Imperial College London

BEng Individual Project Report

Department of Computing

The Provability Semantics of Metaprogramming

Author:
Alyssa Renata

Supervisor:
Dr. Nicolas Wu

Second Marker:
Prof. Alastair Donaldson

June 27, 2022

Abstract

Theorem provers such as Lean and Coq use metaprogramming to automate proof-writing, making the process less tedious. These theorem provers are all variations of Martin-Löf's intuitionistic type theory (**MLTT**), which acts as both a logical theory and a programming language due to a correspondence between proofs and programs.

In principle, this correspondence allows metaprograms to be expressed inside the theory itself. However, incorporating the capacity to write arbitrary metaprograms compromises the theorem prover's logical consistency. It is much easier to write practical metaprograms in a metalanguage, distinct from the type theory. However, this prevents the theorem prover from reasoning about metaprograms.

However, this does not mean metaprogramming cannot be performed in the theory at all. In fact, Gödel's proof of the incompleteness theorems demonstrates that any sufficiently expressive theory is capable of what appears to be a limited form of metaprogramming. In this project, we identify the metaprogramming primitives that can be safely added to **MLTT**, justifying them via an interpretation based on work surrounding the incompleteness theorems, collectively known as *provability*.

A key mechanism in metaprogramming is the ability to evaluate code, allowing the output of the metaprograms to actually be used. Unfortunately, evaluation is inconsistent with the provability interpretation, forcing us to heavily restrict the primitives. We were able to incorporate only the ability to express and reason about metaprograms into **MLTT**, but not to use them. Moreover, this reasoning ability is stilted and impractical due to limitations imposed in order to remain sound with respect to the provability interpretation. Ultimately, provability and metaprogramming are incompatible, despite their similarities.

Acknowledgements

This project would not have been possible without the help of some people. I would like to take this opportunity to thank them.

First and foremost, I would like to thank my parents for having financially and emotionally supported me throughout my degree and in particular while working on this project. You were always eager to listen to my myriad ups & downs in life, and I will be forever grateful for that.

I am also indebted to my supervisor Nick. From the start, you have always encouraged me to be confident in pursuing my own interests in this project. At the same time, you ensured that I was on track to completing the project instead of going on a wild goose chase.

Alessandra, thank you for being my personal tutor and for introducing me to logic in my first year, which has become my primary interest since, culminating in this project.

Contents

1 ⊢ Introduction

Who watches the Watchmen?
– from *Watchmen* by Alan Moore & Dave Gibbons

Interactive theorem provers such as Lean [1], Agda [2] and Coq [3] are computer systems that aid in the development of mathematical proofs by checking their correctness. In order for the proofs to be amenable for mechanical checking, they must be written in the formal language of a logical theory. All three aforementioned theorem provers are based on variants of intuitionistic type theory, first described and formulated by Martin-Löf [4]. Dependent type theory harnesses a well-known correspondence [5] between proofs and functional programs, such that the type of the program corresponds to the theorem being proven. With this computational interpretation, the line between proof and functional program becomes blurry[1], which means theorem provers can also act as functional programming languages.

The requirement to write proofs in a formal language makes proof-writing a pedantic and tedious process, and often makes the proofs themselves less readable. When working informally, mathematicians often assume that certain sections of their proof such as routine algebra and calculations may be 'left to the reader', which also serves to improve clarity as more emphasis is placed on the novel sections of the proof. This is no longer possible if the theorem proving system requires every part of the proof to be explicitly written out. Because of this, much work has gone into automating these tedious parts of the proof-writing process by developing metaprograms known as *tactics* [6] that automatically construct proofs, possibly searching for previously established lemmas and definitions. In general, metaprograms manipulate proofs at the syntactic level, differing from regular programs which manipulate proofs by their value.

For theorem provers, metaprograms are usually written in a *metatheory*, distinct from the *object theory* in which the proofs are written. In programming language research, this is described as *heterogenous metaprogramming* [7]. This distinction between metatheory and object theory is made because for theorem provers, the object theory has to be logically consistent in order for the proofs to be reliable. However, this restricts the computational capabilities of the object theory, in particular with respect to the expression of metaprograms. If we are only concerned with writing metaprograms, then this responsibility is better handled by the metatheory which does not have to be logically consistent.

However, if we also intend to prove properties about metaprograms, then the metatheory must also be logically consistent and imbued with theorem proving capabilities. Following the heterogenous approach, this suggests we need a metametatheory, for which we need a metametametatheory, and so on ad infinitum. Therefore, it seems that the only feasible way forward is to reject heterogeneity and embrace homogeneity: write metaprograms in the object theory, allowing their properties to be proven also from within the object language. For intuitionistic type theory, this is a sensible proposition as it is both a logical theory and a programming language.

Metaprogramming in type theory is an ongoing research area which in the author's opinion, is still in its infancy. In particular, much of the research has focused on establishing computationally sound principles of metaprogramming, without much regard for the consistency of the type theory. In this project, we move in the other direction to examine how we can integrate metaprogramming into intuitionistic type theory in a logically sound way.

The key observation, as made by Gödel in his proofs of the much celebrated incompleteness theorems, is that any sufficiently expressive logical theory is already capable of some metatheoretic reasoning about itself. This line of work surrounding the incompleteness theorems is called *provability*, and is traditionally kept separate from metaprogramming. However, in adapting Gödel's work to intuitionistic type theory, we will see that provability amounts to a form of homogenous metaprogramming, albeit an impractical one. The main aim of this project is therefore to incorporate metaprogramming primitives into intuitionistic type theory, providing a more practical interface to access the provability mechanisms. Viewed from a different direction, we can also see this as using provability to justify the addition of metaprogramming

[1] We will use one term or the other depending on context, but it is ideal to always have both in mind.

primitives to intuitionistic type theory.

1.1 Outline of the Report

Chapter 2 briefly reviews classical first-order logic, providing sufficient background for us to explore Gödel's incompleteness theorems. A key notion introduced in this chapter is the *code* of a formal statement/proof, which is an object encoding the statement/proof's syntax.

In Chapter 3, we begin to make our way towards intuitionistic type theory, detailing the correspondence between proofs and functional programs. Chapter 4 takes these ideas further by introducing Martin-Löf's intuitionistic type theory, followed by an adaptation of the incompleteness theorems to intuitionistic type theory. In doing so, it becomes clear that Gödel's proofs of incompleteness is an exercise in metaprogramming.

While the previous chapters have focused entirely on the incompleteness theorems, Chapter 5 introduces the notion of modal logics. We re-examine the Gödelian notion of metaprogramming under this framework, but also use it to survey the computational notion of metaprogramming found in the literature. With the aid of the unifying framework, we discover an incompatibility between the Gödelian and computational approach.

Our work culminates in Chapter 6, which details an extension of Martin-Löf's type theory with some metaprogramming primitives. We justify this new theory by establishing that these primitives can be translated to the Gödelian mechanisms that already exist in Martin-Löf's type theory.

Finally, Chapter 7 evaluates the work in this report by examining the sort of metaprograms that our new theory can express. Chapter 8 summarises the report and provides a survey of related work, as well as some ideas on how to extend the work in this report.

1.2 Contributions

In this report, we make the following contributions:

1. A detailed exploration of the incompleteness theorems in Martin-Löf's type theory (Section 4.5). To the author's knowledge, no such detailed exploration exists yet in the literature.
2. Summarised the developments in Fitch-style modal lambda calculus and natural deduction (Subsection 5.3.3), providing a metaprogramming intuition to these systems.
3. Compared and contrasted provability against the metaprogramming under the framework of modal logic, discovering that the modal logics representing the two concepts are incompatible (Section 5.4).
4. Established a sound interpretation from Fitch-style modal lambda calculus for the **K** modality into the simply-typed level-annotated lambda calculus for metaprogramming (Section 6.1).
5. Extended this level-annotated lambda calculus to Martin-Löf's type theory, equipping it with a modal type and the metaprogramming primitives of quotes and splices. We Sketch a provability semantics for the theory, justifying informally the logical viability of the theory (Section 6.2).
6. Based on the provability semantics, we identified sound computation rules that the theory should have. Based on our computation rules, we proved the type soundness (progress & preservation) of level-annotated Martin-Löf's type theory (Section 6.3).

2 ⊢ Gödel's Incompleteness Theorems

> Deep in the human unconscious is a pervasive need for a logical universe that makes sense, But the real universe is always one step beyond logic.
>
> –from *The Sayings of Muad'Dib* by the Princess Irulan

A minimum requirement in mathematical reasoning is consistency: one should not be able to prove both a statement and the statement's negation. In the 1920s, Hilbert put forth a programme [8] towards securing the consistency of many new principles of mathematics that dealt with infinite structures, which had been criticized by some mathematicians for its unjustified treatment of infinite structures as objects in and of themselves rather than as limits of ongoing processes.

In response to these criticisms, Hilbert proposed formalisations of these principles of infinity. With a formalisation, these principles become purely manipulations of finite structures - statements and proofs in some formal language. The consistency of these formalisations are then to be proven using only principles of arithmetic: statements and derivations of these formal theories are to be encoded as numbers, so that they may be reasoned about via arithmetical principles. Thus, we can make arithmetic statements about the formal statements in these formalisations, i.e. meta-statements. Arithmetic is given such an elevated status because it deals with natural numbers - the quintessential finite structure. In a very loose sense, Hilbert had proposed a system of heterogenous metaprogramming, where arithmetic served as metalanguage while the formalisations served as object languages.

However, Hilbert's programme was later refuted by Gödel [9], who applied ideas from the programme to a formalisation of arithmetic itself - i.e. homogenous metaprogramming. However, instead of using arithmetic to prove the consistency of arithmetic, he demonstrated that arithmetic cannot even establish its own consistency, let alone the consistency of stronger forms of reasoning which must subsume arithmetic. This is Gödel's second incompleteness theorem.

In order to motivate our exploration of homogenous metaprogramming, we first examine the incompleteness theorems in their original setting.

2.1 First-order Logic

The formal theory of arithmetic that we are interested in is Peano's theory of arithmetic, or **PA** for short. **PA** is a theory defined under the framework of first-order logic (**FOL**), which defines a formal language for the expression of terms and formulas. Terms are meant to denote objects, while formulas denote statements about these objects. In the case of **PA**, the terms are intended to denote natural numbers.

2.1.1 The Syntax of First-order Logic

Terms & formulas are strings of symbols built up from a selection of logical and non-logical symbols. While the logical symbols stay fixed, the non-logical symbols are allowed to vary, allowing the formation of different theories. For arithmetic, the non-logical symbols are $+, \times, s, z$, indicating addition, multiplication, the successor function, and zero respectively.

Definition 2.1 (The Symbols Of FOL)
The logical symbols consist of

1. Parenthesis (and)

2. Variables $x_1, x_2, x_3, \ldots$
3. The equality relation $=$
4. The propositional connectives $\top, \bot, \neg, \wedge, \vee, \rightarrow$
5. Quantifiers $\forall$ and $\exists$

while there are two kinds of non-logical symbols:

1. Function symbols $f_1, f_2, f_3, \ldots$ each associated with an arity $\text{ar}(f_i)$
2. Relation symbols $R_1, R_2, R_3, \ldots$ also associated with arities $\text{ar}(R_i)$

We say a/n to mean that the function/relation symbol a has arity n.

The non-logical symbols of **PA** consist of only function symbols $+/2, \times/2, s/1, z/0$, which means that the only relation we have is equality. Notice that the constant z is expressed as a function with zero arity.

Not all strings composed of these symbols may be considered a term or formula. We only want terms & formulas that make sense relative to the intended reading of the symbols.

Definition 2.2 (FOL Terms & Formulas)
The strings that we may consider terms/formulas are defined inductively by a grammar. Below are the grammars defining terms and formulas respectively, with their intended reading to the right of each rule.

$\mathcal{T}_i$	::=	x	*Variable*
	|	$f_i(\mathcal{T}_1, \ldots, \mathcal{T}_{\text{ar}(f_i)})$	*Function*
$\mathcal{A}, \mathcal{B}$	::=	$R_i(\mathcal{T}_1, \ldots, \mathcal{T}_{\text{ar}(R_i)})$	*R_i holds of $\mathcal{T}_1, \ldots \mathcal{T}_{\text{ar}(R_i)}$*
	|	$\mathcal{T}_1 = \mathcal{T}_2$	*$\mathcal{T}_1$ equals $\mathcal{T}_2$*
	|	$\top$	*True*
	|	$\bot$	*False*
	|	$(\neg\mathcal{A})$	*not $\mathcal{A}$*
	|	$(\mathcal{A} \wedge \mathcal{B})$	*$\mathcal{A}$ and $\mathcal{B}$*
	|	$(\mathcal{A} \vee \mathcal{B})$	*$\mathcal{A}$ or $\mathcal{B}$*
	|	$(\mathcal{A} \rightarrow \mathcal{B})$	*$\mathcal{A}$ implies $\mathcal{B}$*
	|	$(\forall x.\, \mathcal{A})$	*For all x, $\mathcal{A}$ holds*
	|	$(\exists x.\, \mathcal{A})$	*There exists some x for which $\mathcal{A}$ holds*

The formula P is *atomic* iff $P \equiv R(t_1, \ldots, t_k)$, $P \equiv t_1 = t_2$ or $P \equiv \bot$.

For a term of the form $\forall x.\, \mathcal{A}$ or $\exists x.\, \mathcal{A}$, we call $\forall$ or $\exists$ its *binder*, and x its *binding variable*.

We will adopt a right associative convention to omitting parenthesis, with a precedence ordering of $\forall, \exists, \neg, \wedge, \vee, \rightarrow$ from highest to lowest. For example, $\forall x.\, A \wedge B \rightarrow C$ is to be read as $(((\forall x.\, A) \wedge B) \rightarrow C)$, and a repeated binary connective such as $A \rightarrow B \rightarrow C$ is to be read as $((A \rightarrow B) \rightarrow C)$.

The variables are intended for use together with the quantifiers, allowing the formula being quantified to depend on the variable. For example, the formula $\forall x. R(x)$ reads "for all x, $R(x)$ holds". However, variables can appear in a formula without being quantified at all. To deal with this, we introduce the notion of free & bound variables.

Definition 2.3 (Free & Bound Variables)
1. An occurence of a variable in a formula is *free* iff it does not occur inside a quantified subformula

which binds that variable. Otherwise, it is *bound*. Note that a variable may both occur free and bound at different places within a formula.

2. We write $P(x)$ to mean that P is a formula with x possibly occuring free.

3. P is a *sentence* iff no variables occur free in P.

When variables occur free in a formula or term, we may be inclined to replace the free occurences of the variables with some other terms - in some sense this denotes applying a general statement to some particular objects, as denoted by the substituted terms. This operation is called *substitution*.

Definition 2.4 (Simultaneous Substitution)
We write $P[t_1/x_1, \ldots, t_n/x_n]$ to represent the result of substituting each term t_i for free occurences of x_i in the formula P. Similarly, we may write $t[t_1/x_1, \ldots t_n/x_n]$ for substitution in the term t. Substitution is defined inductively on the structure of terms/formulas.

$$
\begin{aligned}
x[t_1/x_1, \ldots, t_n/x_n] &\implies \text{if } x = x_i \text{ then } t_i \text{ else } x \\
f(\mathcal{T}_1, \ldots, \mathcal{T}_{\mathrm{ar}(f)})[t_1/x_1, \ldots, t_n/x_n] &\implies f(\mathcal{T}_1[t_1/x_1, \ldots, t_n/x_n], \ldots, \mathcal{T}_{\mathrm{ar}(f)}[t_1/x_1, \ldots, t_n/x_n]) \\
R(\mathcal{T}_1, \ldots, \mathcal{T}_{\mathrm{ar}(R)})[t_1/x_1, \ldots, t_n/x_n] &\implies R(\mathcal{T}_1[t_1/x_1, \ldots, t_n/x_n], \ldots, \mathcal{T}_{\mathrm{ar}(R)}[t_1/x_1, \ldots, t_n/x_n]) \\
(\mathcal{T}_1 = \mathcal{T}_2)[t_1/x_1, \ldots, t_n/x_n] &\implies \mathcal{T}_1[t_1/x_1, \ldots, t_n/x_n] = \mathcal{T}_2[t_1/x_1, \ldots, t_n/x_n] \\
\top[t_1/x_1, \ldots, t_n/x_n] &\implies \top \\
\bot\,[t_1/x_1, \ldots, t_n/x_n] &\implies \bot \\
(\neg\mathcal{A})[t_1/x_1, \ldots, t_n/x_n] &\implies \neg\mathcal{A}[t_1/x_1, \ldots, t_n/x_n] \\
(\mathcal{A} \wedge \mathcal{B})[t_1/x_1, \ldots, t_n/x_n] &\implies \mathcal{A}[t_1/x_1, \ldots, t_n/x_n] \wedge \mathcal{B}[t_1/x_1, \ldots, t_n/x_n] \\
(\mathcal{A} \vee \mathcal{B})[t_1/x_1, \ldots, t_n/x_n] &\implies \mathcal{A}[t_1/x_1, \ldots, t_n/x_n] \vee \mathcal{B}[t_1/x_1, \ldots, t_n/x_n] \\
(\mathcal{A} \to \mathcal{B})[t_1/x_1, \ldots, t_n/x_n] &\implies \mathcal{A}[t_1/x_1, \ldots, t_n/x_n] \to \mathcal{B}[t_1/x_1, \ldots, t_n/x_n] \\
(\forall x.\, \mathcal{A})[t_1/x_1, \ldots, t_n/x_n] &\implies \text{if } x = x_i \\
&\quad \text{then } \forall x.\, \mathcal{A}[t_1/x_1, \ldots, t_{i-1}/x_{i-1}, t_{i+1}/x_{i+1}, \ldots, t_n/x_n] \\
&\quad \text{else } \forall x.\, \mathcal{A}[t_1/x_1, \ldots, t_n/x_n] \\
(\exists x.\, \mathcal{A})[t_1/x_1, \ldots, t_n/x_n] &\implies \text{if } x = x_i \\
&\quad \text{then } \exists x.\, \mathcal{A}[t_1/x_1, \ldots, t_{i-1}/x_{i-1}, t_{i+1}/x_{i+1}, \ldots, t_n/x_n] \\
&\quad \text{else } \exists x.\, \mathcal{A}[t_1/x_1, \ldots, t_n/x_n]
\end{aligned}
$$

There is some subtlety to substitution. In general, a simultaneous substitution $P[t_1/x_1, \ldots, t_n/x_n]$ will yield different results than the iterated single substitution $P[t_1/x_1] \ldots [t_n/x_n]$. It is also possible that a term containing x is substituted into a location where x is bound by a quantifier. This situation is called *variable capture*, and is undesirable because now the substituted term denotes a range of objects. To deal with this, we adopt *Barendregt's convention* [10] of assuming that free and bound variables are always different and the bound variable is re-labelled whenever a variable capture is about to happen due to a substitution.

