $b \cdot a = one$. It then uses a calculational proof to transform b into $inva$. Each step is justified by either the monoid properties (mul_one, one_mul) or group properties (mul_right_inv). The associativity property mul_assoc allows us to rearrange the parentheses. The symmetric version proves uniqueness from the right:

```
theorem mul_inv_unique ( : Type u) [mulGroup ] :
  ∀ a b : , (a · b = one) → b = (inv a) := by
  intros a b
  intro t
  calc
  b         = one · b := by
              rw [one_mul]
  _         = ((inv a) · a) · b := by
              rw [mul_left_inv]
  _         = (inv a) · (a · b) := by
              rw [mul_assoc]
  _         = (inv a) · one := by
              rw [t]
  _         = (inv a) := by
              rw [mul_one]
```

The final multiplicative group theorem proves the ‘socks and shoes’ property –that the inverse of a product is the product of the inverses in reverse order:

```
theorem mul_shoes_and_socks ( : Type u) [mulGroup ] :
  ∀ a b : , ((inv b) · (inv a) = inv (a · b)) := by
  intros a b
  have t : ((a · b) · ((inv b) · (inv a)) = one) := by
    calc
    (a · b) · ((inv b) · (inv a)) = a · ((b · (inv b)) · (inv a)) := by
                                    rw [mul_assoc, ←mul_assoc b]
```

```
      _                                 = a · (one · (inv a)) := by
                                          rw [mul_right_inv]
      _                                 = a · (inv a) := by
                                          rw [one_mul]
      _                                 = one := by
                                          rw [mul_right_inv]
    exact mul_inv_unique  (a · b) ((inv b) · (inv a)) t
```

This proof introduces a new technique: the **have** tactic, which lets us prove an intermediate result before using it in our main proof. Here we prove that $(a \cdot b) \cdot ((invb) \cdot (inva)) = one$, and use mul_inv_unique to conclude that $(invb) \cdot (inva)$ must be the inverse of $(a \cdot b)$.

Additive Groups Moving to additive groups, we follow the same structural pattern but with different notation and terminology. First, we establish the basic additive magma structure:

```
  class add ( : Type u) where
    add :  →  →

  infix : 65 "+" => add.add
```

The structure is identical to the multiplicative case, but we use lower precedence (65 vs 70) since addition traditionally binds less tightly than multiplication. The `+` notation is more conventional for addition than the · we used for multiplication. For the additive identity element, we define:

```
  class hasNone ( : Type u) where
    zero :
```

Note that while mathematically `zero` and `one` serve analogous roles (in the sense that they are an identity element), we maintain separate type classes to preserve the distinction between additive and multiplicative structures. This separation is important for defining more ellaborate algebraic structures like rings later, and it also just helps us to think about the structures better with the conventional notation for each structure (i.e. `one` for multiplication and `zero` for addition). The additive semigroup structure introduces associativity:

```
  class addSemigroup ( : Type u) extends add  where
    add_assoc : ∀ a b c : , (a + b) + c = a + (b + c)
```

The additive monoid combines the semigroup with the `zero` element:

```
  class addMonoid ( : Type u) extends addSemigroup , hasNone  where
    zero_add : ∀ a : , zero + a = a
    add_zero : ∀ a : , a + zero = a
```

The additive group introduces the negation operation for inverses:

```
  class addGroup ( : Type u) extends addMonoid  where
    neg :  →
    neg_add : ∀ a : , (neg a) + a = zero

  prefix: 60 "-" => addGroup.neg
```

The prefix notation for negation uses precedence 60, ensuring that $-a + b$ is parsed as $(-a) + b$ rather than $-(a + b)$. We create a namespace for additive group theorems:

```
  namespace addGroup
  open add
  open addSemigroup
  open hasNone
  open addMonoid
```

Now we will define some basic theorems for additive groups. Like in the multiplicative case, the below `open` commands make the relevant operations and properties available without qualification within our theorem proofs. The namespace keeps our additive group theorems organised and separate from their multiplicative counterparts.

```
    namespace addGroup
    open add
    open addSemigroup
    open hasNone
    open addMonoid
```

We first demonsrate that the addition of an element and its inverse culminates to `zero` (analogous to the right inverse property for multiplicative groups). In Lean, we encode this as:

```
theorem add_neg ( : Type u) [addGroup ] :
  ∀ a : , a + (- a) = zero := by
  intro a
  calc
  a + (- a) = zero + (a + (- a)) := by
              rw [zero_add]
  _         = ((- (- a)) + (- a)) + (a + (- a)) := by
              rw [neg_add]
  _         = (- (- a)) + ( (- a) + (a + (- a))) := by
              rw [add_assoc]
  _         = (- (- a)) + (zero + (- a)) := by
              rw [←add_assoc (- a), neg_add]
  _         = (- (- a)) + (- a) := by
              rw [zero_add]
  _         = zero := by
              rw [neg_add]
```