2.1.2 Natural Deduction

Traditionally, the study of logic is concerned with modelling how we deduce the truth of certain statements from others. In first-order logic, this is represented formally as a system of formal proofs taking on the shape of formula trees, originally introduced by Gentzen [11, 12]. We call such formal proofs *derivations*, and reserve the word *proof* for talking about informal proofs. The branches of a derivation are formed by applying an inference rule, which accept as premise a fixed number of formulas satisfying a certain syntactic pattern and produces a new formula as conclusion. The inference rules are modelled after natural patterns of reasoning that a mathematician might employ while writing informal proofs. For example, with the premise $A \wedge B$, we have two inference rules that conclude A and B respectively, since $A \wedge B$ denotes "A and B". Because of this, the system is called *natural deduction*.

Definition 2.5 (Natural Deduction For First-Order Logic (N-FOL))
The inference rules are presented with premises above the line, and the conclusion below. Because we want to perform hypothetical reasoning by assuming certain formulas to be true, we carry around a list of such assumptions. Some rules also manipulate this list. In the following rules, Γ and Δ are placeholders for assumption lists, A and B for arbitrary formulas, and s and t for terms.

$$\frac{}{\Gamma, A, \Delta \vdash A}\ (\text{Ass}) \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ (\to I) \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ (\to E)$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ (\wedge I) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ (\wedge E_1) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ (\wedge E_2)$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ (\vee I_1) \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ (\vee I_2) \qquad \frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}\ (\vee E)$$

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ (\neg I) \qquad \frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash \bot}\ (\neg E) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ (\bot\ E) \qquad \frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\ (PC)$$

$$\frac{}{\Gamma \vdash \top}\ (\top I) \qquad \frac{\Gamma \vdash P[y/x]}{\Gamma \vdash \forall x.\, P(x)}\ (\forall I) \qquad \frac{\Gamma \vdash \forall x.\, P(x)}{\Gamma \vdash P[t/x]}\ (\forall E) \qquad \frac{\Gamma \vdash P[t/x]}{\Gamma \vdash \exists x.\, P(x)}\ (\exists I)$$

$$\frac{\Gamma \vdash \exists x.\, P(x) \qquad \Gamma, P[y/x] \vdash B}{\Gamma \vdash B}\ (\exists E) \qquad \frac{}{\Gamma \vdash t = t}\ (\text{refl}) \qquad \frac{\Gamma \vdash s = t \qquad \Gamma \vdash P[s/x]}{\Gamma \vdash P[t/x]}\ (\text{subst})$$

In the last 6 rules, s, t must denote terms whose variables do not occur bound in P to avoid variable capture when substituting, and the variable y must not occur free in Γ or B.

1. In general, we want to determine when a formula follows from some collection of assumptions, which may be infinite. However, all derivations only use finitely many assumptions anyway, since derivations are finite structures. Therefore, we say that A is *derivable* from the assumptions Ω iff there is a derivation ending in $\Gamma \vdash A$ where Γ contains only formulas from Ω. To be concise, we overload the notation and simply write $\Omega \vdash A$ when this is the case.
2. Ω is *consistent* iff $\Omega \nvdash \bot$. In other words, there is no sentence A such that both $\Omega \vdash A$ and $\Omega \vdash \neg A$ hold.

2.1.3 Propositional Logic

When we are purely concerned with the logical composition of statements, and do not care to reason about objects, then a simpler system will do where rather than having function and relation symbols as the smallest unit of syntax, we simply have atomic formulas. This formal system is called *propositional logic* (**PL**). While propositional logic has no immediate bearing on our discussion of the incompleteness theorems, it is instrumental for the rest of this report.

Definition 2.6 (PL Syntax)
The logical symbols of **PL** are the same as for **FOL** minus quantifiers and equality:

1. Connectives $\bot, \neg, \wedge, \vee, \to$
2. Parentheses (and)

The non-logical symbols consist of propositional atoms $p_1, p_2, p_3, \ldots$.

The formulas of **PL** are generated by the following grammar:

$$
\begin{array}{lcll}
\mathcal{A}, \mathcal{B} & ::= & p_i & \textit{Atom} \\
& | & \bot & \textit{False} \\
& | & \neg\mathcal{A} & \textit{not } \mathcal{A} \\
& | & (\mathcal{A} \wedge \mathcal{B}) & \mathcal{A} \textit{ and } \mathcal{B} \\
& | & (\mathcal{A} \vee \mathcal{B}) & \mathcal{A} \textit{ or } \mathcal{B} \\
& | & (\mathcal{A} \to \mathcal{B}) & \mathcal{A} \textit{ implies } \mathcal{B}
\end{array}
$$

The natural deduction system for **PL** is the same as for **FOL**, removing the rules for the connectives that are no longer available.

Definition 2.7 (Natural Deduction For Propositional Logic (N-PL))

$$\frac{}{\Gamma \vdash \top}\ (\top I) \qquad \frac{}{\Gamma, A, \Delta \vdash A}\ (\text{Ass}) \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ (\to I) \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ (\to E)$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ (\wedge I) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ (\wedge E_1) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ (\wedge E_2)$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ (\vee I_1) \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ (\vee I_2) \qquad \frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}\ (\vee E)$$

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ (\neg I) \qquad \frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash \bot}\ (\neg E) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ (\bot\ E) \qquad \frac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\ (PC)$$

2.2 Metaprogramming in the Theory of Arithmetic PA

2.2.1 The Axioms of PA

While we have described the framework of first-order logic, we have yet to describe what constitutes a theory, and in particular **PA**. **PA** is given in terms of *axioms*, a minimal collection of sentences that characterize the objects in the theory.

Definition 2.8 (Axioms of PA [13])
In the following sentences, we will use infix notation for the function symbols $+/2$ and $\times/2$, in line with informal use and to avoid a proliferation of parentheses. For the same reason, we use z and t' to refer to $z()$ and $s(t)$ respectively. Finally, we will write $t_1 \neq t_2$ as an abbreviation for $\neg t_1 = t_2$, and $A \leftrightarrow B$ for $(A \to B) \wedge (B \to A)$. The axioms of **PA** consist of the following 8 sentences

1. $\forall x.\, \forall y.\, (x' = y' \to x = y)$
2. $\forall x.\, x' \neq \mathsf{z}$
3. $\forall x.\, (x = \mathsf{z} \vee \exists y.\, x = y')$
4. $\forall x.\, x + \mathsf{z} = x$
5. $\forall x.\, \forall y.\, x + y' = (x + y)'$
6. $\forall x.\, x \times \mathsf{z} = \mathsf{z}$
7. $\forall x.\, \forall y.\, x \times y' = (x \times y) + x$
8. $\forall x.\, \forall y.\, x < y \leftrightarrow \exists k.\, k' + x = y$

along with a sentence of the form

$$\forall y_1. \ldots \forall y_n. ((A[\mathsf{z}/x] \land \forall x. (A(x) \to A[x'/x])) \to \forall x. A(x))$$

for each formula $A(x)$ with variables $x, y_1 \ldots y_n$ possibly occuring free. Notice that the last sentence is the induction principle for natural numbers.

Now, denoting the above collection of axioms as $\mathbf{PA}_0$, the theory **PA** is obtained as the collection of all sentences that can be proven from $\mathbf{PA}_0$. In other words, it is the closure of the axioms under $\vdash$.

$$\mathbf{PA} = \{A \mid \mathbf{PA}_0 \vdash A\}$$

This suggests a general definition of theory in first-order logic.

Definition 2.9 (First-order Theory)
A set of **FOL** sentences Ω is a *theory* iff it contains every sentence it proves, i.e. $\Omega = \{A \mid \Omega \vdash A\}$.

2.2.2 Encoding Formulas as Numbers

Following Hilbert's programme, Gödel encoded the formulas and derivations of **PA** as numbers - the objects of **PA**. We refer to the result of encoding a formula/derivation as its *Gödel code* or just *code*.

First, recall that other than the variables, we only have finitely many symbols in the language of first-order logic and **PA**. Therefore, we can simply assign the first few natural numbers as an encoding for these symbols, and use the remaining natural numbers to encode the variables. For example,

$\forall$	$\exists$	$\land$	$\lor$	$\neg$	$\to$	$\bot$	$=$	$+$	$\times$	z	s	x_1	$\ldots$	x_i	$\ldots$
0	1	2	3	4	5	6	7	8	9	10	11	12	$\ldots$	$11+i$	$\ldots$

The exact assignment is not important, however it is important that the encoding assignment be injective: two distinct objects must have distinct codes. An injective encoding ensures no information is lost, so that given the code of some object, we can discern the unique object having that code.

Now, terms & formulas are strings composed of such symbols, so we need a way of turning strings of symbols into a number. We can already transform each symbol into its code, so this becomes the problem of compacting a strings of numbers into a single number. One way of doing this is to take the power of some prime factor to each number in the string - the product of these factor powers is the code of the string.

Definition 2.10 (Encoding Strings [14])
Let $\langle\!\langle a_1, \ldots, a_n \rangle\!\rangle$ denote the encoding of the string $a_1 \ldots a_n$

$$\langle\!\langle a_1, \ldots, a_n \rangle\!\rangle \triangleq p_1^{a_1+1} \cdot \ldots \cdot p_n^{a_n+1}$$

where p_i is the i^{th} prime number in ascending order. We add 1 to each a_i so that we can distinguish sequences containing zero, for example between $\langle\!\langle 2, 5, 6 \rangle\!\rangle$ and $\langle\!\langle 2, 5, 0, 6, 0, 0 \rangle\!\rangle$.

By the prime factorisation theorem, any two distinct sequences are encoded differently since they are encoded as two different factorisations. Let us denote the encoding of a term t and formula A by this method as ${}^{\#}t^{\#}$ and ${}^{\#}A^{\#}$.

For the natural deduction derivation trees, an encoding has to follow the tree structure of the derivation.

Definition 2.11 (Encoding Trees [14])
A tree can be encoded as

$$\langle\!\langle k, \delta_1, \ldots, \delta_k, l \rangle\!\rangle$$

where k is the number of subtrees of this tree, $\delta_1 \ldots \delta_k$ are the codes of the subtrees, and l is a label at the root of the tree.

In the context of derivations, the label will be $\langle\!\langle {}^{\#}\Gamma \vdash A^{\#}, m \rangle\!\rangle$, consisting of the code of the conclusion $\Gamma \vdash A$ along with the code m of the inference rule being applied. If $\Gamma = B_1 \ldots B_n$, then one option for the code ${}^{\#}\Gamma \vdash A^{\#}$ is $\langle\!\langle {}^{\#}A^{\#}, n, {}^{\#}B_1{}^{\#}, \ldots, {}^{\#}B_n{}^{\#} \rangle\!\rangle$. As for the inference rule's code, since there are only finitely many inference rules we can simply assign a number to each rule, as we did with the symbols.

Note that well-formed formulas also have a tree structure, so we could have defined an encoding in this way for well-formed formulas as well. Such an encoding will only be well-defined for well-formed formulas, but this is not a real problem as we are only really concerned with well-formed formulas.

Finally, in **PA** any natural number n has an obvious canonical representation

$$\overline{n} \equiv \underbrace{s(s(\ldots s(z)\ldots))}_{n \text{ times}}$$

so we may represent a formula A in the language of **PA** as $\overline{{}^{\#}A^{\#}}$, which we shall abbreviate as $\ulcorner A \urcorner$.

2.2.3 Representing Functions & Relations in PA

The next phase in Gödel's incompleteness proof is to represent functions and relations on formulas and derivations inside **PA**. Of particular interest is the relation $\mathrm{prf}_{\mathbf{PA}}(\delta, A)$ which holds iff δ is a derivation of formula A with **PA** as the set of assumptions, i.e. $\mathbf{PA} \vdash A$. This will allow **PA** to reason about the derivability of its own formulas. Of course, **PA** can really only represent numbers, and functions/relations on numbers. Therefore are two steps to this phase:

1. Define what it means for **PA** to represent functions/relations on numbers.
2. Translate functions and relations on formulas/derivations as functions and relations on the encoding of the functions and relations.

For the first step, **PA** does not provide the means to define new functions or relations, so we will have to represent them by using formulas.

Definition 2.12 (Representable Functions & Relations [14])
1. A function $f(x_1, \ldots, x_k)$ is said to be *represented* by the **PA**-formula $\overline{f}(x_1, \ldots, x_k, y)$ iff for every natural number $n_1, \ldots, n_k, m$

$$f(n_1, \ldots n_k, m) \text{ implies } \mathbf{PA} \vdash \forall y.\ (\overline{f}(\overline{n_1}, \ldots, \overline{n_k}, y) \leftrightarrow y = \overline{m})$$

2. A relation $R(x_1, \ldots, x_k)$ is *represented* by $\overline{R}(x_1, \ldots, x_k)$ iff for every $n_1, \ldots, n_k$

$$\begin{gathered} \text{if } R(n_1, \ldots, n_k) \text{ holds then } \mathbf{PA} \vdash \overline{R}(\overline{n_1}, \ldots, \overline{n_k}) \\ \text{if } R(n_1, \ldots, n_k) \text{ does not hold then } \mathbf{PA} \vdash \neg\overline{R}(\overline{n_1}, \ldots, \overline{n_k}) \end{gathered}$$

As it turns out however, not all functions/relations are representable in **PA**; only the computable ones are [14]. Because we are dealing with functions on natural numbers, the most fitting model of computation to adopt is that of the recursive functions.

Definition 2.13 (Recursive Functions & Relations [14])

1. First, the *partial recursive functions* are defined inductively, noting that whenever we define a partial function f in terms of other partial functions $g_1, \ldots, g_k$, f is only well-defined on some input value when $g_1, \ldots, g_k$ are themselves defined on their given input values. Otherwise, f is undefined.

   (a) The function $\text{zero}(x) \triangleq 0$ is partial recursive.

   (b) The function $\text{succ}(x) \triangleq x + 1$ is partial recursive.

   (c) For each natural number n and $1 \leq i \leq n$, the function $\text{pr}^n_i(x_1, \ldots x_n) \triangleq x_i$ is partial recursive.

   (d) If $f(x_1, \ldots, x_k)$ and $g_1(x_1, \ldots, x_n) \ldots g_k(x_1, \ldots, x_n)$ are partial recursive, then so is their composition $\text{comp}[f, g_1, \ldots, g_k](x_1, \ldots, x_n) \triangleq f(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n))$.

   (e) If $f(x_1, \ldots x_n)$ and $g(x_1, \ldots, x_{n+2})$ are partial recursive, then the recursively defined partial function

   $$\begin{aligned} \text{rec}[f, g](x_0, \ldots, x_n, 0) &\triangleq f(x_0, \ldots, x_n) \\ \text{rec}[f, g](x_0, \ldots, x_n, y+1) &\triangleq g(x_0, \ldots, x_n, y, \text{rec}[f, g](x_0, \ldots, x_n, y)) \end{aligned}$$

   is also partial recursive.

   (f) If $f(x_0, \ldots x_n, y)$ is partial recursive, then so is the unbounded search function

   $$\mu[f](x_0, \ldots, x_n) = y \iff \text{for all } i < y,\ f(x_0, \ldots, x_n, i) > 0 \text{ and } f(x_0, \ldots, x_n, y) = 0$$

2. The primitive recursive functions are those constructed using only 1.a) - 1.e) above. Because only 1.f) introduces partiality, the primitive recursive functions are total.

3. The total recursive functions, or *recursive functions* for short, are the partial recursive functions that are defined for all inputs.

4. A relation $R(x_0, \ldots, x_n)$ is *recursively decidable* iff its characteristic function

   $$\chi_R(x_0, \ldots, x_n) = \begin{cases} 1 & \text{if } R(x_0, \ldots, x_n) \text{ holds} \\ 0 & \text{otherwise} \end{cases}$$

   is a recursive function.