The theorem statement uses type parameters similar to our multiplicative group theorems. The square brackets around `[addGroup` α`]` indicate a type class instance requirement –Lean must know that α has an additive group structure. The proof uses the calculational style with the `calc` block, where each step is justified by rewriting with previously established properties. Note the use of ← in one step to apply the associativity property in reverse. Next, we prove that inverses are unique in additive groups:

```
theorem add_inv_unique ( : Type u) [addGroup ] :
  ∀ a b : , (a + b = zero) → b = (- a) := by
  intros a b
  intro t
  calc
  b = zero + b := by
      rw [zero_add]
  _ = ((- a) + a) + b := by
      rw [neg_add]
  _ = (- a) + (a + b) := by
      rw [add_assoc]
  _ = (- a) + zero := by
      rw [t]
  _ = (- a) := by
      rw [add_zero]
```

This theorem shows that if adding two elements gives zero, then one must be the negative of the other. The theorem takes three arguments: the type α, an instance of addGroup for α, and a proof that `a + b = zero`. The conclusion states that b must equal $-a$. The proof then uses `calc` again, with each step justified by properties from our additive group structure. The `intros` tactic brings both variables into scope, while `intro t` names our hypothesis that `a + b = zero`. The next theorem proves uniqueness from the other direction:

```
theorem add_unique_inv ( : Type u) [addGroup ] :
  ∀ a b : , (b + a = zero) → b = (- a) := by
  intros a b
  intro t
  calc
  b   = b + zero := by
        rw [add_zero]
  _   = b + (a + (- a)) := by
        rw [add_neg]
  _   = (b + a) + (- a) := by
        rw [add_assoc]
  _   = zero + (- a) := by
        rw [t]
  _   = (- a) := by
        rw [zero_add]
```

This theorem mirrors `add_inv_unique` but handles the case where b is a left inverse of a. In Lean's encoding, we need both theorems because we haven't yet shown or assumed commutativity. The proof structure follows a similar pattern, using the `calc` block to chain together equalities. Note how Lean's tactic framework allows us to reference our hypothesis `t` directly in the rewrite step. The next theorem proves that the negative of a negative returns the original element:

```
theorem add_inv_of_inv ( : Type u) [addGroup ] :
  ∀ a : , (- (- a)) = a := by
  intro a
  calc
  (- (-a)) = zero + (- (-a)) := by
             rw [zero_add]
  _         = (a + (- a)) + (- (-a)) := by
             rw [add_neg]
  _        = a + ((- a) + (- (- a))) := by
             rw [add_assoc]
  _        = a + zero := by
             rw [add_neg]
  _        = a := by
             rw [add_zero]
```

This theorem demonstrates how Lean handles nested applications of the negation operation. The proof uses the additive group properties we've established to show that double negation cancels out. Notice how Lean's type system ensures that expressions like `- (-a)` are well-formed when we have an additive group structure. Finally, we prove the additive version of the 'socks and shoes' theorem:

```
theorem add_shoes_and_socks ( : Type u) [addGroup ] :
  ∀ a b : , (- b) + (- a) = (- (a + b))  := by
  intros a b
  have t : ((a + b) + ((- b) + (- a)) = zero) := by
    calc
    (a + b) + ((- b) + (- a)) = a + (b + ((- b) + (- a))) := by
                                rw [add_assoc]
    _                         = a + ((b + (- b)) + (- a)) := by
                                rw [add_assoc]
    _                         = (a) + (zero + (- a)) := by
                                rw [add_neg]
    _                         = (a) + (- a) := by
                                rw [zero_add]
    _                         = zero := by
                                rw [add_neg]
  exact add_inv_unique  (a + b) ((- b) + (- a)) t
```

This theorem shows that the negative of a sum equals the sum of the negatives in reverse order. The proof introduces an auxiliary fact using the `have` tactic, proving that $(a + b) + ((-b) + (-a)) = zero$. This intermediate result is then used with `add_inv_unique` to establish the main theorem. Note how Lean's `exact` tactic allows us to apply `add_inv_unique` with explicit type and term arguments. Next in our hierarchy comes the definition of Abelian groups, which adds commutativity to our additive group structure:

```
class addAbelian ( : Type u) extends addGroup  where
  add_comm : ∀ a b : , a + b = b + a
```

In this code structure, the `addAbelian` type class extends `addGroup`, adding a single new field `add_comm`. This field represents the commutativity axiom as a proposition that must hold for all elements a and b in our type α. The universal quantification `∀ a b :` α tells Lean that this property must hold for any pair of elements we might choose.

Instantiation With both multiplicative and additive groups structurally defined, one might wonder what to do with them. We will now examine the process of implementing type class objects in Lean. The `instance` keyword processes pre-defined type class definitions as a blueprint for which properties any object of that classification must have. When we instanciate the cyclic group C_2, for example, we're creating a specific implementation that satisfies the requirements made by the declared `type class`. This is similar to how in mathematics we first define what a group is axiomatically, then show that specific objects (such as C_2) satisfy those axioms. In Lean, we do this by first inductively defining what C_2 is, then providing instances that implement each required operation and prove each required property. Each instance declaration is effectively a proof that C_2 satisfies part of the group axioms, culminating in a complete verification that C_2 is indeed a group under our defined operations. To demonstrate, we begin with our inductive definition:

```
inductive C2 where
| zero : C2
| one : C2
deriving Repr
```

This definition uses Lean's inductive type system to create a type with exactly two elements, named `zero` and `one`. The `deriving Repr` we saw earlier command tells Lean to automatically generate code for displaying these values as strings. After defining the type itself, we need to provide instances of our type classes to give C_2 its group structure. We start with the additive identity:

```
instance : hasNone C2 where
  zero := C2.zero
```

This instance declaration tells Lean that C_2 has an additive identity element. The `:=` syntax maps the `zero` field from our `hasNone` type class to the `C2.zero` constructor we defined in our inductive type. This establishes which element serves as the additive identity. Next comes the implementation of addition:

```
instance : add C2 where
  add := fun a b =>
    match a, b with
    | C2.zero, x => x
    | x, C2.zero => x
    | C2.one, C2.one => C2.zero
```

This instance provides the addition operation for C_2. The implementation uses pattern matching to define how elements combine. When either argument is `zero`, the result is the other element –this encodes that zero is the identity element. When both arguments are `one`, the result is `zero` –this encodes that $1 + 1 = 0$ in modulo 2 arithmetic. The pattern matching syntax in Lean is a clear way to specify all possible combinations of inputs when the options are limited, but far too verbose and barbaric for many combinations. It also isn't exceptionally elegant on that front, but it works. Moving to prove the associativity of addition:

```
instance : addSemigroup C2 where
  add_assoc := by
```

```
    intros a b c
    cases a ; cases b ; cases c ; rfl
```

This instance proves that C_2 forms a semigroup. The proof uses case analysis with the `cases` tactic to handle all possible combinations of elements. The `<;>` operator chains tactics, applying the subsequent tactics to all subgoals generated by the previous tactic. Since C_2 has only two elements, this generates eight cases total. The `rfl` tactic proves each case by showing that both sides of the equation reduce to the same value. For the monoid structure:

```
instance : addMonoid C2 where
  zero_add := by
    intro a
    cases a ; rfl
  add_zero := by
    intro a
    cases a ; rfl
```

This instance proves that C_2 satisfies the monoid axioms. We need to prove both `zero_add` $(0 + a = a)$ and `add_zero` $(a + 0 = a)$ since we haven't assumed commutativity yet. Each proof follows the same pattern: introduce an arbitrary element a, perform case analysis on it, and prove each case by reduction.
We now extend this even further to claim that C_2 is not just an additive monoid, but an additive group:

```
instance : addGroup C2 where
  neg := fun a =>
    match a with
    | C2.zero => C2.zero
    | C2.one => C2.one
  neg_add := by
    intro a
    cases a ; rfl
```

This instance defines the negation operation and proves it satisfies the group axioms. In C_2, each element is its own inverse (since we're working modulo 2). The proof of `neg_add` again uses case analysis –since both elements are self-inverse, the property holds trivially in both cases.

```
instance : addAbelian C2 where
  add_comm := by
    intros a b
    cases a ; cases b ; rfl
```

Finally, we prove C_2 is abelian. The proof performs case analysis on both inputs and shows that $a+b = b+a$ in each case. Since C_2 has two elements, this generates four cases, each proved by reduction. These instances collectively build up to show that C_2 satisfies all the axioms of an abelian group. Each instance adds another layer of structure, progressively proving that C_2 meets all requirements of our type class definitions.

12.3 Mathlib

Transitioning 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). For example, 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.

This document presents the mathematical foundations of theorem proving from formal logic and computability to their implementation in Lean 4. Through an examination of logical frameworks and the Curry-Howard Correspondence, we established the theoretical basis of modern proof assistants. Proceeding from Natural Deduction through lambda calculi to Dependent Type Theory and the Calculus of Inductive Constructions, we connected logic with computation like Ms. Frizzle taking her students aboard the Magic School Bus. Our analysis of Lean 4 demonstrated the theories in implementation, and gave a grounding for formalising mathematics in the language. Through examination of proof terms, tactics, and syntax, we connected abstract mathematics with concrete computation. The formalisation of mathematical structures shows us one of the applications of machine-verified proofs, among many. The connection between mathematics and computation in these systems provides the basis for research in verification, proof theory, and computational logic. It’s a wild world of out there, and we just touched the surface of it (AI, 2025).

Appendices

The Lambda Cube

The lambda Cube, introduced by Henk Barendregt (1991), is a three-dimensional framework used to classify a family of typed lambda calculi based on the kinds of abstraction they permit. It provides a structured way to understand how different systems extend the basic notion of function abstraction found in the untyped lambda calculus by adding type-level abstractions. The cube is not a lambda calculus itself, it is a meta-structure that organises eight different typed lambda calculi at its vertices. 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 is the theoretical foundation for the proof assistant Lean (Coquand and Huet, 1988). Figure 1 below gives a visual representation of the cube.

$$\begin{tikzcd}
 & \lambda\omega \equiv \lambda 2\underline{\omega} \arrow[rr, dash] \arrow[dl, dash] \arrow[dd, dash] & & \lambda C \equiv \lambda P\omega \arrow[dl, dash] \arrow[dd, dash] \\
\lambda 2 \arrow[rr, dash, crossing over] \arrow[dd, dash] & & \lambda P2 & \\
 & \lambda\underline{\omega} \arrow[rr, dash] \arrow[dl, dash] & & \lambda P\underline{\omega} \arrow[dl, dash] \\
\lambda\rightarrow \arrow[rr, dash] & & \lambda P \arrow[from=uu, dash, crossing over] &
\end{tikzcd}$$