This means that any function we want to represent in **PA** has to first be shown to be recursive. However, the constructions are long and tedious, so we refer to chapter 3 of [14] for the particulars of the construction. In particular, we borrow the result that the following functions are primitive recursive and therefore recursive, since they are directly used in the incompleteness proof.

Lemma 2.14

The following function and relation are primitive recursive/recursively decidable, and therefore representable [14].

- The relation $\text{prf}_{\mathbf{PA}}(d, a)$, which holds iff d codes a derivation, with the axioms of **PA** as assumptions, of the formula coded by a. Intuitively, this is computable because one simply has to check each application of an inference rule follows the proper shape. For (Ass), one has to check whether the conclusion is an axiom of **PA**, but this is simply checking whether it is one

of the first 8 axioms or whether it fits the shape of the induction principle.

- The function $\text{diag}(a)$ which computes the code of $A[\ulcorner A(x) \urcorner / x]$ if a codes the formula $A(x)$ with exactly one free variable x. This is also intuitively computable as one can simply iterate over the symbols and check whether each is a free variable, and if so replace it by $\ulcorner A(x) \urcorner$.

2.3 Gödel's Incompleteness Theorems

The first incompleteness theorem states that in **PA**, there are certain sentences that cannot be derived to be true nor false. That is to say, the theory is incomplete. The second incompleteness theorem, which follows as a corollary of the first, states that the **PA** sentence $\neg \exists x.\ \overline{\text{prf}_{\mathbf{PA}}}(x, \ulcorner \bot \urcorner)$ expressing **PA**'s own consistency also cannot be derived to be true nor false. For the remainder of this section, we drop the subscript **PA** from prf, since it is obvious that we are talking about **PA**.

Definition 2.15 (Complete Theory [14])
The theory Ω is *complete* iff for every sentence A in the language of Ω, either $\Omega \vdash A$ or $\Omega \vdash \neg A$. Otherwise, it is *incomplete*.

A sentence A for which $\Omega \nvdash A$ and $\Omega \nvdash \neg A$ is said to be *independent* of Ω.

Gödel's second theorem served a massive blow to Hilbert's programme as it means arithmetic is not even able to prove its own consistency. We shall prove the first incompleteness theorem and then sketch a proof for the second theorem, as a complete proof will be quite long & tedious.

The first proof proceeds by explicitly constructing the sentence G, which has the property

$$\mathbf{PA} \vdash G \leftrightarrow \neg \text{Prov}[\ulcorner G \urcorner / x] \tag{2.1}$$

where $\text{Prov}(x)$ is abbreviation for $\exists y.\ \overline{\text{prf}}(y, x)$. More concisely, we say G is the *fixed-point* of $\text{Prov}(x)$. To obtain a neater proof, we describe the construction of fixed-points in general.

Lemma 2.16 (Fixed-point Property)
For any formula $B(x)$ with only x possibly occuring free, there is a sentence A such that

$$\mathbf{PA} \vdash A \leftrightarrow B[\ulcorner A \urcorner / x].$$

Proof. A may be defined using $\text{diag}(x)$. For details see [14] or Chapter 4, where we prove the same lemma but for **MLTT**. ⊣

A contradiction should be apparent from an informal gloss of the fixed-point property (2.1):

"G holds if and only if G is not provable."

We leverage this towards a proof by contradiction of G's independence, where assuming either $\mathbf{PA} \vdash G$ or $\mathbf{PA} \vdash \neg G$ leads to a contradiction, precisely because of the fixed-point property.

Now, the incompleteness theorem has to assume the consistency of **PA**, since otherwise an inconsistent theory can prove anything via the ($\bot$E) rule of natural deduction. Unfortunately, the assumption of consistency is not quite strong enough to prove the contradiction, so we have to assume the somewhat

awkward but stronger condition of ω-consistency[1] It is possible to obtain a variation of the theorem [15] which only assumes consistency, but this has to be down with a more awkward definition of $\text{Prov}(x)$.

Definition 2.17 (ω-consistency [14])
PA is ω-consistent iff for any formula $A(x)$ with only x possibly occuring free, if $\mathbf{PA} \vdash \neg A[\overline{m}/x]$ for all natural numbers m, then $\mathbf{PA} \nvdash \exists x.\, A(x)$.

ω-consistency easily implies consistency because if **PA** is inconsistent, then it can derive anything - in particular, $A[\overline{m}/x]$ and $\neg A[\overline{m}/x]$ for any $A(x)$ and m. The former derivations allow us to derive $\exists x.\, A(x)$, which in combination with the latter derivations means **PA** is ω-inconsistent. Armed with this knowledge, we are now ready to tackle the first incompleteness theorem.

Theorem 2.18 (First Incompleteness Theorem [14])
If **PA** is ω-consistent, then G is independent of **PA**, where G is the fixed-point of $\text{Prov}(x)$.

Proof. [14]

($\mathbf{PA} \nvdash G$) Since **PA** is ω-consistent, it is consistent. Now suppose for a contradiction that $\mathbf{PA} \vdash G$, which means there is a natural deduction proof of G coded by the number d. By the representability of prf, we then have that $\mathbf{PA} \vdash \overline{\text{prf}}(\overline{d}, \ulcorner G \urcorner)$, and so $\mathbf{PA} \vdash \text{Prov}(\ulcorner G \urcorner)$. At the same time, $\mathbf{PA} \vdash \neg\text{Prov}(\ulcorner G \urcorner)$ by applying the fixed-point property to $\mathbf{PA} \vdash G$. Together, these imply **PA** is inconsistent, contradicting the fact that **PA** is consistent.

($\mathbf{PA} \nvdash \neg G$) Suppose for a contradiction that $\mathbf{PA} \vdash \neg G$. By the fixed-point property yet again, $\mathbf{PA} \vdash \text{Prov}(\ulcorner G \urcorner)$ which is just shorthand for

$$\mathbf{PA} \vdash \exists x.\, \overline{\text{prf}}(x, \ulcorner G \urcorner). \tag{2.2}$$

Since **PA** is consistent, we must also have $\mathbf{PA} \nvdash G$. Because there is no derivation of G, no number codes a derivation of G. Hence by the representability of $\overline{\text{prf}}$, for any n, $\mathbf{PA} \vdash \neg\overline{\text{prf}}(\overline{n}, \ulcorner G \urcorner)$. Combining this with (2.2) shows that **PA** is ω-inconsistent, which is a contradiction. ⊣

The second incompleteness theorem is obtained as a corollary of the first incompleteness theorem by *formalising* the first theorem inside **PA**. In particular, we show that **PA** can derive the first half of the first incompleteness theorem

Lemma 2.19 (Formalised First Theorem [16])

$$\mathbf{PA} \vdash \text{Con}_{\mathbf{PA}} \rightarrow \neg\text{Prov}(\ulcorner G \urcorner)$$

where $\text{Con}_{\mathbf{PA}} \equiv \neg\text{Prov}(\ulcorner \bot \urcorner)$. As with prf, we will drop the subscript for now. A complete proof of this formalisation is tedious and depends on the particular coding of formulas & proofs[2]. Fortunately, Hilbert & Bernays [17] isolated some high-level conditions, later simplified by Löb [18] that $\text{Prov}(x)$ must satisfy in order to carry out a simpler proof of the formalised first theorem.

[1] The notation borrows from the study of ordinal numbers, where ω denotes the lowest upper bound of the set of all natural numbers $\{0, 1, 2, \ldots\}$.

[2] Even Gödel only gave a brief sketch in his original paper [9].

Definition 2.20 (Hilbert-Bernays-Löb Conditions [14])
For all sentences A and B,

1. if **PA** $\vdash A$ then **PA** $\vdash \mathrm{Prov}(\ulcorner A \urcorner)$.
2. **PA** $\vdash \mathrm{Prov}(\ulcorner A \to B \urcorner) \to (\mathrm{Prov}(\ulcorner A \urcorner) \to \mathrm{Prov}(\ulcorner B \urcorner))$.
3. **PA** $\vdash \mathrm{Prov}(\ulcorner A \urcorner) \to \mathrm{Prov}(\ulcorner \mathrm{Prov}(\ulcorner A \urcorner) \urcorner)$.

Assuming these conditions to hold of our provability predicate, we can now prove the formalised first theorem.

Proof. [14]

PA $\vdash G \to (\mathrm{Prov}(\ulcorner G \urcorner) \to \bot)$ (2.3)
From the fixed-point property of G and the equivalence $\neg A \leftrightarrow (A \to \bot)$
PA $\vdash \mathrm{Prov}(\ulcorner G \to (\mathrm{Prov}(\ulcorner G \urcorner) \to \bot) \urcorner)$ (2.4)
Apply Definition 2.20.1 to (2.3)
PA $\vdash \mathrm{Prov}(\ulcorner G \urcorner) \to \mathrm{Prov}(\ulcorner \mathrm{Prov}(\ulcorner G \urcorner) \urcorner) \to \mathrm{Prov}(\ulcorner \bot \urcorner)$ (2.5)
Apply Definition 2.20.2 twice to (2.4)
PA $\vdash \mathrm{Prov}(\ulcorner G \urcorner) \to \mathrm{Prov}(\ulcorner \bot \urcorner)$ (2.6)
By Definition 2.20.3, the second premise in (2.5) is redundant
PA $\vdash \mathrm{Con} \to \neg \mathrm{Prov}(\ulcorner G \urcorner)$ (2.7)
By contraposition of (2.6) and by definition of Con ⊣

Now, if **PA** is consistent, it cannot be that **PA** $\vdash$ Con, since we can apply the formalised first theorem to derive $\neg \mathrm{Prov}(\ulcorner G \urcorner)$ and this is equivalent to G by the fixed-point property. This of course contradicts the (informal) first theorem.

On the other hand, if **PA** is ω-consistent, then it also does not derive $\neg$Con. Suppose for a contradiction that **PA** $\vdash \neg$Con. Unfolding definitions, this is equivalent to

$$\textbf{PA} \vdash \exists x.\, \overline{\mathrm{prf}}(x, \ulcorner \bot \urcorner). \tag{2.8}$$

However, since **PA** is consistent as implied by its ω-consistency, it cannot derive $\bot$. This means that for all n, **PA** $\vdash \neg\overline{\mathrm{prf}}(\overline{n}, \ulcorner \bot \urcorner)$. In combination with (2.8), this means **PA** is ω-inconsistent, a contradiction. bears us the second incompleteness theorem.

Theorem 2.21 (Second Incompleteness Theorem [14])
If **PA** is ω-consistent, $\mathrm{Con}_{\mathbf{PA}}$ is independent of **PA**.

The most important part of this chapter is not the incompleteness theorems themselves, but rather the development of "metaprogramming" mechanisms in **PA**. The incompleteness theorems simply serve to delineate the boundaries of what we can do with these mechanisms. However, calling these mechanisms "metaprogramming" is not quite appropriate, because **PA** is not a programming language. In the following chapters, we adapt Gödel's ideas to Martin-Löf's type theory, which can serve as both a mathematical theory and as a programming language. In such a theory, Gödel's ideas really can be considered metaprogramming.

3 ⊢ The Curry-Howard Correspondence

> One cannot inquire into the foundations and nature of mathematics without delving into the question of the operations by which the mathematical activity of the mind is conducted.
>
> –L.E.J Brouwer

In the previous chapter, we observed how, even though we had assumed **PA** to be consistent, **PA**'s consistency remained independent of **PA**. This suggests a gap between the notion of truth and formal provability. In this chapter, we shall examine intuitionistic logic: a mathematically inclined reformulation of logic from first principles, in which logical truth instead corresponds to the notion of *constructive proof*, an idea originating from Brouwer's constructive approach to mathematics [19]. In constructive mathematics, mathematical objects exist only if they can be explicitly constructed, and this includes logical statements whose constructions are proofs. The result is a system of logic radically different from the first-order or propositional logic of the previous chapter, which we shall henceforth collectively call "classical logic". It is weaker in the sense that classical logic validates general principles which intuitionistic logic does not. However, intuitionistic logic is stronger in the sense that it demands more from the proof of a statement.

Brouwer takes "construction" to mean computation, in the sense of Turing machines [20], Church's lambda calculus [21] or the recursive functions we investigated in the previous chapter. Because of this, proofs in intuitionistic logic carry some inherent computational content. Of the three computational models described however, only the lambda calculus contains programs in direct structural correspondence with intuitionistic proofs [22]. This correspondence is called the Curry-Howard correspondence, and is exactly what allows intuitionistic logic to take the role of both logical theory and programming language.

This chapter and the next are dedicated to intuitionistic logic. In this chapter, we ease in the core ideas of the Curry-Howard correspondence by exploring the correspondence for propositional intuitionistic logic. This provides a system suitable for programming, but too simplistic for expressing mathematical theorems. In the next chapter, we rectify this by extending the system to include quantifiers, the equality relation, and common mathematical objects.

3.1 The BHK Interpretation

While Brouwer himself was not interested in a formalisation of intuitionistic logic, his contemporaries sought a formalisation that allowed a comparison with classical logic. This lead to the semi-formal Brouwer-Heyting-Kolmogorov interpretation [23, 24, 25] of the logical connectives.

Definition 3.1 (BHK Interpretation)

- $\bot$ has no proof.
- $\top$ trivially always has a proof.
- A proof of $A \wedge B$ constitutes a pair consisting of the proof of A and a proof of B.
- A proof of $A \vee B$ constitutes exactly one of either a proof of A or a proof of B, and a specification of whether it is a proof of A or B.
- A proof of $A \to B$ is a construction that gives a proof of B given a proof of A.

- A proof of $\neg B$ is a construction that gives a proof of $\bot$ given a proof of B. In other words, $\neg B$ is shorthand for $B \to \bot$.

The BHK interpretation is highly reminiscent of the introduction rules in the natural deduction system **N-PL** from the previous chapter. Note that in **N-PL**, any proposition A may also be proven by assuming the negation $\neg A$ and using it to derive a contradiction. However, under the BHK interpretation this merely refutes $\neg A$ rather than proves A. It is therefore not a valid inference of intuitionistic logic. Hence, a natural deduction system **N-IPL** for intuitionistic propositional logic is given by omitting the rule proof by contradiction rule (PC) from **N-PL**. One consequence of the removal is that we no longer have equivalence between $\neg\neg A$ and A - the latter implies the former but not vice versa.

Another consequence of the removal is that the law of excluded middle $A \vee \neg A$, which is provable for any proposition A in classical logic, is no longer generally provable for intuitionistic logic. This follows from the BHK interpretation: a proof of $A \vee \neg A$ must constitute either a proof of A or $\neg A$, and its not always possible to give a proof of one or the other. A good example we have seen is $\text{Con}_{\mathbf{PA}}$, for **PA** cannot derive $\text{Con}_{\mathbf{PA}}$ or its negation. Of course **PA** can still derive $\text{Con}_{\mathbf{PA}} \vee \neg\text{Con}_{\mathbf{PA}}$ since it is formulated in classical logic, which only serves to demonstrate the difference between intuitionistic and classical connectives.

3.2 Lambda Calculus

The notion of "construction" is not defined in the BHK interpretation. However, we will see that there is very good sense in using the lambda calculus to fill the role of "construction". The lambda calculus was originally developed by Alonzo Church in 1932 [21]. It reformulated functions as rules for obtaining output from input, instead of as graphs from input values to output. In its purest form, the only construction of the pure lambda calculus are functions, leading to a concise syntax.

Definition 3.2 (λ-term [26]) $\mathcal{M}, \mathcal{N}$

$$\begin{array}{rcll} & ::= & x & \textit{Variable} \\ & | & (\lambda x.\ \mathcal{M}) & \textit{Abstraction} \\ & | & (\mathcal{M}\ \mathcal{N}) & \textit{Application} \end{array}$$

As with the quantified formulas in **FOL**, given a term of the form $\lambda x.\ \mathcal{M}$, we call λ the binder and x the *binding variable* of the term.

We will omit any unnecessary parenthesis according to a left-associative convention for application, i.e. $M_1 M_2 M_3 \dots M_k$ is to be read as $(\dots((M_1 M_2) M_3)\dots M_k)$. Application is given a higher precedence than abstraction, such that $\lambda x.\ M\ N$ reads as $\lambda x.\ (M\ N)$ rather than $(\lambda x.\ M)\ N$.

The notion of substitution for the lambda calculus is very important because the computation of λ-terms are expressed in terms of substitutions. The definition of a free/bound occurence remains the same as in **FOL**, so substitution replaces free occurences of variables in much the same way. However, it is worth reiterating the substitution for clarity.

Definition 3.3 (Simultaneous Substitution)

$$\begin{array}{rcl} x[t_1/x_1, \dots, t_n/x_n] & \Longrightarrow & \text{if } x = x_i \text{ then } t_i \text{ else } x \\ (\mathcal{M}\ \mathcal{N})[M_1/x_1, \dots, M_n/x_n] & \Longrightarrow & \mathcal{M}[M_1/x_1, \dots, M_n/x_n]\ \mathcal{N}[M_1/x_1, \dots, M_n/x_n] \\ (\lambda x.\ \mathcal{M})[M_1/x_1, \dots, M_n/x_n] & \Longrightarrow & \text{if } x = x_i \\ & & \text{then } \lambda x.\ \mathcal{M}[M_1/x_1, \dots, M_{i-1}/x_{i-1}, M_{i+1}/x_{i+1}, \dots, M_n/x_n] \\ & & \text{else } \lambda x.\ \mathcal{M}[M_1/x_1, \dots, M_n/x_n] \end{array}$$

As with **FOL**, we assume Barendregt's convention to avoid variable capture.