Figure 1: The Lambda Cube

.0.1 Dimensions of the Cube

The three dimensions of the Lambda Cube represent three fundamental forms of dependency or abstraction within type theory. The first dimension, often denoted as the λ axis or the 'terms depending on terms' axis, distinguishes between systems that allow function abstraction over terms. All systems in the cube inherently possess this, as they are all lambda calculi. However, this axis is still relevant in differentiating systems when considered in combination with other dimensions. The second dimension, referred to as the Π or 'types depending on terms' axis, introduces dependent types. Moving along this axis allows types to depend on values. This means we can have type families indexed by terms. For example, we could define a type `Vector(n)` where n is a term representing the length of the vector. Systems along this axis provide the ability to express more specific type constraints and are at the ground floor of constructive type theory. The third dimension, the ω or 'types depending on types' axis, introduces polymorphism and type operators. Moving along this axis allows types to be abstracted over types. This enables the definition of polymorphic functions and type constructors. For example, we can define a polymorphic identity function that works for any type, or a type constructor `List(T)` that creates a list type for any type T. Systems along this axis increase the expressiveness in terms of generic programming and type-level computation.

.0.2 Vertices of the Cube

The Lambda Axis

Each vertex of the Lambda Cube corresponds to a specific typed lambda calculus, determined by which of these dependency dimensions are enabled. The simplest system, located at the origin $(0, 0, 0)$, is the STLC $(\lambda \rightarrow)$. This system only supports term abstraction; types are simple and fixed, and do not depend on terms or other types. Continuing from the STLC $(\lambda \rightarrow)$, we can explore the systems obtained by enabling different forms of dependency. We first consider adding polymorphism, which corresponds to moving along the λ axis to the vertex $(1, 0, 0)$. This leads to `System F`, also known as the Polymorphic Lambda Calculus or $\lambda 2$ (Reynolds, 1974). In System F, we introduce the capability for terms to depend on types. This is achieved through type abstraction and type application. We can abstract over types in terms, creating polymorphic functions. For example, we can define a polymorphic identity function that works for any type. This is written using a type abstraction, which we will denote with a capital Lambda (Λ). So, the polymorphic identity function in System F could be defined as $\Lambda A.\lambda x : A.x$. Here, ΛA abstracts over a type variable A, and the function $\lambda x : A.x$ is the identity function for a specific type A. To use this polymorphic function, we need to apply it to a type, for example, $(\Lambda A.\lambda x : A.x)[Int]$ would be the identity function specifically for integers.

The Pi Axis

We now consider movement along the Π axis from the STLC, which corresponds to the vertex $(0, 1, 0)$. This gives us `System` $\lambda\Pi$, which we will refer to as the Dependent Types Lambda Calculus (Yiyun, 2024). In this system we introduce dependent product types, which are types that depend on terms, and use the Π notation to denote these dependent function types. As discussed in Section 6, a dependent function type $\Pi x : A.B(x)$ indicates a function that takes an argument x of type A, and returns a result of type $B(x)$, where the type B can depend on the value of x. The classic example is the type of vectors of a certain length. We could have a type `Vector(n)` indexed by a natural number n. A function that constructs a vector of length n might then have a type $\Pi n : Nat.Vector(n)$ (Bauer, 2022). This is a palpable increase in expressiveness as it allows us to express properties of programs within the type system itself, providing more precision for type checking and verification.

The Omega Axis

Next, we consider motion along the ω axis from the STLC, reaching the vertex $(0, 0, 1)$. This leads to `System` $\lambda\omega$, also known as Higher-Order Polymorphic Lambda Calculus or System Fω. In System Fω, we introduce type operators, which are essentially functions at the type level (Nguyen, 2022). This means types can depend on other types in a more structured way, allowing us to construct new types from existing ones. For example, we can define a type constructor `List` that takes a type A and produces the type `List(A)`. This is different from simple polymorphism where we abstract over types in terms; here, we are abstracting over types to build new types. Type operators are used for representing concepts like parameterised data structures and for type-level computation (Yallop, 2018). System Fω allows for higher-order types, meaning types can be applied to types and produce types, leading to a richer type system compared to System F (Girard, 1972) and System $\lambda\Pi$ considered individually. It provides the ability to work with type constructors and is a basic necessity for type system properties found in languages that support generics and type families (Chapman, 2019).

Combining Dependency Dimensions

We have explored systems obtained by enabling each dimension individually starting from the STLC. Now we will consider enabling combinations of these features. Take System F ($\lambda 2$), which has polymorphism (terms depending on types) (Girard, 1972), and adding dependent types (types depending on terms). This means we are moving along the Π axis from System F. Alternatively, we can start with System $\lambda\Pi$, which has dependent types, and add polymorphism (terms depending on types) by moving along the λ axis. Both of these approaches lead to the same system, which is the system at vertex $(1, 1, 0)$ of the Lambda Cube. This system is often referred to as `System FΠ` or $\lambda\Pi 2$, but is perhaps less commonly named than others in the cube. System FΠ combines the features of both System F and System $\lambda\Pi$. It allows for both polymorphic functions (functions that can work with different types through type abstraction) and dependent types (types that can depend on term values). This system is significantly more expressive than either System F or System $\lambda\Pi$ alone. For example, in System FΠ, you can define polymorphic functions that operate on dependent data structures, or define dependent types that are themselves parameterized by types (Nguyen, 2022). This combination provides a powerful framework for expressing complex relationships between terms and types.

Combining Polymorphism Dimensions

Next, we consider combining polymorphism (terms depending on types) from System F and type operators (types depending on types) from System Fω. Starting from System Fω and adding polymorphism (terms depending on types) by moving along the λ axis, or starting from System F and adding type operators (types depending on types) by moving along the ω axis, both lead to the system at vertex $(1, 0, 1)$. We denote this system as System Fω as well, although sometimes explicitly denoted as $\lambda 2\omega$ with reference to the combination of features. The name System Fω is predominantly used, and context usually clarifies whether it refers to the system with just type operators (vertex $(0, 0, 1)$) or the system with both polymorphism and type operators (vertex $(1, 0, 1)$). In this expanded System Fω (vertex $(1, 0, 1)$), we have both the ability to abstract terms over types (polymorphism as in System F) and the ability to define type operators (as in System Fω at vertex $(0, 0, 1)$). This allows for even more expressive type-level programming (Nguyen, 2022). We can have polymorphic type operators, and we can use type operators to construct the types that polymorphic functions operate on.

The King of the Castle

Finally, we consider the system at the corner of the cube (vertex (1,1,1)), which is obtained by enabling all three forms of dependency: terms depending on terms, types depending on terms, and types depending on types. This system is the Calculus of Constructions (CoC), which we denote as λC. CoC can be seen as extending System FΠ (or System Fω) by adding the remaining dependency feature. Equivalently, starting from any of the systems at vertices $(1, 1, 0)$ or $(1, 0, 1)$ and enabling the last remaining dependency feature will lead to CoC. The Calculus of Constructions is a very expressive type theory that brigs together higher-order polymorphism, dependent types, and type operators in a single framework (Coquand and Huet, 1988). In CoC, types can depend on terms, terms can depend on types, and types can depend on types, and importantly, these dependencies can be combined and nested in very expressive ways. CoC is the most expressive system within the Lambda Cube. It forms the basis for proof assistants and programming languages. It is expressive enough to encode higher-order logic and is the foundation for systems like the Calculus of Inductive Constructions (CoIC), which further extends CoC with inductive types and is the basis of the Lean proof assistant.

The Lambda Cube provides a structured way to understand the relationships between different typed lambda calculi (Sørensen and Urzyczyn, 2006). Moving along each axis adds a new form of abstraction, increasing the expressiveness of the system. Starting from the STLC, we can reach increasingly expressive systems by adding polymorphism, dependent types, and type operators, culminating in the Calculus of Constructions, which combines all of these features. Each system in the cube builds upon the previous ones, offering a spectrum of expressiveness suitable for different applications in logic, programming languages, and formal verification.

Bibliography

Aberdein, A. (2007). The Informal Logic of Mathematical Proof. In Van Kerkhove, B. and Van Bendegem, J. P., editors, *Perspectives on Mathematical Practices*, pages 135–151. Springer, Dordrecht.

Aczel, P. (1999). Notes on the Simply Typed Lambda Calculus. In Sambin, G. and Smith, J. M., editors, *Types and Computational Properties of Lambda Calculi*, pages 1–15. Springer, Berlin, Heidelberg.

Adams, R. (2006). Pure type systems with judgemental equality. *Journal of Functional Programming*, 16(2):219–246.

AI (2025). Ai assistance in research and diagram generation. Used as a tool for research and for generating diagrams. AI was used as a tool in the research process and for the creation of diagrams in this work.

Arbiser, A., Miquel, A., and Ríos, A. (2006). A lambda-calculus with constructors. In Hermann, M. and Voronkov, A., editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 4246 of *Lecture Notes in Computer Science*, pages 162–177. Springer, Berlin, Heidelberg.

Aschieri, F. and Zorzi, M. (2016). On natural deduction in classical first-order logic: Curry-Howard correspondence, strong normalization and Herbrand's theorem. *Theoretical Computer Science*, 625:125–146.

Assaf, A. (2014). A calculus of constructions with explicit subtyping. In *Types for Proofs and Programs*.

Avigad, J., de Moura, L., and Kong, S. (2023). *Theorem Proving in Lean 4*. Lean Community, Online.

Barendregt, H. (1991). Lambda calculi with types. *Handbook of Logic in Computer Science*, 2:117–309.

Barendregt, H. (2012). *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam.

Barendregt, H. (2013). *Lambda Calculus with Types*. Number 290 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge.

Barthe, G. and Coquand, T. (2000). An introduction to dependent type theory. In Buss, S. R., Hájek, P., and Pudlák, P., editors, *Proof Theory and Logical Complexity*, volume 235 of *Synthese Library*, pages 1–25. Springer Netherlands, Dordrecht.

Barthe, G. and Elbers, H. (1996). Towards Lean Proof Checking. In Berardi, S. and Coppo, M., editors, *Types for Proofs and Programs*, pages 27–44. Springer, Berlin, Heidelberg.