Abstraction constructs a function which takes input x and returns M. When an abstraction is applied to a term, i.e. $(\lambda x.\ M)\ N$, we can evaluate the result of the function on N by substituting it for x in M. Therefore, computation takes place by repeatedly reducing applied abstractions to substitutions. This is expressed in terms of an inductively defined binary relation called *β-reduction*.

Definition 3.4 (β-reduction [26])
β-reduction takes place when an abstraction is applied to another term:

$$\frac{}{(\lambda x.\ M)\ N \rightsquigarrow_\beta M[N/x]}$$

Any term of the form $(\lambda x.\ M)\ N$ is called a *redex*.

Congruence rules allow the reduction to take place at an arbitrary subterm.

$$\frac{M \rightsquigarrow_\beta M'}{\lambda x.\ M \rightsquigarrow_\beta \lambda x.\ M'} \qquad \frac{M \rightsquigarrow_\beta M'}{M\ N \rightsquigarrow_\beta M'\ N} \qquad \frac{N \rightsquigarrow_\beta N'}{M\ N \rightsquigarrow_\beta M\ N'}$$

The relation $M \rightsquigarrow_\beta^* N$ expresses whether M can reduce to N in a finite amount of steps, and is defined as the reflexive transitive closure of $\rightsquigarrow_\beta$. The relation $M =_\beta N$ is the equivalence relation expressing whether M and N reduce to some common term. It is defined as the reflexive transitive symmetric closure of $\rightsquigarrow_\beta$.

Definition 3.5 (β-normal and Neutral Terms)
A term that contains no redex is said to be in *β-normal form*, since it can no longer be reduced. We give an explicit description of terms in this form. We define normal terms $\mathcal{U}$ coinductive with neutral terms $\mathcal{V}$, which can be thought of as an application that is stuck because it is not a redex.

$$\begin{array}{rcl} \mathcal{V} & ::= & x \\ & | & (\mathcal{V}\ \mathcal{U}) \end{array} \qquad\qquad \begin{array}{rcl} \mathcal{U} & ::= & \mathcal{V} \\ & | & (\lambda x.\ \mathcal{U}) \end{array}$$

3.3 The Correspondence For $\rightarrow$

In the BHK interpretation, a proof of $A \rightarrow B$ is essentially a function from proofs of A to proofs of B, although it is not specified what sort of function it should be. One option for this is to use λ-terms. They are good candidates because the structure of λ-terms is in direct correspondence with the structure of natural deduction derivations concerning only the $\rightarrow$ connective. We can see this by annotating the derivation's conclusion with terms, and the formulas in the context with variables.

Definition 3.6 (Simply-typed Lambda Calculus)

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ (\text{Ax}) \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\ M : A \rightarrow B}\ (\rightarrow I) \qquad \frac{\Gamma \vdash M : A \rightarrow B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\ (\rightarrow E)$$

From the computational perspective, this system of derivations is seen as an assignment of types to a subset of the λ terms, with $A \rightarrow B$ serving as the type of functions from A to B. Hence, the above system is called the simply typed lambda calculus [27].

Following the BHK interpretation, we seek to identify derivations of $A \to B$ with terms of the lambda calculus. We can see that this is indeed the case: the introduction rule ($\to$I) is associated with abstraction, the elimination rule ($\to$E) with application, and the (Ass) rule with variables. In other words, the syntax of the term determines the structure of the proof. Unfortunately, with our current definition, a λ-term does not yet carry enough information to determine a unique derivation. For example, the term $\lambda x.\, x$ can be typed in infinitely many ways by any formula of the form $A \to A$. To fix this, we annotate the binding variable of a λ abstraction by its expected type [27],

$$\lambda x : A.\, M : A \to B$$

allowing the following uniqueness properties about the derivations that correspond to a lambda term.

Theorem 3.7 (Uniqueness of Typing [22])

If $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$, then $A = B$.

Theorem 3.8 (Uniqueness of Derivation)
Any two derivation trees of $\Gamma \vdash M : A$ are equal.

When a term uniquely determines a derivation, we say that the system is *syntax directed*. We will often omit the binding variable annotations for brevity, whenever they can be easily inferred.

3.4 Extending The Correspondence to Other Connectives of IPL

While the lambda calculus fills in the role of proofs concerning $\to$, it does not have the necessary structure to represent the proofs of the other connectives. We can however extend the lambda calculus with additional programming structures that allow exactly that.

Definition 3.9 (Extended λ-term [22])

$\mathcal{M}, \mathcal{N}$	::=	$x \in \mathcal{X}$	*Variable*
	|	$\lambda x.\, \mathcal{M}$	*Abstraction*
	|	$\mathcal{M}\ \mathcal{N}$	*Application*
	|	$\langle\rangle$	*Unit*
	|	$\langle \mathcal{M}, \mathcal{N} \rangle$	*Pair*
	|	$\pi_1(\mathcal{M})$	*Left Projection*
	|	$\pi_2(\mathcal{M})$	*Right Projection*
	|	$\mathsf{in}_1(\mathcal{M})$	*Left Injection*
	|	$\mathsf{in}_2(\mathcal{M})$	*Right Injection*
	|	$\mathsf{case}(\mathcal{M}, x.\mathcal{N}_1, x.\mathcal{N}_2)$	*Case Split*
	|	$\mathsf{expl}(M)$	*Explosion*

Definition 3.10 (Extended β-reduction)
In addition to the reduction rules for functions, add the following rules:

$$\frac{}{\pi_1(\langle M, N\rangle) \rightsquigarrow_\beta M} \qquad \frac{}{\pi_2(\langle M, N\rangle) \rightsquigarrow_\beta N}$$

$$\frac{}{\mathsf{case}(\mathsf{in}_1(M), x.N_1, x.N_2) \rightsquigarrow_\beta N_1[M/x]} \qquad \frac{}{\mathsf{case}(\mathsf{in}_2(M), x.N_1, x.N_2) \rightsquigarrow_\beta N_2[M/x]}$$

as well as congruence rules:

$$\frac{M \rightsquigarrow_\beta M'}{\langle M, N\rangle \rightsquigarrow_\beta \langle M', N\rangle} \qquad \frac{M \rightsquigarrow_\beta N'}{\langle M, N\rangle \rightsquigarrow_\beta \langle M, N'\rangle} \qquad \frac{M \rightsquigarrow_\beta M'}{\pi_1(M) \rightsquigarrow_\beta \pi_1(M')} \qquad \frac{M \rightsquigarrow_\beta M'}{\pi_2(M) \rightsquigarrow_\beta \pi_2(M')}$$

$$\frac{M \rightsquigarrow_\beta M'}{\mathsf{in}_1(M) \rightsquigarrow_\beta \mathsf{in}_1(M')} \qquad \frac{M \rightsquigarrow_\beta M'}{\mathsf{in}_2(M) \rightsquigarrow_\beta \mathsf{in}_2(M')}$$

$$\frac{M \rightsquigarrow_\beta M'}{\mathsf{case}(M, x.N_1, x.N_2) \rightsquigarrow_\beta \mathsf{case}(M', x.N_1, x.N_2)}$$

$$\frac{N_1 \rightsquigarrow_\beta N_1'}{\mathsf{case}(M, x.N_1, x.N_2) \rightsquigarrow_\beta \mathsf{case}(M, x.N_1', x.N_2)} \qquad \frac{N_2 \rightsquigarrow_\beta N_2'}{\mathsf{case}(M, x.N_1, x.N_2) \rightsquigarrow_\beta \mathsf{case}(M, x.N_1, x.N_2')}$$

$\langle M, N\rangle$ is a pair consisting of two terms M and N, while $\pi_1(M)$ and $\pi_2(M)$ extracts the left and right members (respectively) of the pair M. This corresponds to conjunction. The structure that corresponds to disjunction is the disjoint union or option. $\mathsf{in}_1(M)$ constructs the left option, while $\mathsf{in}_2(M)$ constructs the right option. $\mathsf{case}(M, x.N_1, x.N_2)$ extracts the contents of a disjoint union by substituting it for x in N_1 if M is a left option, or in N_2 if M is a right option.

Notice the general pattern: we have syntax that denotes the construction of some structure, along with syntax that denotes its destruction. β-reduction is always defined by destructors "cancelling out" a constructor. The constructors correspond to the introduction rules, while the destructors correspond to elimination rules.

We now give the annotated deduction rules for the remaining connectives, noting that some additional term formers also have to be annotated to ensure syntax-directedness.

Definition 3.11 (Extended Simply-typed Lambda Calculus)

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N\rangle : A \wedge B}\ (\wedge I) \qquad \frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \pi_1(M) : A}\ (\wedge E_1) \qquad \frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \pi_2(M) : B}\ (\wedge E_2)$$

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{in}_1(M)^B : A \vee B}\ (\vee I_1) \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \mathsf{in}_2(M)^A : A \vee B}\ (\vee I_2)$$

$$\frac{\Gamma \vdash M : A \vee B \qquad \Gamma, x : A \vdash N_1 : C \qquad \Gamma, x : B \vdash N_2 : C}{\Gamma \vdash \mathsf{case}(M, x.N_1, x.N_2) : C}\ (\vee E) \qquad \frac{\Gamma \vdash M : \bot}{\Gamma \vdash \mathsf{expl}(M)^A : A}\ (\bot\ E)$$

The $\top$ type only has a single constructor $\langle\rangle$ with no arguments, so it has no destructor or elimination rule since there's no useful information to be extracted. On the other hand, the $\bot$ type has no constructor, but still requires a destructor to correspond to elimination - which we have as $\mathsf{expl}(M)$. It is so named since ($\bot$E) is commonly called the principle of explosion. Neither $\langle\rangle$ nor $\mathsf{expl}(M)$ have β-reductions, since there is either no destructor or constructor to cancel out.

In the next chapter, we will extend the Curry-Howard correspondence even further to cover quantifiers, relations and the construction of arbitrary mathematical objects. This extended theory becomes suitable for use as a foundations of mathematics, while still maintaining the computational interpretation.

4 ⊢ Martin-Löf's Intuitionistic Type Theory

> It no longer seems possible to distinguish the discipline of programming from constructive mathematics.
>
> –Per Martin-Löf

In the previous chapter, we observed how proofs in intuitionistic propositional logic can be identified with λ-terms via the Curry-Howard correspondence. However, propositional logic is not well-suited for mathematical reasoning. For this, we need first-order logic, or more generally the ability to reason about objects, functions and relations using quantifiers. This is where Martin-Löf's intuitionistic type theory (**MLTT**) comes in. In **MLTT**, we can not only represent proofs by λ-terms, but also mathematical objects such as the natural numbers and functions on these objects. Relations can be considered as types parametrized by the inhabitants/terms of some other type(s), and these too can be construed as functions on the type of types. Finally, quantifiers quantify over the inhabitants of a type, rather than some particular domain of objects as in first-order logic.

MLTT has endured and inspired the development of theorem provers and their underlying theory. For example, Coq, Lean & Agda are all based on some variant of **MLTT**. The computational nature of the proofs and mathematical objects mean that checking proofs for correctness can be expressed as a computation, specifically as the procedure of type checking, which can be carried out automatically via an algorithm.

Once we introduce **MLTT**, we will show that it cannot prove its own consistency, following the same lines of reasoning as in chapter 2. Some care will have to be taken for we can no longer take certain classical principles for granted, such as the law of excluded middle ($A \vee \neg A$) or double-negation elimination ($A \leftrightarrow \neg\neg A$). The proofs for the incompleteness theorems involve manipulating encodings of syntax. Due to the Curry-Howard correspondence, we may view these proofs as programs, thus demonstrating that provability is a form of metaprogramming.

4.1 Type Universes

In order to construct relations as functions on types, we first need a type of types $\mathcal{U}$ to serve as the function's codomain, with the idea that any type A can be typed as $A : \mathcal{U}$. Implicit in this idea is that types are now also terms, since types themselves have types. Unfortunately, we cannot type $\mathcal{U}$ by itself ($\mathcal{U} : \mathcal{U}$) because this leads to a paradox along the same lines as Russell's paradox of the set of all sets [28]. In order to maintain that a type is a term and must therefore have a type, we introduce an infinite, cumulative hierarchy of universes $\mathcal{U}_0 : \mathcal{U}_1 : \ldots$ More formally, this is represented by the rules

Definition 4.1 (Universe Formation & Cumulativity Rules [29])

$$\frac{}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}}\ (\mathcal{U}\text{–F}) \qquad \frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash A : \mathcal{U}_{i+1}}\ (\mathcal{U}\text{–cumul})$$

The universes are closed under type formers, e.g. if $A : \mathcal{U}_i$ and $B : \mathcal{U}_j$, then $A \to B : \mathcal{U}_{i \sqcup j}$, where $i \sqcup j$ picks out the larger number of i and j. For the work in this report, we are not going to use more than one universe, so we will often just call it $\mathcal{U}$ when we are not defining rules of the theory.

4.2 The Judgements of MLTT

4.2.1 Well-formedness of Contexts

When discussing natural deduction for **FOL** or for the simply typed lambda calculus, we were concerned only with defining $\cdot \vdash \cdot$ or $\cdot \vdash \cdot : \cdot$, which we call *judgements*. Informally, $\Gamma \vdash A$ is the judgement that A follows from the assumptions in Γ. With **MLTT**, we are still primarily concerned with $\cdot \vdash \cdot : \cdot$. However, as we will see in the following sections, types are now more complicated and are not necessarily well-formed just because they follow a certain syntactic structure. As a result, not all contexts Γ are well-formed either. In particular, types may now contain terms, so in a context $x : A, y : B$, the type B may refer to x. We express the well-formedness of Γ as the judgement Γ ctx, inductively defined by the following rules. The rules assert that each type in the context is well-formed with respect to the variables that come before.

Definition 4.2 (Well-Formedness Of Contexts [29])
Let $*$ denote the empty list.

$$\frac{}{* \text{ ctx}}\ (\text{ctx}-\text{emp}) \qquad \frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma, y : A \text{ ctx}}\ (\text{ctx} - \text{var})$$

Implicit in the (ctx − var) rule is the assumption that $\Gamma \vdash A : \mathcal{U}_i$ implies Γ ctx. Indeed, we will have to define the typing judgement $\vdash$ such that checks of Γ ctx are made in rules where they are necessary. For example, the rule ($\mathcal{U}$−F) ought to have been

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}}\ (\mathcal{U}-\text{F})$$

4.2.2 Definitional Equality

Another judgement we must have is actually one that we have seen before: β-reduction. This time, we express the reduction rules under context and typing, which means we only describe computation for well-typed terms. Additionally, we directly define the equality notion associated with the computation, rather than starting with a directed reduction relation and taking its equivalence closure. The judgement $\Gamma \vdash M \equiv N : A$, which reads the term M of type A is *definitionally equal* to the term N of type A (under the context Γ). We present here the basic rules of definitional equality, however leave the presentation of the actual *beta*-reduction to when we discuss the types. We will also omit congruence rules, for they can be inferred from the structure of the terms.

Definition 4.3 (Basic rules of Definitional Equality [29])
Definitional equality is an equivalence relation:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M \equiv M : A}\ (\equiv -\text{refl}) \qquad \frac{\Gamma \vdash M \equiv N : A}{\Gamma \vdash N \equiv M : A}\ (\equiv -\text{symm})$$

$$\frac{\Gamma \vdash M \equiv N : A \qquad \Gamma \vdash N \equiv O : A}{\Gamma \vdash M \equiv O : A}\ (\equiv -\text{trans})$$

Definitionally equal-types may replace each other.

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \equiv B : \mathcal{U}_i}{\Gamma \vdash M : B}\ (\equiv -\text{replace}_1)$$

$$\frac{\Gamma \vdash M \equiv N : A \qquad \Gamma \vdash A \equiv B : \mathcal{U}_i}{\Gamma \vdash M \equiv N : B}\ (\equiv -\text{replace}_2)$$

4.3 Inductive Types

Now, we move on to consider inductive types, where its inhabitants are generated by a combination of constructors. We have seen examples of this in the simply typed lambda calculus with the propositional connectives $\wedge$, $\vee$, and $\bot$. Here we give a more uniform treatment of these types under the framework of inductive types. Types are defined by describing 4 types of rules:

1. Formation rules describe how to construct the type.
2. Introduction rules determine how to construct inhabitants of the type, with one rule corresponding to each constructor.
3. Elimination rules determine how to use or "destroy" an arbitrary instance of the type. These also come with term formers called *destructors*, *eliminators* or more suggestively, *induction principles*.
4. Computation rules express how the constructors and destructor interact to cancel each other out, akin to β-reduction in the previous chapter. These are expressed as definitional equalities.

4.3.1 The Type of Natural Numbers

Let us build up our first type of mathematical objects: the natural numbers $\mathbb{N}$. $\mathbb{N}$ is trivially well-formed since its just a constant, so it inhabits any universe, giving us a simple formation rule. The terms of $\mathbb{N}$ are also simple to introduce: following **PA**, we have two constructors - z and $\mathrm{s}(\cdot)$.