Bauer, A. (2022). Dependent types lecture notes.

Berline, C. (2000). From computation to foundations via functions and application: The λ-calculus and its webbed models. *Theoretical Computer Science*.

Besnard, P. (1989). First Order Logic. In *An Introduction to Default Logic*, pages 41–67. Springer, Berlin, Heidelberg.

Bjesse, P. (2005). What is formal verification. *ACM Sigda Newsletter*, 35(2):1–4.

Blanqui, F. (2003). Inductive types in the calculus of algebraic constructions. In *Typed Lambda Calculi and Applications*, volume 2701 of *Lecture Notes in Computer Science*, pages 46–60. Springer.

Blanqui, F., Jouannaud, J.-P., and Okada, M. (1999). The calculus of algebraic constructions. In Jouannaud, J.-P., editor, *Typed Lambda Calculi and Applications*, volume 1581 of *Lecture Notes in Computer Science*, pages 20–44. Springer, Berlin, Heidelberg.

Boffa, M. (1984). Arithmetic and the Theory of Types. *Journal of Symbolic Logic*, 49(4):1348–1355.

Brogi, C. P. (2021). Curry–Howard–Lambek Correspondence for Intuitionistic Belief. *Studia Logica*, 109(6):1313–1346.

Bunder, M. W. and Seldin, J. P. (2004). Variants of the basic calculus of constructions. *Journal of Applied Logic*.

Buzzard, K. and Massot, P. (2021). *Functional Programming in Lean*. Lean Community, Online. Online book.

Chapman, J. (2019). *System F in Agda, for Fun and Profit*. PhD thesis, University of Edinburgh.

Constable, R. L. (1980). Programs and types. In *21st Annual Symposium on Foundations of Computer Science*, pages 118–128. IEEE.

Constable, R. L. (1991). Type Theory as a Foundation for Computer Science. In Takayasu, I. and Meyer, A. R., editors, *Theoretical Aspects of Computer Software*, pages 409–423. Springer, Berlin, Heidelberg.

Copeland, B. J. (1996). What is computation. *Synthese*, 108(3):335–359.

Coquand, T. and Huet, G. (1988). The calculus of constructions. *Information and Computation*, 76(2-3):95–120.

Crossley, J. N. (2011). What Is Mathematical Logic? A Survey. In Schwichtenberg, H. and Wainer, S., editors, *Proof and Computation*, pages 3–27. Springer, Berlin, Heidelberg.

Culik, K. (1983). On formal and informal proofs for program correctness. *Sigplan Notices*, 18(8):39–47.

de Moura, L., Kong, S., Avigad, J., van Doorn, F., and von Raumer, J. (2015). The Lean Theorem Prover (System Description). In Felty, A. P. and Middeldorp, A., editors, *Automated Deduction - CADE-25*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388, Berlin, Heidelberg. Springer.

de Moura, L. and Ullrich, S. (2021). The Lean 4 Theorem Prover and Programming Language. In Platzer, A. and Sutcliffe, G., editors, *Automated Deduction - CADE-28*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635, Cham. Springer.

Dietrich, D. and Buckley, M. (2007). Verification of Proof Steps for Tutoring Mathematical Proofs. In Luckin, R., Koedinger, K. R., and Greer, J., editors, *Artificial Intelligence in Education: Building Technology Rich Learning Contexts That Work*, pages 560–562, Amsterdam. IOS Press.

Dixon, L. and Fleuriot, J. (2006). A proof-centric approach to mathematical assistants. *Journal of Applied Logic*, 4(4):505–532.

Dowek, G. (2012). A theory independent curry-de bruijn-howard correspondence. In Beringer, L. and Felty, A., editors, *Interactive Theorem Proving*, pages 19–33. Springer, Berlin, Heidelberg.

Duggan, D. and Bent, F. (1996). Explaining type inference. *Science of Computer Programming*, 27(1):37–83.

Díez, G. F. (2000). Kolmogorov, heyting and gentzen on the intuitionistic logical constants. *Critica-Revista Hispanoamericana De Filosofia*, 32(96):43–57.

Emerich, J. (2016). How are programs found? speculating about language ergonomics with Curry-Howard. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2016, pages 63–78, New York, NY. ACM.

Fairtlough, M. and Mendler, M. (2000). On the Logical Content of Computational Type Theory: A Solution to Curry's Problem. In Berger, U. and Schwichtenberg, H., editors, *Types and Proofs in Computing*, pages 63–78. Springer, Berlin, Heidelberg.

Fischer, M. J. (1972). Lambda calculus schemata. In *Proceedings of a conference on Proving assertions about programs*, SIGPLAN Notices, pages 107–119, New York, NY, USA. ACM.

Găină, D., Zhang, M., Chiba, Y., and Arimoto, Y. (2013). Constructor-based inductive theorem prover. In *Automated Technology for Verification and Analysis*, volume 8172 of *Lecture Notes in Computer Science*, pages 328–332. Springer.

Garner, R. (2009). On the strength of dependent products in the type theory of Martin-Löf. *Annals of Pure and Applied Logic*.

Garofalo, J., Trinter, C. P., and Swartz, B. A. (2015). Engaging with Constructive and Nonconstructive Proof. *Mathematics Teacher: Learning and Teaching PK–12*, 108(6):422–428.

Geuvers, J. H. (1992). *The calculus of constructions and higher order logic.* PhD thesis, University of Nijmegen.

Geuvers, J. H. and Nederpelt, R. P. (1994). Typed lambda-calculus. In Gabbay, D. M., Hogger, C. J., and Robinson, J. A., editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 367–468. Elsevier Science B.V.

Girard, J.-Y. (1972). System f. *Archive for Mathematical Logic*, 12(1):55–65.

Grue, K. (2001). λ-Calculus as a Foundation for Mathematics. In Troelstra, A. S., Schwichtenberg, H., and Wainer, S. S., editors, *Logic and Foundations of Mathematics*, volume 292 of *Synthese Library*, pages 255–286. Springer Netherlands, Dordrecht.

Hindley, J. R. (1997). *Basic Simple Type Theory*, volume 42 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge.

Hofmann, M. (1994). Elimination of extensionality in Martin-Löf type theory. In Barendregt, H. and Nipkow, T., editors, *Types for Proofs and Programs*, pages 166–183. Springer, Berlin, Heidelberg.

Humberstone, L. and Makinson, D. (2011). Intuitionistic logic and elementary rules. *Mind*, 120(480):1035–1051.

Ihlemann, C., Jacobs, S., and Sofronie-Stokkermans, V. (2008). On local reasoning in verification. In Ramakrishnan, C. R. and Rehof, J., editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 265–281. Springer, Berlin, Heidelberg.

Indrzejczak, A. (2010). Standard Natural Deduction. In Artemov, S. and Fitting, M., editors, *Natural Deduction, Hybrid Systems and Modal Logics*, pages 21–38. Springer, Dordrecht.

Ireland, A. (1993). On Exploiting the Structure of Martin-Löf's Theory of Types. In Basin, D. and Giunchiglia, F., editors, *Automated Deduction in Nonstandard Logics*, pages 245–260. Springer, Berlin, Heidelberg.

Irwin, R. J. (2008). Review of “Derivation and Computation: Taking the Curry-Howard Correspondence Seriously by Harold Simmons,” Cambridge University Press, 2000. *Sigact News*, 39(1):25–27.

Kazmierczak, E. (1991). *Algebraic reasoning in lambda calculus.* University of Sussex.

Kurokawa, H. and Kushida, H. (2013). Substructural Logic of Proofs. In Libkin, L., Kohlenbach, U., and de Queiroz, R., editors, *Logic, Language, Information, and Computation*, pages 181–193. Springer, Berlin, Heidelberg.

Lean Development Team (2024). *Lean 4 Reference Manual.* Lean Community.

Li, D. (1992). A Natural Deduction Automated Theorem Proving System. In Kapur, D., editor, *Automated Deduction - CADE-11*, pages 728–732. Springer, Berlin, Heidelberg.

Li, W. (1989). A Type-Theoretic Approach to Program Development. In *IFIP Congress*, Amsterdam. North-Holland.

Longo, G. (2011). Theorems as Constructive Visions. In *Foundations of the Formal Sciences VII*, pages 47–72. Springer, Dordrecht.

Macbeth, H. (2021). *Mathematical Proofs: A Transition to Advanced Mathematics.* GitHub, Online.

Machado, R. (2013). An Introduction to Lambda Calculus and Functional Programming. In *Workshop-School on Theoretical Computer Science.*

Markov, A. A. (1968). An Approach to Constructive Mathematical Logic. In Lakatos, I., editor, *Studies in Logic and the Foundations of Mathematics*, pages 283–294. North-Holland, Amsterdam.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall, Upper Saddle River, NJ.

Martin, R. M. (1943). A Homogeneous System for Formal Logic. *Journal of Symbolic Logic*, 8(1):1–23.

McBride, C. (2000). *Dependently Typed Functional Programs and their Proofs.* PhD thesis, University of Edinburgh.

McCawley, J. D. (1991). Natural Deduction and Ordinary Language Discourse Structure. In Cooper, R., Mukai, K., and Perry, J., editors, *Situation Theory and its Applications*, pages 253–272. CSLI Publications, Stanford.

McKinna, J. (2006). Why dependent types matter. In *Symposium on Principles of Programming Languages.*

Mella, P. (2012). The Kingdom of Circular Processes: The Logical Foundations of Systems Thinking. In Minati, G., Abram, M., and Pessa, E., editors, *Systems Thinking in Practice*, pages 3–30. Springer, Milan.

Merz, S. (1997). Rules for Abstraction. In Gordon, A. D. and Pitts, A. M., editors, *Higher Order Operational Techniques in Semantics*, pages 99–116. Cambridge University Press, Cambridge.

Nederpelt, R. R. and Geuvers, H. (2014). *Type Theory and Formal Proof: An Introduction.* Cambridge University Press, Cambridge.

Newen, A., Nortmann, U., and Stuhlmann-Laeisz, R. (2001). *Building on Frege: New essays about sense, content and concepts.* CSLI Publications, Stanford, CA.

Nguyen, M. (2022). Lambda calculus – λ, system f, and system fω. Personal webpage.