Definition 4.4 (Formation & Introduction Rules For $\mathbb{N}$ [29])

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \mathbb{N} : \mathcal{U}_i}\ (\mathbb{N}F) \qquad \frac{\Gamma \text{ ctx}}{\Gamma \vdash \mathrm{z} : \mathbb{N}}\ (\mathbb{N}I_1) \qquad \frac{\Gamma \vdash M : \mathbb{N}}{\Gamma \vdash \mathrm{s}(M) : \mathbb{N}}\ (\mathbb{N}I_2)$$

The elimination rule of an inductive type is determined from the structure of the constructors. In general, the eliminator derives that a predicate $P(x)$ holds for any inhabitant x of the inductive type, assuming we can derive that $P(x)$ holds of the terms formed using each constructor. In other words, it is an induction principle for the type. For $\mathbb{N}$, this corresponds to the usual induction principle that we saw in chapter 2.

Definition 4.5 (Elimination Rule For $\mathbb{N}$ [29])

$$\frac{\Gamma, x : \mathbb{N} \vdash P : \mathcal{U}_i \qquad \Gamma \vdash z : P[\mathrm{z}/x] \qquad \Gamma, x : \mathbb{N}, p : P \vdash s : P[\mathrm{s}(x)/x] \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}}(x.P, z, xp.s, n) : P[n/x]}\ (\mathbb{N}\mathrm{E})$$

The notation $x.P$ and $xp.s$ binds the variable x in P and x, p in s. This is a generalisation of the binding mechanism of a λ abstraction to term formers with multiple subterms.

Notice that in the induction principle, we immediately apply the predicate to an arbitrary number n. This is because stating the resulting universal statement requires quantifiers, which we have not discussed. It is also always better to decouple the presentation of two different types, as this makes the theory more modular (i.e. we can remove or add new types without breaking old ones).

Each computation rule expresses how the induction principle cancels out when the arbitrary number n is formed using each constructor. Since there are two constructors for $\mathbb{N}$, it has two computation rules. While this is the same idea as the β-reduction of the previous chapter, this time we express the computation rules in terms of judgements, which means we only describe computation for well-typed

terms. Additionally, we directly define the equality notion associated with the computation, rather than starting with a directed reduction relation and taking its equivalence closure.

Definition 4.6 (Computation Rules For $\mathbb{N}$ [29])

$$\frac{\Gamma, x : \mathbb{N} \vdash P : \mathcal{U}_i \qquad \Gamma \vdash z : P[\mathrm{z}/x] \qquad \Gamma x : \mathbb{N}, p : P \vdash s : P[\mathrm{s}(x)/x]}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}}(x.P, z, xp.s, \mathrm{z}) \equiv z : P[\mathrm{z}/x]} \ (\mathbb{N}\mathrm{C}_1)$$

$$\frac{\Gamma, x : \mathbb{N} \vdash P : \mathcal{U}_i \qquad \Gamma \vdash z : P[\mathrm{z}/x] \qquad \Gamma x : \mathbb{N}, p : P \vdash s : P[\mathrm{s}(x)/x] \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}}(x.P, z, xp.s, \mathrm{s}(n)) \equiv s[n/x, \mathrm{ind}_{\mathbb{N}}(x.P, z, xp.s, n)/p] : P[\mathrm{s}(n)/x]} \ (\mathbb{N}\mathrm{C}_2)$$

4.3.2 Recasting Some Propositional Connectives as Inductive Types

We can recast the types of the propositional connectives we have seen as inductive types. For $\to$ and $\wedge$, we postpone their discussion to the next section as they are special cases of the quantifier types, and $\to$ cannot be placed under the framework of inductive types.

The type of $A \vee B$ is recast as the disjoint sum type $A + B$. The reason for this notation is that if A has n inhabitants and B has m inhabitants, then $A + B$ has $n + m$ inhabitants. The rules remain largely the same as in the simply typed lambda calculus.

Definition 4.7 (Rules For + [29])

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_j}{\Gamma \vdash A + B : \mathcal{U}_{i \sqcup j}} \ (+\mathrm{F}) \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash \mathrm{in}_1(M)^B : A + B} \ (+\mathrm{I}_1) \qquad \frac{\Gamma \vdash M : B}{\Gamma \vdash \mathrm{in}_2(M)^A : A + B} \ (+\mathrm{I}_2)$$

$$\frac{\Gamma, x : A + B \vdash P : \mathcal{U}_i \quad \Gamma, y : A \vdash N_1 : P[\mathrm{in}_1(y)/x] \quad \Gamma, z : B \vdash N_2 : P[\mathrm{in}_2(z)/x] \quad \Gamma \vdash M : A + B}{\Gamma \vdash \mathrm{ind}_+(x.P, y.N_1, z.N_2, M) : P[M/x]} \ (+\mathrm{E})$$

$$\frac{\Gamma, x : A + B \vdash P : \mathcal{U}_i \quad \Gamma, y : A \vdash N_1 : P[\mathrm{in}_1(y)/x] \quad \Gamma, z : B \vdash N_2 : P[\mathrm{in}_2(z)/x] \quad \Gamma \vdash M : A}{\mathrm{ind}_+(x.P, y.N_1, y.N_2, \mathrm{in}_1(M)) \equiv N_1[M/x] : P[\mathrm{in}_1(M)/x]} \ (+\mathrm{C}_1)$$

$$\frac{\Gamma, x : A + B \vdash P : \mathcal{U}_i \quad \Gamma, y : A \vdash N_1 : P[\mathrm{in}_1(y)/x] \quad \Gamma, z : B \vdash N_2 : P[\mathrm{in}_2(z)/x] \quad \Gamma \vdash M : B}{\mathrm{ind}_+(x.P, y.N_1, y.N_2, \mathrm{in}_2(M)) \equiv N_2[M/x] : P[\mathrm{in}_2(M)/x]} \ (+\mathrm{C}_2)$$

$\bot$ is recast as the empty type $\mathbb{0}$ with no constructors, while $\top$ is recast as the unit type $\mathbb{1}$ with a single constructor taking no arguments. This notation also follows from the number of inhabitants.

Definition 4.8 (Rules For $\mathbb{0}$ [29])
Because there are no constructors, $\mathbb{0}$ has no introduction rule nor computation rule.

$$\frac{\Gamma \ \mathrm{ctx}}{\Gamma \vdash \mathbb{0} : \mathcal{U}_i} \ (\mathbb{0}\mathrm{F}) \qquad \frac{\Gamma, x : \mathbb{0} \vdash P : \mathcal{U}_i \qquad \Gamma \vdash M : \mathbb{0}}{\Gamma \vdash \mathrm{ind}_{\mathbb{0}}(x.P, M) : P[M/x]} \ (\mathbb{0}\mathrm{E})$$

As with the simply typed lambda calculus before, $\neg M$ is simply shorthand for $M \to \mathbb{0}$.

Definition 4.9 (Rules For $\mathbb{1}$ [29])

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \mathbb{1} : \mathcal{U}_i}\ (\mathbb{1}\text{F}) \qquad \frac{\Gamma \text{ ctx}}{\Gamma \vdash \langle\rangle : \mathbb{1}}\ (\mathbb{1}\text{I})$$

$$\frac{\Gamma, x : \mathbb{1} \vdash P : \mathcal{U}_i \qquad \Gamma \vdash N : P[\langle\rangle/x] \qquad \Gamma \vdash M : \mathbb{1}}{\Gamma \vdash \text{ind}_{\mathbb{1}}(x.P, N, M) : P[M/x]}\ (\mathbb{1}\text{E})$$

$$\frac{\Gamma, x : \mathbb{1} \vdash P : \mathcal{U}_i \qquad \Gamma \vdash N : P[\langle\rangle/x]}{\Gamma \vdash \text{ind}_{\mathbb{1}}(x.P, N, \langle\rangle) \equiv N : P[\langle\rangle/x]}\ (\mathbb{1}\text{C})$$

Notice that $\mathbb{1}$ now comes with an elimination rule and destructor/induction principle. This is necessary because types can now depend on terms, so non-trivial types concerning $\langle\rangle$ can be constructed.

4.3.3 The Identity Type

Definitional equality provides us a notion of equality between terms, however this relation exists only in the meta-theory, i.e. it exists outside of the language of types and the lambda calculus. What we desire now is a type corresponding to the equality relation, commonly called the *propositional equality* or *identity* type. It should at least reflect definitional equality (i.e. if $M \equiv N$, then $M = N$ is inhabited) since definitional equality represents a "weak" form of equality determined only by syntactic reduction [30]. Since $M \equiv N$ only if they have the same type, equality must also be defined only between terms of the same type. Hence, for each type A, we should have a family of types $M =_A N$ indexed by $M, N : A$. We will omit the subscript when the type of the terms being identified is obvious.

We intend to define this family inductively, by giving it constructors. Note that this differs from the previous types we have considered - they have all been single types, rather than a family. Since equality is the quintessential equivalence relation, a reasonable choice for constructors express reflexivity, symmetry and transitivity. It turns out however, that we can derive symmetry and transitivity using the induction principle on the equality type with only one constructor for reflexivity [31].

Definition 4.10 (Formation & Introduction Rules For = [29])

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash M : A \qquad \Gamma \vdash N : A}{\Gamma \vdash M =_A N : \mathcal{U}_i}\ (=\text{F}) \qquad \frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash M : A}{\Gamma \vdash \text{refl}(M) : M =_A M}\ (=\text{I})$$

Notice that while the introduction rule only introduces inhabitants of $M = M$, we are able to use the replacement rule from Definition 4.3 along with the congruence rules of = to inhabit $M = N$ as long as $M \equiv N$. Thus, we satisfy the requirement of reflecting definitional equality.

The induction principle & elimination rule for the identity type is determined by the shape of its constructor. However, note that because we are defining a family of types, the induction principle is defined on the collection of all triples $M_1, M_2 : A$ and $M_3 : x = y$, rather than just a particular $M : x = y$. In the literature, this is commonly called *path induction*.

Definition 4.11 (Elimination & Computation Rules For = [29])

$$\frac{\begin{array}{c}\Gamma, x : A, y : A, z : x = y \vdash P : \mathcal{U}_i \\ \Gamma x : A \vdash N : P[x/x, x/y, \text{refl}(x)/z] \qquad \Gamma \vdash M_1 : A \qquad \Gamma \vdash M_2 : A \qquad \Gamma \vdash M_3 : M_1 = M_2\end{array}}{\Gamma \vdash \text{ind}_{=_A}(xyz.P, x.N, M_1, M_2, M_3) : P[M_1/x, M_2/y, M_3/z]}\ (=\text{E})$$

$$\frac{\Gamma, x : A, y : A, z : x = y \vdash P : \mathcal{U}_i \qquad \Gamma x : A \vdash N : P[x/x, x/y, \text{refl}(x)/z] \qquad \Gamma \vdash M : A}{\Gamma \vdash \text{ind}_{=_A}(xyz.P, x.N, M, M, \text{refl}(M)) \equiv N[M/x] : P[M/x, M/y, \text{refl}(M)/z]}\ (=\text{C})$$

We can then define symmetry using path induction. Defining transitivity requires quantifiers, so we will not discuss it here.

$$a : A, b : A, p : a = b \vdash \mathsf{ind}_{=_A}(xyz.y = x, x.\mathsf{refl}_x, a, b, p) : b = a$$

With the inductive types we discussed before, the induction principle effectively expresses the idea that all inhabitants of the type can be expressed using the constructors. This remains the case for the identity type **family** $x = y$ as x, y vary over the inhabitants of A, which means the family is uniquely generated by refl. Counterintuitively however, if we fix some $M : A$, we cannot show that $\mathsf{refl}(M)$ is the only inhabitant $M = M$. On the other hand, it is also not possible to exhibit such a counter-example inhabitant of $M = M$, so this property called "Uniqueness of Identity Proofs" (UIP) is independent of our version of **MLTT**.

One solution adds to **MLTT** the following rule that reflects identity as definitional equality [30], allowing one to prove UIP.

$$\frac{\Gamma \vdash N : M_1 =_A M_2}{\Gamma \vdash M_1 \equiv M_2 : A}$$

However, such a rule makes type checking (recursively) undecidable [32], where type checking is the problem of determining, given a term M and type A, whether $M : A$. We shall not have much to say about this rule because it prevents a straightforward treatment of incompleteness.

Another solution is to instead add an axiom rectifying the independence. The axiom can either enforce UIP [33] or embrace the existence of non-refl equality proofs by providing a way to generate such proofs. The most prominent example of the latter is homotopy type theory where identifications are interpreted as paths in homotopy theory [29]. For **MLTT** with such extensions, our treatment of incompleteness should be immediately applicable as it depends only on the basic aspects of **MLTT**.

4.4 Quantifiers as Dependent Type Formers

As with the propositional connectives, we start our investigation of quantifiers from their BHK interpretation, which is as follows.

Definition 4.12 (BHK Interpretation For The Quantifiers [22])

- A proof of $\forall x.\, P(x)$ is a construction that given an arbitrary object t, produces a proof of $P[t/x]$.
- A proof of $\exists x.\, P(x)$ constitutes a pair consisting of an object t and a proof of $P(t)$.

Notice that proofs of $\forall x.P(x)$ and $A \to B$ are the same sort of construction: functions. The only difference now is that instead of a proof of the antecedent A, the function takes as input arbitrary objects, and the formula $P(x)$ depends on this object. However, we have already identified the extended lambda calculus as a unified language for the expression of both objects and proofs. Therefore, as the Curry-Howard correspondence for $\forall$, we consider a generalisation of $A \to B$ in which the consequent type B may depend on the given object/proof of type A. We denote this by $\prod_{x:A} B(x)$, where x may occur free in the type $B(x)$. $A \to B$ is subsumed under the case when x does not occur free in B. Π is not an inductive type because the abstraction constructor binds a variable. Nevertheless, we may consider formation, introduction, elimination and computation rules for Π. The rules are similar to the simply-typed version we have already seen.

Definition 4.13 (Rules For Π [29])

$$\frac{\Gamma, \vdash A : \mathcal{U}_i \qquad \Gamma, x : A \vdash B : \mathcal{U}_j}{\Gamma \vdash \prod_{x:A} B : \mathcal{U}_{i \sqcup j}}\ (\Pi\mathsf{F}) \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A.\, M : \prod_{x:A} B}\ (\Pi\mathsf{I})$$

$$\frac{\Gamma \vdash M : \prod_{x:A} B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[N/x]}\ (\Pi\mathrm{E}) \qquad \frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x.\ M)\ N \equiv M[N/x] : B[N/x]}\ (\Pi\mathrm{C})$$

When we have repeated Πs in a term with binders of the same type, we abbreviate this by putting all the binders under a single Π. For example, $\prod_{x:A} \prod_{y:A} B$ is abbreviated as $\prod_{xy:A} B$.

As with $\forall$, we have already seen proofs of $\exists x.\ P(x)$ before - they are pairs, just like proofs of $A \wedge B$. We may therefore consider their Curry-Howard correspondence a generalisation of $A \wedge B$ where the type B depends on the object/proof of type A. We denote this by $\sum_{x:A} B(x)$, with $A \wedge B$ being subsumed under the case when x does not occur free in B, denoted $A \times B$. Unlike Π, Σ's constructor does not bind any variables, so it can be expressed as an inductive type.

Definition 4.14 (Rules For Σ [29])

$$\frac{\Gamma, \vdash A : \mathcal{U}_i \qquad \Gamma, x : A \vdash B : \mathcal{U}_j}{\Gamma \vdash \sum_{x:A} B : \mathcal{U}_{i \sqcup j}}\ (\Sigma\mathrm{F}) \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B[M/x]}{\Gamma \vdash \langle M, N \rangle : \sum_{x:A} B(x)}\ (\Sigma\mathrm{I})$$

$$\frac{\Gamma, x : \sum_{y:A} B \vdash P : \mathcal{U}_i \qquad \Gamma, y : A, z : B \vdash N : P[\langle y, z \rangle / x] \qquad \Gamma \vdash M : \sum_{y:A} B}{\Gamma \vdash \mathrm{ind}_\Sigma(x.P, yz.N, M) : P[M/x]}\ (\Sigma\mathrm{E})$$

$$\frac{\Gamma, x : \sum_{y:A} B \vdash P : \mathcal{U}_i \qquad \Gamma, y : A, z : B \vdash N : P[\langle y, z \rangle / x] \qquad \Gamma \vdash M_1 : A \qquad \Gamma, y : A \vdash M_2 : B}{\Gamma \vdash \mathrm{ind}_\Sigma(x.P, yz.N, \langle M_1, M_2 \rangle) \equiv N[M_1/y, M_2/z] : P[\langle M_1, M_2 \rangle / x]}\ (\Sigma\mathrm{C})$$

Repeated Σs are abbreviated in the same way as repeated Πs.