Nowak, S. (1977). Kinds of Propositions. In Przelecki, M. and Wojcicki, R., editors, *Studies in Logic and Scientific Reasoning*, pages 71–89. Springer, Dordrecht.

O'Regan, G. (2012). History of Programming Languages. In *A Brief History of Computing*, pages 149–166. Springer, London.

Ou, X., Tan, G., Mandelbaum, Y., and Walker, D. (2004). Dynamic typing with dependent types. In Dybjer, P., Nordström, B., and Smith, J., editors, *Types in Logic and Type Theory*, volume 228 of *NATO Science Series II Computer and Systems Sciences*, pages 557–582. Kluwer Academic Publishers, Dordrecht.

Paquette, D. (2009). *Categorical quantum computation.* PhD thesis, McGill University.

Paulin-Mohring, C. (1993). Inductive definitions in the system coq - rules and properties. In *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer.

Peckhaus, V. (1997). The way of logic into mathematics. *Theoria-Revista De Teoria Historia Y Fundamentos De La Ciencia*, 12(1):39–67.

Pelletier, F. J. and Hazen, A. (2012). A History of Natural Deduction. In Gabbay, D. M. and Woods, J., editors, *Handbook of the History of Logic*, pages 299–356. North-Holland, Amsterdam.

Pierce, B. C. (2002). *Types and Programming Languages.* MIT Press, Cambridge, MA.

Pitts, A. M. (2019). *Lecture Notes on Type Systems.* University of Cambridge, Computer Laboratory, Cambridge, UK.

Plato, J. (2001). Natural deduction with general elimination rules. *Archive for Mathematical Logic*, 40(7):541–567.

Restall, G. (2004). Laws of Non-Contradiction, Laws of the Excluded Middle, and Logics. In Priest, G., Beall, J. C., and Armour-Garb, B., editors, *The Law of Non-Contradiction*, pages 73–85. Oxford University Press, Oxford.

Reynolds, J. C. (1974). Towards a theory of type structure. *Lecture Notes in Computer Science*, pages 408–425.

Sato, M. (1997). Classical Brouwer-Heyting-Kolmogorov Interpretation. *Theoretical Aspects of Computer Software*, pages 162–181.

Schroeder-Heister, P. (2014). Generalized Elimination Inferences, Higher-Level Rules, and the Implications-as-Rules Interpretation of the Sequent Calculus. In Prawitz, D. and Gibbons, P., editors, *The Life of Proofs*, pages 1–29. Springer, Dordrecht.

Scott, D. (1980). Lambda Calculus: Some Models, Some Philosophy. In Barwise, J., Keisler, H. J., and Kunen, K., editors, *Studies in Logic and the Foundations of Mathematics*, pages 223–265. North-Holland, Amsterdam.

Seisenberger, M. (2003). *On the Constructive Content of Proofs.* PhD thesis, Ludwig Maximilian University of Munich.

Seldin, J. P. (1997). On the proof theory of Coquand's calculus of constructions. *Annals of Pure and Applied Logic.*

Sieg, W. and Byrnes, J. (2005). Normal Natural Deduction Proofs (in classical logic). In Beckert, B., editor, *Automated Reasoning with Analytic Tableaux and Related Methods*, pages 123–138. Springer, Berlin, Heidelberg.

Siles, V. (2010). *Investigation on the typing of equality in type systems.* PhD thesis, Unknown School.

Stefanova, M. and Geuvers, H. (1995). A simple model construction for the calculus of constructions. In Nordström, B., Palo, K., and Ranta, A., editors, *Types for Proofs and Programs*, volume 996 of *Lecture Notes in Computer Science*, pages 517–532. Springer, Berlin, Heidelberg.

Steinberger, F. (2016). Explosion and the normativity of logic. *Mind*, 125(498):385–419.

Sørensen, M. H. and Urzyczyn, P. (2006). *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics.* Elsevier.

Thiel, C. (1982). From Leibniz to Frege: Mathematical Logic Between 1679 and 1879. *Studies in the History of Mathematical Logic*, 10:300–325.

Troelstra, A. S. (1977a). Aspects of Constructive Mathematics. *Studies in Logic and the Foundations of Mathematics*, 90:973–1052.

Troelstra, A. S. (1977b). Axioms for Intuitionistic Mathematics Incompatible with Classical Logic. In Barwise, J. and Keisler, H. J., editors, *Studies in Logic and the Foundations of Mathematics*, pages 239–251. North-Holland, Amsterdam.

Wadler, P. (2015). Propositions as types. *Communications of the ACM*, 57(12):45–53.

Yallop, J. (2018). System fω. Technical report, University of Cambridge.

Yan, X. and Hanna, G. (2023). Using the Lean interactive theorem prover in undergraduate mathematics. *International Journal of Mathematical Education in Science and Technology*, 54(7):1933–1954.

Yaqub, A. M. (2013). *An Introduction to Logical Theory.* Broadview Press, Peterborough, Ontario.

Yiyun, L. (2024). Short and mechanized logical relation for dependent type theories. *University of Pennsylvania.*

Álvez, J. and Lucio, P. (2005). An algorithm for local variable elimination in normal logic programs. In Etalle, S., editor, *Logic Based Program Synthesis and Transformation*, pages 76–90. Springer, Berlin, Heidelberg.