The projection functions we are familiar with from the simply-typed calculus may be re-obtained from the induction principle. If $\Gamma \vdash M : \sum_{y:A} B$, then define $\pi_1(M) :\equiv \mathrm{ind}_\Sigma(x.A, yz.y, M)$ and $\pi_2(M) :\equiv \mathrm{ind}_\Sigma(x.B[\pi_1(x)/y], yz.z, M)$. It can then be shown that

$$\Gamma \vdash \pi_1(M) : A \quad \text{and} \quad \Gamma \vdash \pi_2(M) : B[\pi_1(M)/x].$$

In classical logic, the quantifiers are interdefinable in the sense that $\exists x.\ A(x)$ can instead be defined as shorthand for $\neg\forall x.\ \neg A(x)$. This essentially boils down to double negation elimination and the way $\neg$ commutes with the quantifiers:

$$\mathbf{FOL} \vdash \neg\exists x.\ A(x) \leftrightarrow \forall x.\ \neg A(x)$$
$$\mathbf{FOL} \vdash \neg\forall x.\ A(x) \leftrightarrow \exists x.\ \neg A(x)$$

We know that intuitionistic logic (and by extension **MLTT**) does not allow double negation elimination, but it also does not allow the commutation $\Gamma \vdash \neg \prod_{x:B} A(x) \leftrightarrow \sum_{x:B} \neg A(x)$. This is because to prove $\sum_{x:B} \neg A(x)$ we need to exhibit an explicit term of type B, which $\neg \prod_{x:B} A(x)$ does not provide.

4.5 Incompleteness, Revisited

Now that we have introduced **MLTT**, we demonstrate that the incompleteness theorems also apply to it. The overall structure of the proofs do not change much, but we do have to work around the loss of certain classical principles, in particular to do with double negation elimination. We will also simply assume (but not specify) an encoding of terms (including types) and judgements as natural numbers. Finally, we will assume we have the primitive recursive function $\mathrm{diag}(x)$ and relation $\mathrm{prf}_{\mathbf{MLTT}}(x, y, z)$ which act on encodings of terms.

The function $\mathrm{diag}(x)$ takes the code of a one-place predicate $P : \mathbb{N} \to \mathcal{U}$ and produces the code of $P\ \ulcorner P \urcorner$. The relation $\mathrm{prf}_{\mathbf{MLTT}}(x, y, z)$ corresponds to type checking for **MLTT** by determining whether y codes

a derivation δ and x and y code terms M and A such that δ is a valid derivation with the conclusion $* \vdash M : A$. Recall that in order to prove the incompleteness theorems for **PA**, we needed an internal representation of $\text{prf}_{\mathbf{PA}}$, which exists only if $\text{prf}_{\mathbf{PA}}$ is recursively decidable. The decidability of $\text{prf}_{\mathbf{MLTT}}$ follows from the decidability of type checking for **MLTT**, as established in [32].

4.5.1 Pattern Matching Definitions

The definitions for $\pi_1(\cdot)$ and $\pi_2(\cdot)$ using induction principles are difficult to decipher. To prove the incompleteness theorems, we will have to work inside **MLTT** to some capacity and so desire a better way to define new functions using pattern matching instead of induction, similar to the mechanism found in Haskell or Agda. For example, we can instead define $\pi_1(\cdot)$ as a function in the following way:

$$\Gamma \vdash \pi_1 : (\textstyle\sum_{y:A} B) \to A$$
$$\pi_1\ \langle y, z\rangle :\equiv y$$

When Γ is empty, we will omit the $\vdash$ entirely. With this syntax, we can also make recursive calls, provided that the recursion follows the structure of the constructor being pattern matched. Additionally, certain usage of pattern matching on identity types allow a form of UIP to be proven [34]. Agda contains checks disallowing unsound usage of pattern matching [35], but we do not have the luxury of an automated checker. For this reason, we will default back to induction principles when dealing with identity types.

4.5.2 Representing Recursive Functions & Relations

We begin by establishing the result that **MLTT** represents all & only the recursive functions, following [14]. We use an alternate definition of total recursive functions which restricts the use of $\mu[f]$ only when we can ensure termination of the unbounded search for all possible inputs. It also removes the need for $\text{rec}[f, g]$, as recursion can be simulated using $\mu[f]$.

Definition 4.15 (Alternate Definition Of Recursive Functions [14])
The recursive functions are defined inductively:

1. The functions zero, succ and $\text{pr}^{\text{n}}_{\text{i}}$ (for each natural numbers n and $1 \leq i \leq n$) are recursive. So are $\text{add}(x_1, x_2) \triangleq x_1 + x_2$ and $\text{mult}(x_1, x_2) \triangleq x_1 \times x_2$.
2. The characteristic function $\chi_{=}(x_1, x_2) \triangleq$ if $x_1 = x_2$ then 1 else 0 of equality is recursive.
3. If $f(x_1, \ldots, x_k)$ and $g_1(x_1, \ldots, x_n) \ldots g_k(x_1, \ldots, x_n)$ are recursive, then so is $\text{comp}[f, g_1, \ldots, g_k]$.
4. If $f(x_1, \ldots x_k, y)$ is recursive and for all natural numbers $n_1, \ldots, n_k$ there is a natural number m s.t. $f(n_1, \ldots, n_k, m) = 0$, then $\mu[f]$ is recursive.

With this inductive definition, we can prove by induction on the structure of recursive functions that they are representable. The definition of representable remains the same: we just have to replace the connectives by their **MLTT** version.

Definition 4.16 (Representable Functions & Relations in MLTT)
1. A function $f(x_1, \ldots, x_k)$ is represented by the **MLTT** predicate $\overline{f} : \underbrace{\mathbb{N} \to \ldots \to \mathbb{N}}_{k+1 \text{ times}} \to \mathcal{U}$ iff for every natural number $n_1, \ldots, n_k, m$

$f(n_1, \ldots n_k, m)$ implies there is a term M_f such that $* \vdash M_f : \prod_{(y:\mathbb{N})}(\overline{f}\ \overline{n_1}\ \ldots\ \overline{n_k}\ y \leftrightarrow y = \overline{m})$

2. A relation $R(x_1, \ldots, x_k)$ is represented by $\overline{R} : \underbrace{\mathbb{N} \to \ldots \to \mathbb{N}}_{k \text{ times}} \to \mathcal{U}$ iff for every $n_1, \ldots, n_k$,

if $R(n_1, \ldots, n_k)$ holds then there is a term M_R s.t. $* \vdash M_R : \overline{R}\ \overline{n_1}\ \ldots\ \overline{n_k}$
if $R(n_1, \ldots, n_k)$ does not hold then there is a term M_R s.t. $* \vdash M_R : \neg(\overline{R}\ \overline{n_1}\ \ldots\ \overline{n_k})$

We separate the proof by induction into a series of lemmas, one for each case.

Lemma 4.17
The functions zero, succ, pr^n_i, add and mult are representable.

Proof. For each function f, the proof proceeds by constructing representing predicates $\overline{f}$ such that whenever $f(n_1, \ldots, n_k) = m$, $\overline{f}\ \overline{n_1}\ \ldots\ \overline{n_k}\ y$ is definitionally equal to $y = \overline{m}$. With that, we can derive

$$* \vdash \lambda y.\ \langle \lambda x.\ x, \lambda x.\ x \rangle : \prod_{y:\mathbb{N}}(\overline{f}\ \overline{n_1}\ \ldots\ \overline{n_k}\ y \leftrightarrow y = \overline{m})$$

1. $\mathsf{zero}(x) \triangleq 0$ is represented by $\overline{\mathsf{zero}} :\equiv \lambda xy : \mathbb{N}.\ y = \mathsf{z}$ because for any n, $(\overline{\mathsf{zero}}\ \overline{n}\ y) \equiv (y = \overline{0})$. We can see this by simply reducing the LHS and noting that $\overline{0}$ is shorthand for z.
2. $\mathsf{succ}(x) \triangleq x+1$ is represented by $\overline{\mathsf{succ}} :\equiv \lambda xy : \mathbb{N}.\ y = \mathsf{s}(x)$. As before, reduce the LHS and unfold $\overline{n+1}$ to see that $\overline{\mathsf{succ}}\ \overline{n}\ y \equiv y = \overline{n+1}$.
3. $\mathsf{pr}^n_i(x_1 \ldots x_n)$ is represented by $\overline{\mathsf{pr}^n_i} :\equiv \lambda x_1 \ldots x_n\ y : \mathbb{N}.\ y = x_i$.
4. To represent $\mathsf{add}(x_1, x_2) \triangleq x_1 + x_2$, we construct an addition function inside **MLTT**.

$$\begin{aligned} &\mathsf{plus} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ &\mathsf{plus}\ x\ \mathsf{z} \quad :\equiv x \\ &\mathsf{plus}\ x\ \mathsf{s}(y) :\equiv \mathsf{s}(\mathsf{plus}\ x\ y) \end{aligned}$$

   Then, we define $\overline{\mathsf{add}} :\equiv \lambda x_1 x_2 y.\ y = \mathsf{plus}\ x_1\ x_2$. To see that $\overline{\mathsf{add}}\ \overline{n_1}\ \overline{n_2}\ y \equiv y = \overline{n_1 + n_2}$, induce on n_2.
5. $\mathsf{mult}(x_1, x_2) \triangleq x_1 \times x_2$ is represented by $\overline{\mathsf{mult}} :\equiv \lambda x_1 x_2 y.\ y = \mathsf{times}\ x_1\ x_2$, where

$$\begin{aligned} &\mathsf{times} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ &\mathsf{times}\ x\ \mathsf{z} \quad :\equiv \mathsf{z} \\ &\mathsf{times}\ x\ \mathsf{s}(y) :\equiv \mathsf{plus}\ (\mathsf{times}\ x\ y)\ x \end{aligned}$$

   Finally, induce on n_2 to see that $\overline{\mathsf{times}}\ \overline{n_1}\ \overline{n_2}\ y \equiv y = \overline{n_1 \times n_2}$. ⊣

For the next case of the characteristic function of equality, we will have to reason about the identity type on natural numbers, in particular using it to derive a contradiction when we have a inhabitant of $\overline{n_1} = \overline{n_2}$. This is possible using path induction, but having to utilize path induction every time leads to a proof with less clarity. Instead, we establish a specific equality type on $\mathbb{N}$ that is definitionally equal to either $\mathbb{0}$ or $\mathbb{1}$, depending on whether they are the same number or not.

Definition 4.18 (Observational Equality on $\mathbb{N}$ [31])
The *observational equality* $\mathsf{Eq}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N} \to \mathcal{U}$ is defined by induction on both arguments.

$$\begin{array}{ll} \mathsf{Eq}_{\mathbb{N}}\ \mathsf{z}\ \mathsf{z} :\equiv \mathbb{1} & \mathsf{Eq}_{\mathbb{N}}\ \mathsf{z}\ \mathsf{s}(x) :\equiv \mathbb{0} \\ \mathsf{Eq}_{\mathbb{N}}\ \mathsf{s}(y)\ \mathsf{z} :\equiv \mathbb{0} & \mathsf{Eq}_{\mathbb{N}}\ \mathsf{s}(y)\ \mathsf{s}(x) :\equiv \mathsf{Eq}_{\mathbb{N}}\ y\ x \end{array}$$

The use of path induction is encapsulated in establishing a correspondence between the identity type and observational equality [31].

$$\text{obs-of-id} : \textstyle\prod_{x_1 x_2 : \mathbb{N}} (x_1 = x_2 \to \mathsf{Eq}_{\mathbb{N}}\ x_1\ x_2) \quad \text{and} \quad \text{id-of-obs} : \textstyle\prod_{x_1 x_2 : \mathbb{N}} (\mathsf{Eq}_{\mathbb{N}}\ x_1\ x_2 \to x_1 = x_2)$$

Because the first two arguments x_1 and x_2 can be discerned from the type of the third argument, we will omit them when using obs-of-id and id-of-obs Equipped with observational equality, we can now move on to the next case.

Lemma 4.19
The function $\chi_=$ is representable.

Proof. $\chi_=(x_1, x_2)$ is defined differently depending on whether $x_1 = x_2$ or not, so we have to represent this piecewise reasoning in the representing predicate.

$$\overline{\chi_=} :\equiv \lambda x_1 x_2 y : \mathbb{N}.\ (x_1 = x_2 \to y = \overline{1}) \times ((\neg x_1 = x_2) \to y = \overline{0})$$

Now, suppose we have n_1, n_2, m such that $\chi_=(x_1, n_2) = m$. If $n_1 = n_2$, then $m = 1$, and $\overline{n_1}$ is the exact same term as $\overline{n_2}$. We can therefore establish that $\overline{\chi_=}$ does represent $\chi_=$, splitting the proof into two:

$$\begin{aligned} y : \mathbb{N} \vdash \text{l-to-r} &: \overline{\chi_=}\ \overline{n_1}\ \overline{n_2}\ y \to y = \overline{m} \\ \text{l-to-r}\ x &:\equiv \pi_1(x)\ \mathsf{refl}_{n_1} \end{aligned}$$

$$\begin{aligned} y : \mathbb{N} \vdash \text{r-to-l} &: y = \overline{m} \to \overline{\chi_=}\ \overline{n_1}\ \overline{n_2}\ y \\ \text{r-to-l}\ x &:\equiv \langle \lambda r.\ x, \lambda r.\ \mathsf{ind}_{\mathbb{0}}(z.y = \overline{0}, r\ \mathsf{refl}_{n_1}) \rangle \end{aligned}$$

If $n_1 \neq n_2$, then $m = 0$ and $\overline{n_1}$ is syntactically distinct from $\overline{n_2}$. We establish the representation by splitting the proof into two as well. This is where observational equality comes into play, as we use it to obtain an instance of $\mathsf{Eq}_{\mathbb{N}}\ \overline{n_1}\ \overline{n_2}$, which in this case is definitionally equal to $\mathbb{0}$.

$$\begin{aligned} y : \mathbb{N} \vdash \text{l-to-r} &: \overline{\chi_=}\ \overline{n_1}\ \overline{n_2}\ y \to y = \overline{m} \\ \text{l-to-r}\ x &:\equiv \pi_2(x)\ (\lambda r : \overline{n_1} = \overline{n_2}.\ \text{obs-of-id}\ r) \end{aligned}$$

$$\begin{aligned} y : \mathbb{N} \vdash \text{r-to-l} &: y = \overline{m} \to \overline{\chi_=}\ \overline{n_1}\ \overline{n_2}\ y \\ \text{r-to-l}\ x &:\equiv \langle \lambda r : \overline{n_1} = \overline{n_2}.\ \mathsf{ind}_{\mathbb{0}}(z.y = \overline{1}, \text{obs-of-id}\ r), \lambda r.\ x \rangle \end{aligned}$$

⊣

For the next case, we need some more mechanisms for identity types. In particular, we need to substitute an equal term for another in a predicate. This is encapsulated by the transport function [31].

$$\mathsf{tr} : \textstyle\prod_{B : \mathbb{N} \to \mathcal{U}} \prod_{x y : \mathbb{N}} (x = y \to Bx \to By)$$

As before, we will omit the first three arguments when using tr because they can be inferred from the type of the remaining arguments.

We also require an alternative induction principle called *based* path induction [29]. It is based because we fix one side of the identity before doing the induction. We use the universally quantified version of the induction principle.

$$ind'_{=} : \prod_{(a:A)} \prod_{(P:\ \prod_{(x:A)} a = x \to \mathcal{U})} (P\ x\ \mathrm{refl}_a) \to \prod_{x:A} \prod_{p:a=x} P\ x\ p$$

Lemma 4.20
If $f(x_1, \ldots, x_k)$ and $g_1(x_1, \ldots, x_n) \ldots g_k(x_1, \ldots, x_l)$ are representable, then so is $\mathrm{comp}[f, g_1, \ldots, g_k]$.

Proof. For clarity and brevity, we will demonstrate the proof for when f has only one argument, since the proof can be easily generalised anyway.

Let us denote the representability term for f and g in the following way.

$$\begin{array}{lll} f(n) = m & \text{implies} & * \vdash M_{f(n)=m} : \prod_{(y:\mathbb{N})} (\overline{f}\ \overline{n}\ y \leftrightarrow y = \overline{m}) \\ g(n_1, \ldots, n_k) = m & \text{implies} & * \vdash M_{g(n_1,\ldots,n_k)=m} : \prod_{(y:\mathbb{N})} (\overline{g}\ \overline{n_1}\ \ldots\ \overline{n_k}\ y \leftrightarrow y = \overline{m}) \end{array}$$

Let $\overline{\mathrm{comp}[f,g]} :\equiv \lambda x_1 \ldots x_k y : \mathbb{N}.\ \sum_{z:\mathbb{N}} (\overline{g}\ x_1\ \ldots\ x_k\ z) \times (\overline{f}\ z\ y)$, and suppose that $\mathrm{comp}[f,g](n_1 \ldots n_k) = m$. Letting $p = g(n_1, \ldots, n_k)$, we have that $f(p) = m$ As before, we split the proof of representability into two.

$$\begin{array}{l} y : \mathbb{N} \vdash \text{l-to-r} : \overline{(\mathrm{comp}[f,g]}\ \overline{n_1}\ \ldots\ \overline{n_k}\ y) \to y = \overline{m} \\ \qquad \text{l-to-r}\ \langle z, \langle x_g, x_f \rangle \rangle :\equiv \pi_1(M_{f(p)=m}\ y)\ \underbrace{(\mathrm{tr}\ (\pi_1(M_{g(n_1 \ldots n_k)=p}\ z)\ x_g)\ x_f)}_{\text{Substitute } z=\overline{p} \text{ into } x_f : \overline{f}\ z\ y} \end{array}$$

For the other half of the proof, we will need to use based path induction, omitting the predicate because it can be inferred from the return type. We also use symm, a universally quantified version of the symmetry of identity types we defined previously.

$$\begin{array}{l} y : \mathbb{N} \vdash \text{r-to-l} : y = \overline{m} \to \overline{(\mathrm{comp}[f,g]}\ \overline{n_1}\ \ldots\ \overline{n_k}\ y) \\ \qquad \text{r-to-l}\ x :\equiv \mathrm{ind}'_{=}\ \overline{m}\ \_\ \langle \overline{p}, \langle \pi_2(M_{g(n_1 \ldots n_k)=p}\ \overline{p})\ \mathrm{refl}_{\overline{p}}, \pi_2(M_{f(p)=m}\ \overline{m})\ \mathrm{refl}_{\overline{m}} \rangle \rangle\ y\ (\mathrm{symm}\ x) \quad \dashv \end{array}$$

The next and final case in the inductive proof is considerably more complicated, requiring us to define the less-than relation inside **MLTT**. We use the usual infix mathematical notation $<$, for readability.

$$\begin{array}{l} \_ < \_ : \mathbb{N} \to \mathbb{N} \to \mathcal{U} \\ x < y :\equiv \sum_{k:\mathbb{N}} (\neg k = \mathrm{z}) \times (\mathrm{plus}\ x\ k = y) \end{array}$$

we also require the following lemmas about natural numbers.

Lemma 4.21 (Canonical Form)
For every natural number n, there exists a term canon such that $* \vdash \mathrm{canon} : \prod_{x:\mathbb{N}} (x < \overline{n} \to (x = \overline{0} + \ldots + x = \overline{n-1}))$. When $n = 0$, the empty disjunction corresponds to $\mathbb{0}$.

Proof. See Appendix 1.1. ⊣

Lemma 4.22
There exists a term tricho such that $* \vdash \text{tricho} : \prod_{xy:\mathbb{N}}((y < x + x < y) + x = y)$

Proof. See Appendix 1.2 ⊣

Lemma 4.23
If $f(x_1, \ldots, x_k, y)$ is representable and for all natural numbers $n_1, \ldots, n_k$ there exists m such that $f(n_1, \ldots, n_k, m) = 0$, then $\mu[f]$ is also representable.

Proof. For brevity, we present only the proof for $k = 1$. We will also only describe the proof terms informally, relying on the reader to explicitly construct the proof term from the informal description. As before, let us denote the representability term for f in the following way:

$$f(n_1, n_2) = m \text{ implies } * \vdash M_{f(n_1,n_2)=m} : \prod_{(y:\mathbb{N})}(\overline{f}\ \overline{n_1}\ \overline{n_2}\ y \leftrightarrow y = \overline{m}).$$

To represent $\mu[f]$, we simply describe what it means to do an unbounded search, in the language of **MLTT**.

$$\overline{\mu[f]} :\equiv \lambda xy : \mathbb{N}.\ (\overline{f}\ x\ y\ \overline{0}) \times \prod_{i:\mathbb{N}}(i < y \to \neg(\overline{f}\ x\ i\ \overline{0}))$$

If $\mu[f](n) = m$, then $f(n, m) = 0$ and for all $i < m$, $f(n, i) \neq 0$. This means that there exists $j_i \neq 0$ such that $f(n, i) = j_i$.

Now, to demonstrate the representability we have to demonstrate a term of type

$$\prod_{y:\mathbb{N}}(\overline{\mu[f]}\ \overline{n}\ y \leftrightarrow y = \overline{m})$$

. For the right-to-left direction, we assume $y = \overline{m}$ and first have to show that $\overline{f}\ \overline{n}\ y\ \overline{0}$. This can be achieved by using the transport function to substitute $\overline{m}$ for y in $\pi_2(M_{f(n,m)=0}\ \overline{0})\ \text{refl}_{\overline{0}}$. We also have to show $\prod_{i:\mathbb{N}}(i < y \to (\overline{f}\ \overline{n}\ i\ \overline{0} \to \mathbb{0}))$. For this second goal, first take an arbitrary i with $i < y$ and $\overline{f}\ \overline{n}\ i\ \overline{0}$, which means the goal becomes to prove $\mathbb{0}$ AKA to derive a contradiction. We may now substitute in $\overline{m}$ for y, so $i < \overline{m}$. Using lemma 4.21, we have a term of type $i = \overline{0} + \ldots + i = \overline{m-1}$. If $m = 0$, then this is the same as $\mathbb{0}$, and we are done. Otherwise, we have to eliminate on the disjunctions, essentially doing a proof by cases with m cases. Luckily, each case follows the same steps, as follows. Since $i = \overline{l}$ for some l less than m, we can substitute to obtain $\overline{f}\ \overline{n}\ \overline{l}\ \overline{0}$. But now, we can use $M_{f(n,l)=j_l}$ to obtain $\overline{0} = \overline{j_l}$. Of course, we know that $j_l \neq 0$, so we can use observational equality to derive an instance of $\mathbb{0}$.

For the left-to-right direction, we assume

$$a : \overline{f}\ \overline{n}\ y\ \overline{0}$$
$$b : \prod_{i:\mathbb{N}}(i < y \to (\overline{f}\ \overline{n}\ i\ \overline{0} \to \mathbb{0}))$$

in order to show $y = \overline{m}$. For this proof, we utilise lemma 4.22 to obtain a trichotomy between y and $\overline{m}$, showing that both the case when $\overline{m} < y$ and $y < \overline{m}$ lead to contradictions. Consider what happens if $\overline{m} < y$: by assumption b we have $\overline{f}\ \overline{n}\ \overline{m}\ \overline{0} \to \mathbb{0}$. Of course, we can derive $\overline{f}\ overline n\ \overline{m}\ \overline{0}$ by $M_{f(n,m)=0}$, so we have a contradiction. On the other hand, if $y < \overline{m}$ then we can derive a contradiction via lemma 4.21 and a, in the same way as in the proof of the right-to-left direction. Therefore, the only possibility is $y = \overline{m}$. ⊣

Taking the above lemmas together, we may conclude the following theorem by induction on the structure of recursive functions.

Theorem 4.24 (Representability of Recursive Functions)
Given a function $f(x_1, \ldots, x_k)$ on natural numbers, if f is recursive then it is representable.

As a corollary, we obtain a similar result for recursively decidable relations.

Corollary 4.25
A relation $R(x_1, \ldots, x_k)$ on natural numbers is representable if it is recursively decidable.

Proof. Since R is recursively decidable, its characteristic function χ_R is recursive. By theorem 4.24, χ_R is representable, which means we can define $\overline{R} :\equiv \lambda x_1 \ldots x_k : \mathbb{N}.\ (\overline{\chi_R}\ x_1\ \ldots\ x_k\ \overline{1})$.

If $R(n_1 \ldots n_k)$ holds, then $\chi_R(n_1, \ldots, n_k) = 1$ so we can derive

$$* \vdash \pi_2(M_{\chi_R(n_1,\ldots,n_k)=1}\ \overline{1})\ \mathsf{refl}_{\overline{1}} : \overline{R}\ \overline{n_1}\ \ldots\ \overline{n_k}$$

If If $R(n_1 \ldots n_k)$ does not hold, then $\chi_R(n_1, \ldots, n_k) = 0$ so we instead derive

$$* \vdash \lambda r.\ \text{obs-of-id}(\pi_1(M_{\chi_R(n_1,\ldots,n_k)=0}\ \overline{1})\ r) : \overline{R}\ \overline{n_1}\ \ldots\ \overline{n_k} \to \mathbb{0}$$

While the representability of recursive functions takes some work to establish, we briefly observe that primitive recursive functions have a straightforward representation in **MLTT** since **MLTT** allows us to define functions, unlike **PA**. We can express this in an alternate definition of representability.

Definition 4.26 (Functional Representability)
A function $f(x_1, \ldots, x_k)$ is represented by the **MLTT** function $f^* : \underbrace{\mathbb{N} \to \ldots \to \mathbb{N}}_{k \text{ times}} \to \mathbb{N}$ iff for every natural number $n_1, \ldots, n_k, m$

$$f(n_1, \ldots n_k, m) \text{ implies } * \vdash f^*\ \overline{n_1}\ \ldots\ \overline{n_k} \equiv \overline{m} : \mathbb{N}.$$

Every primitive recursive function is clearly functionally representable, since we are able to express the zero, successor and projection functions in **MLTT**. Additionally, composition of functions and recursion on $\mathbb{N}$ is expressible in **MLTT**. Functional representation implies the usual notion of representation.

Theorem 4.27 (Functional Representability implies Representability)
If f is functionally represented, then it is represented.

Proof. Let $\overline{f} :\equiv \lambda x_1 \ldots x_k y.\ (y = f^*\ x_1\ \ldots\ x_k)$. Then we have

$$* \vdash \lambda y.\ \langle \lambda x.\ x, \lambda x.\ x \rangle : \textstyle\prod_{y:\mathbb{N}} (\overline{f}\ \overline{n_1}\ \ldots\ \overline{n_k}\ y) \leftrightarrow (y = \overline{m})$$

which is valid because $\overline{f}\ \overline{n_1}\ \ldots\ \overline{n_k}\ y \equiv\ (y = f^*\ \overline{n_1}\ \ldots\ \overline{n_k}) \equiv\ (y = \overline{m})$. ⊣

4.5.3 The Incompleteness of MLTT

In order to discuss **PA** and **MLTT** together without confusion, we write **MLTT** $\vdash M : A$ and **MLTT** $\vdash A$ to say that M is a term of type A and A has an inhabiting term, in the empty context. Before we consider the incompleteness theorems, we must prove the fixed-point property for **MLTT**. There is not much to reconsider here as we can follow the same lines of reasoning as for **PA**.

Lemma 4.28 (Fixed-point Property)
For any predicate $B : \mathbb{N} \to \mathcal{U}$, there is a type A such that **MLTT** $\vdash A \leftrightarrow B\ \ulcorner A \urcorner$.

Proof. In order to define A, we first define an auxiliary predicate which diagonalises the argument and applies B to the diagonalisation. The diagonalisation is done using the functional representation of the primitive recursive diag function, which takes the code of a one-place predicate P and produces the code of its diagonalisation $P\ \ulcorner P \urcorner$.

$$E :\equiv \lambda x.\ B\ (\mathrm{diag}^*\ x)$$

Of note here is that E is itself a one-place predicate, and so may be diagonalised. A diagonalisation of E should be equivalent to B applied to a diagonalisation of E, which is exactly what we need. Hence, we define $A :\equiv E\ \ulcorner E \urcorner$. With the functional representation of diag, we can in fact establish that A and $B\ \ulcorner A \urcorner$ are definitionally equal, rather than just equivalent.

$$\begin{aligned}
A &\equiv E\ \ulcorner E \urcorner && \text{By definition of } A \\
&\equiv B\ (\mathrm{diag}^*\ \ulcorner E \urcorner) && \text{Unfolding the definition of } E \text{ and applying the abstraction} \\
&\equiv B\ \ulcorner E\ \ulcorner E \urcorner \urcorner && \text{By definition of } \mathrm{diag}^* \\
&\equiv B\ \ulcorner A \urcorner && \text{By definition of } A
\end{aligned}$$

Now, the structure of the incompleteness proofs for **MLTT** remain the same as for **PA**. However, in our proof for **PA**'s first theorem, in order to show **PA** $\nvdash \neg G_{\mathbf{PA}}$ we made crucial use of double-negation elimination in the application of the fixed-point property. We can no longer use double-negation elimination, and as a result we have to re-evaluate our definition of ω-consistency. The following definition is classically equivalent to our original definition, but not so intuitionistically.

Definition 4.29 (ω-consistency For MLTT [16])
MLTT is ω-consistent iff for all $A : \mathbb{N} \to \mathcal{U}$, if **MLTT** can derive $\neg \prod_{x:\mathbb{N}} A\ x$ then it cannot derive $A\ \overline{m}$ for some m.

This version of ω-consistency still implies consistency since if **MLTT** were inconsistent, it can derive $A\ \overline{m}$ and $\neg(A\ \overline{m})$ for all A and m. However, the latter implies $\vdash_{\mathbf{MLTT}} \neg \prod_{x:\mathbb{N}} A\ x$, which in combination with the former means **MLTT** is ω-inconsistent.

As in **PA**, we define Prov $:\equiv \lambda z : \mathbb{N}.\ \sum_{xy:\mathbb{N}} \overline{\mathsf{prf}_{\mathbf{MLTT}}}(x, y, z)$ and obtain $G_{\mathbf{MLTT}}$ as the fixed-point of Prov, allowing us to prove the first incompleteness theorem again.

Theorem 4.30 (First Incompleteness Theorem)
If **MLTT** is ω-consistent, then $G_{\mathbf{MLTT}}$ is independent, where $G_{\mathbf{MLTT}}$ is the fixed-point of Prov.

Proof.

(**MLTT** $\nvdash G_{\mathbf{MLTT}}$) Essentially the same as for **PA** (theorem 2.18).

(**MLTT** $\nvdash \neg G_{\mathbf{MLTT}}$) Suppose for a contradiction that **MLTT** $\vdash \neg G$. By the fixed-point property, **MLTT** $\vdash \neg\neg \mathsf{Prov}\ \ulcorner G \urcorner$ which is intuitionistically equivalent to

$$\mathbf{PA} \vdash \neg \prod_{(xy:\mathbb{N})} \neg \overline{\mathsf{prf}_{\mathbf{MLTT}}}(x, y, \ulcorner G \urcorner). \tag{4.1}$$

Since **MLTT** is consistent, **MLTT** $\nvdash G_{\mathbf{MLTT}}$ implying no number codes a derivation of $G_{\mathbf{MLTT}}$. Hence, for any n, **MLTT** $\vdash \neg \overline{\mathsf{prf}_{\mathbf{MLTT}}}\ \overline{n}\ \ulcorner G \urcorner$. Combining this with (4.1) shows that **MLTT** is ω-inconsistent, contradicting our assumption. ⊣

The proof for the second incompleteness theorem then proceeds in much the same way assuming that our Prov satisfies the Hilbert-Bernays-Löb conditions. As with the first theorem, we have to slightly adjust the proof to account for our new definition of ω-incompleteness. The proof of the formalised first theorem remains valid intuitionistically.

In our proof of the incompleteness theorems, we were constructing terms in **MLTT** whose express purpose is to represent recursively computable manipulations of encodings of **MLTT** terms & derivations. Because of **MLTT**'s computational interpretation by the Curry-Howard correspondence, we were quite literally constructing metaprograms. However, it was a tedious process. In the next chapters, we investigate metaprogramming primitives that can serve as a more useable interface for the manipulation of code. Another way of viewing this is that we will be using our work here with provability to logically justify metaprogramming.

5 ⊢ Modal Logics for Provability & Metaprogramming

> I don't think outside the box, I think of what I can do with the box.
>
> –Henri Matisse

Modal logics are classical/intuitionistic logics equipped with additional *modal* operators. A modal is an expression that qualifies the truth of a statement [36], where "qualify" here means "to make less absolute". In particular, we are concerned with propositional logics (both classical and intuitionistic) equipped with the unary modal operator $\Box$. The formula $\Box A$ reads "it is necessary that A holds", although here "necessary" does not mean "logically necessary", but rather a form of qualified necessity. For example, in epistemic logic [37] $\Box A$ reads as "the agent knows that A holds", while in deontic logic [38] it reads as "A is obligated to hold". Depending on which qualification one is interested in, the behaviour of $\Box$ changes.

As we will soon see, the Hilbert-Bernays-Löb provability conditions show that $\mathsf{Prov}_{\mathbf{PA}}(\ulcorner \cdot \urcorner)$ can be viewed as a modal necessity operator. Intuitively, we can read $\mathsf{Prov}_{\mathbf{PA}}(\ulcorner A \urcorner)$ as "**PA** knows that A holds"[1]. Defining a modal logic to capture the behavior of $\mathsf{Prov}_{\mathbf{PA}}(\ulcorner \cdot \urcorner)$ in a simpler system allows us to reason more clearly about it, and to abstract away from the particularities of how $\mathsf{Prov}_{\mathbf{PA}}(\ulcorner \cdot \urcorner)$ is defined. Assuming the same modal nature holds for $\mathsf{Prov}_{\mathbf{MLTT}}(\ulcorner \cdot \urcorner)$, we may also employ modal logic to reason about provability in **MLTT**, at least informally.

Independent of developments in modal logic for provability, practitioners of metaprogramming have observed that modal logic serves as a good guiding principle and abstraction for assigning types to metaprograms. In particular, the type $\Box A$ is inhabited by code of terms with type A. This interpretation of the modality is particularly vague, with no specification of a particular encoding. This serves to elucidate the core concepts of metaprogramming and abstract away from particular encodings of syntax.

Despite this however, we desire a more logically rigorous interpretation of the type $\Box A$ in metaprogramming. Using the modal logics associated with provability and metaprogramming, we investigate the viability of an interpretation of the type $\Box A$ as the provability term $\mathsf{Prov}(\ulcorner A \urcorner)$. Because there are many logics and proof systems to discuss in this chapter, we will not give full characterisations of most of these systems, focusing instead on highlighting their key features.

5.1 Axiomatic Deduction Systems for Modal Logic

Over time, there has been a proliferation of modal logics because the behavior of $\Box$ changes depending on the particular concept we are trying to model. One way to categorise classical propositional modal logics is to isolate certain axioms which in combination with principles of classical reasoning, characterises the logic. For this, we use axiomatic systems: the inference rules remain the same between logics, but the axioms are allowed to vary.

For completeness, we first describe the language of propositional modal logic, which is the language of **PL** plus the unary connective $\Box$.

Definition 5.1 (Language of Propositional Modal Logic [39])

[1] there are some problems with this reading, but it works as a first approximation of why we can view it as a modal necessity operator

$$
\begin{array}{rcll}
\mathcal{A}, \mathcal{B} & ::= & p_i & \textit{Atom} \\
& | & \top & \textit{True} \\
& | & \bot & \textit{False} \\
& | & (\neg \mathcal{A}) & \textit{Not } \mathcal{A} \\
& | & (\mathcal{A} \wedge \mathcal{B}) & \mathcal{A} \textit{ and } \mathcal{B} \\
& | & (\mathcal{A} \vee \mathcal{B}) & \mathcal{A} \textit{ or } \mathcal{B} \\
& | & (\mathcal{A} \to \mathcal{B}) & \mathcal{A} \textit{ implies } \mathcal{B} \\
& | & (\Box \mathcal{A}) & \mathcal{A} \textit{ necessarily holds}
\end{array}
$$

In an axiomatic deduction system, derivations are finite sequences of formulas of modal logic. Starting from an empty list, derivations are constructed by either inserting a new formula that is an instance of an axiom, or by applying an inference rule to produce a new formula based on previous formulas in the list.

Definition 5.2 (Axioms & Inference Rules for Propositional Modal Logic [39])
The axioms include all tautologies of classical[2] propositional logic (i.e. the $\Box$-less formulas A s.t. $\vdash A$), as well as some additional modal axioms that we allow to vary depending on the system we intend to model. The inference rules remain fixed, given below.

(**MP**) Given A and $A \to B$, prove B.

(**Nec**) Given A, prove $\Box A$.

(**Sub**) Given A, prove A' where A' is the result of uniformly replacing the propositional atoms in A by arbitrary formulas. This simply allows the axioms to be used for arbitrary formulas rather than just propositional atoms.

Given a particular collection of modal axioms **T**, we say $\vdash_{\mathbf{T}} A$ iff A appears in a valid axiomatic derivation using axioms in **T**.

In an axiomatic system, we do not consider hypothetical derivations (i.e. $\Gamma \vdash A$ where Γ is non-empty) at all. This is the key distinction from natural deduction systems. We now consider our first example, the modal logic **K**. This modal logic will serve as a base logic, which is extended by all the other systems we will consider in this chapter.

Definition 5.3 (The Base Modal Logic K)
A very common axiom in the study of modal logic is axiom (**K**) $\Box(p \to q) \to \Box p \to \Box q$, which simply states that necessity is closed under modus ponens. This is an uncontroversial axiom[3]: if an agent knows $A \to B$ and A, then the agent can surely make the inference to find out that B holds. Similarly, if $\mathsf{Prov}(\ulcorner A \to B \urcorner)$ and $\mathsf{Prov}(\ulcorner A \urcorner)$, then we can simply devise the numerical operation that applies $(\to \mathrm{I})$ to the codes of the derivations. We call the classical modal logic with this axiom to be **K**, and the intuitionistic modal logic **IK**.

[2] This can be tautologies of intuitionistic logic, if the intent is to model intuitionistic modal logic.

[3] To more rigorously understand why **K** is so ubiquitous, we have to consider the possible worlds semantics of modal logic, but this is unfortunately beyond the scope of this report.

5.2 Provability Modal Logic

The exposition in this section is based on [40]. The classical propositional modal logic that arises from treating $\mathrm{Prov}(\ulcorner\cdot\urcorner)$ as necessity is called **GL**, short for "Gödel-Löb". It is also often called provability (modal) logic. The intended semantics or *provability semantics* for **GL** translates **GL**-sentences into **PA**-sentences, where each atom is interpreted as an arbitrary sentence.

Definition 5.4 (GL Provability Semantics)
Let an *arithmetic realisation* be a function mapping propositional atoms to **PA**-sentences. Given such a realisation r, we can translate arbitrary **GL**-formulas into **PA** sentences:

$$\begin{aligned}
(\!| p_i |\!)_r &= r(p_i) \\
(\!| \bot |\!)_r &= \bot \\
(\!| \neg A |\!)_r &= \neg (\!| A |\!)_r \\
(\!| A \wedge B |\!)_r &= (\!| A |\!)_r \wedge (\!| B |\!)_r \\
(\!| A \vee B |\!)_r &= (\!| A |\!)_r \vee (\!| B |\!)_r \\
(\!| A \to B |\!)_r &= (\!| A |\!)_r \to (\!| B |\!)_r \\
(\!| \Box A |\!)_r &= \mathrm{Prov}(\ulcorner (\!| A |\!)_r \urcorner)
\end{aligned}$$

We write $(\!| A |\!)$ to mean the set $\{(\!| A |\!)_r \mid r \text{ an arithmetic realisation}\}$ of **PA**-sentences that A can translate into, and $\mathbf{PA} \vdash (\!| A |\!)$ to mean that $\mathbf{PA} \vdash A'$ holds for each $A' \in (\!| A |\!)$. This is what we take as semantics: A **GL**-formula A is valid iff **PA** proves every translation of A.

$$\vDash_{\mathbf{GL}} A \quad \text{iff} \quad \mathbf{PA} \vdash (\!| A |\!)$$

This is a perfectly fine definition, but we seek to characterise **GL** by a set of axioms under the axiomatic system in order to compare and contrast it against other systems. For this purpose, we already have a good starting point with the Hilbert-Bernays-Löb conditions for **PA**, which we recall now (omitting the subscript for this section) - this time along with the associated modal axioms/inference rules.

(**Nec**)	If $\mathbf{PA} \vdash A$ then $\mathbf{PA} \vdash \mathrm{Prov}(\ulcorner A \urcorner)$.	Given A, prove $\Box A$.
(**K**)	$\mathbf{PA} \vdash \mathrm{Prov}(\ulcorner A \to B \urcorner) \to (\mathrm{Prov}(\ulcorner A \urcorner) \to \mathrm{Prov}(\ulcorner B \urcorner))$	$\Box(p \to q) \to (\Box p \to \Box q)$
(**4**)	$\mathbf{PA} \vdash \mathrm{Prov}(\ulcorner A \urcorner) \to \mathrm{Prov}(\ulcorner \mathrm{Prov}(\ulcorner A \urcorner) \urcorner)$	$\Box p \to \Box\Box p$

It is quite clear from this that (**K**) and (**4**) can serve as axioms of **GL**. However, it turns out that (**K**) and (**4**) do not completely characterise the logic with respect to the provability semantics. In other words, there are certain formulas validated by the provability semantics that cannot be derived from these axioms alone.

The missing axiom

$$(\mathbf{L})\ \Box(\Box p \to p) \to \Box p$$

was discovered by Löb, which turned out to be an internalisation of the following theorem.

Theorem 5.5 (Löb's Theorem [18])
If $\mathbf{PA} \vdash \mathrm{Prov}(\ulcorner A \urcorner) \to A$, then $\mathbf{PA} \vdash A$.

(**K**) and (**L**) together can derive (**4**), so we can omit (**4**) as an axiom for the axiomatic deduction system for **GL**. The completeness of this deduction system with respect to the intended provability semantics of **GL** was proven by Solovay.

Theorem 5.6 (Solovay's Arithmetic Completeness [41])
$$\vdash_{\{\mathbf{K},\mathbf{L}\}} A \quad \text{iff} \quad \vDash_{\mathbf{GL}} A \quad \text{iff} \quad \mathbf{PA} \vdash (\!| A |\!)$$

5.3 Modal Type Systems for Staged Metaprogramming

5.3.1 Staged Metaprogramming

Staged computation is the practice of separating an algorithm into stages, usually for performance or for clarity in expressing the algorithm. The typical example is the separation of executing a program written in a high-level programming language into two stages: the compilation of the program and the execution of the resulting machine code. While this example demonstrates staging for performance, the execution of a compiler is also separated into stages demarcating the different phases of the algorithm: lexer → parser → semantic analyser → code generator. This separation makes the underlying algorithm clearer and more modular, allowing one to swap in a different code generator targetting a different machine architecture, for example.

The example of the separation between compilation and execution leads to asking whether we can do the same staging within the programming language itself. In other words, to write programs that generate programs to be executed in the next stage. This necessarily involves programs manipulating syntax to construct other programs, i.e. metaprogramming. As a small example of *staged metaprogramming*, consider the power $k\ n$ function which computes n^k.

$$\begin{aligned} &\text{power } 0 && n = 1 \\ &\text{power } (k+1) && n = (power\ k\ n) * n \end{aligned}$$

If we are interested in a particular exponent only however, say $k = 3$, then we should just define $\text{power3} n = n * n * n$ as it is more efficient without the recursive calls. We may need a few different exponents however, and it becomes tedious to define each of them separately. This is no longer necessary if we can define a function which takes argument k and produces the code of the powerk function.

To allow this, we introduce the programming constructs known as quasiquotations $[\![\cdot]\!]$ and splices $\$(\cdot)$, originally due to Quine [42]. The understanding is that the term inside the quasiquotation is quoted, and so becomes an object representing syntax, i.e. *code*. The exception is that when we encounter a splice $\$(M)$ inside the quasiquotation, then the splice evaluates by evaluating M into a quote and substituting the quoted term into the surrounding quasiquotation. In other words, we have the β-reduction $\$([\![M]\!]) \rightsquigarrow_\beta M$. With quasiquotations and splices, we can express the function

$$\begin{aligned} &\text{superpower } 0 && = [\![\lambda n.\ 1]\!] \\ &\text{superpower } (k+1) && = [\![\lambda n.\ (\$(superpower\ k)\ n) * n]\!] \end{aligned}$$

such that $\$(\text{superpower } 3) \rightsquigarrow_\beta \lambda n.\ n * n * n$.

While this is a fairly artificial example, it serves to demonstrate how quasiquotation works. Clearly, quasiquotations require a type system to prevent ill-typed quotes and splices. To begin with, we require a type for terms that represent code. We also need to ensure that the term in a splice has this new code type, otherwise attempting to splice e.g. a natural number is ill-defined.

5.3.2 The Modal Analysis of Davies & Pfenning

Independently of developments in provability logic, Davies & Pfenning [43] performed an analysis of staged metaprogramming and arrived at the conclusion that the intuitionistic propositional modal logic **IS4** provides a good framework for assigning types to λ-calculus equipped with quasiquotations and splices. The type $\Box A$ is introduced, inhabited by codes of terms with type A.

Consider the case when M has type A, which means $[\![M]\!]$ should have the type $\Box A$. This suggests $[\![\cdot]\!]$ to be the constructor for $\Box A$. However, M might contain free variables whose type can only be inferred from the context Γ, i.e. $\Gamma \vdash M : A$. When we quote the term into code, the free variables lose their meaning as we no longer keep track of their original context. Hence, $[\![M]\!]$ is meaningful only when M is a closed term with no free variables, which suggests the introduction rule

$$\frac{\vdash M : A}{\vdash [\![M]\!] : \Box A}\ (\Box I)$$

Notice that this is very similar to the necessitation rule (**Nec**) from the modal axiomatic system, in the sense that it proves $\Box A$ from a derivation of A in the empty context[4]. Unfortunately, this is not flexible enough as we have no way of factoring in splices, which rely on access to the context outside the quote.

The solution by Pfenning & Davies is to modify the structure of the context allowing for two kinds of variables

$$\underbrace{x_1 : A_1 \ldots x_n : A_n}_{\text{code variables}}; \underbrace{y_1 : B_1 \ldots y_m : B_m}_{\text{ordinary variables}} \vdash M : C$$

The code variables are intended to stand in for splices, so they are allowed inside a quasiquotation. This use of code variables is expressed by the elimination rule for $\Box$. We have to change the structure of splices so that they be represented by code variables. Therefore, we obtain the following introduction and elimination rules for $\Box$.

Definition 5.7 (Introduction & Elimination Rule for $\Box$ [43])

$$\frac{\Delta; \cdot \vdash M : A}{\Delta; \Gamma \vdash [\![M]\!] : \Box A}\ (\Box \mathsf{I}) \qquad \frac{\Delta; \Gamma \vdash M : \Box A \qquad \Delta, u : A; \Gamma \vdash N : B}{\Delta; \Gamma \vdash \mathsf{let}\ [\![u]\!] = M\ \mathsf{in}\ N : B}\ (\Box \mathsf{E})$$

The introduction & elimination rules for the other connectives only manipulate the ordinary variables while leaving the code variables intact.

The **IS4** modal logic is characterised by axioms (**K**), (**4**) and (**T**). The first two axioms should be familiar from our investigation of **GL**. The new axiom (**T**) $\Box p \to p$ states that we can evaluate code as a regular term, i.e. splice code outside of any quote. We saw an instance of this in the superpower example, since we needed to evaluate the code of superpower k to be able to use the generated code. We now show that the system of Davies & Pfenning indeed validate these axioms.

$$*; * \vdash \lambda x.\ \lambda y.\ \mathsf{let}\ [\![u]\!] = x\ \mathsf{in}\ (\mathsf{let}\ [\![v]\!] = y\ \mathsf{in}\ [\![u\ v]\!]) : \Box(A \to B) \to (\Box A \to \Box B)$$
$$*; * \vdash \lambda x.\ \mathsf{let}\ [\![u]\!] = x\ \mathsf{in}\ [\![[\![u]\!]]\!] : \Box A \to \Box\Box A$$
$$*; * \vdash \lambda x.\ \mathsf{let}\ [\![u]\!] = x\ \mathsf{in}\ u : \Box A \to A$$

One interesting aspect of this system is that it disallows the congruence rule

$$\frac{M \rightsquigarrow_\beta M'}{[\![M]\!] \rightsquigarrow_\beta [\![M']\!]}$$

since the quotation is treated as code, the code must be sensitive to the particular syntax of the term. reduction amounts to changing the syntax, which violates this sensitivity.

[4] Recall that all axiomatic derivations are in the empty context, since there is no support for hypothetical deduction.

5.3.3 Fitch-Style Natural Deduction for Modal Logic

Davies & Pfenning's **IS4** system is developed based on particular insights about metaprogramming. As a result, it is not very clear how to extend or restrict their system to other modal logics. We present now an alternate development of natural deduction for modal logic, independent of metaprogramming. We then investigate its application towards metaprogramming.

The Gentzen-style natural deduction systems we have seen so far are based on trees. Historically, the search for modal Gentzen-style natural deduction systems have been fraught with difficulties. There is however an alternate formulation of natural deduction for classical logic (without modalities) due to Fitch [44] where derivations are instead represented as lists, as in an axiomatic system. In addition to formulas though, a Fitch-style derivation may also contain other lists/derivations. Derivations within derivations are used for hypothetical reasoning, and we call such lists *subordinate derivations* or *subderivations* for short. For example, to prove $A \to B$ using the introduction rule, we open a subordinate derivation which starts with A already assumed.

1.	A	assumption
2.	$\vdots$	
3.	B	got this somehow
4.	$A \to B$	$\to I(1-3)$

Here, lines 1 - 3 (the lines in the box) constitute the subderivation, and line 4 is obtained by applying the ($\to$ I) rule to the subderivation. This is represented by the annotations on the right of the entry. Inference rules such as the introduction and elimination rules behave as they do in axiomatic systems: they are applied to previous entries in the list to produce a new formula. The difference now is that a rule may be applied to a subordinate derivation as well.

Technically if an inference rule is applied inside a derivation δ, then it can only be applied to formulas/-subderivations occuring strictly in δ. However, we may use the import rule in a subderivation to import a copy of a formula in the outer derivation. For example,

1.	B	got this somehow
2.	A	assumption
3.	$\vdots$	
4.	B	import(1)
5.	$A \to B$	$\to$I(2 - 4)

Fitch later observed that subordinate derivations can also be used for the modal introduction rule ($\Box$I) [45]. This time however, the subderivation is not used to introduce a hypothetical assumption, but rather to restrict what formulas can be imported into the subderivation. Since the usage is different, we call it a *strict subderivation*, and label it appropriately.

1.		strict
2.	$\vdots$	
3.	B	got this somehow
4.	$\Box B$	$\Box$I(2 - 5)

The particular imports that are allowed vary depending on the modal logic we intend to model, just like the axioms in an axiomatic system. For the base modal logic **K**, only formulas of the form $\Box A$ are allowed to be imported, and the $\Box$ is removed inside the strict subderivation [46]. To model a logic with axiom (**4**), we add the choice to not drop the $\Box$ when importing [46]. Finally, modelling axiom **T** requires adding the additional inference rule that derives A from $\Box A$. We can view this as a strict import of $\Box A$ as A from

inside the same derivation. In particular, this means a strict-import-**T** can be performed at the top level, outside of any subderivation.

1.	$\Box A$	got this somehow
2.		strict
3.	A	strict-import-**K**(1)
4.	⋮	
5.	B	got this somehow
6.	$\Box B$	$\Box$I(2 - 5)

1.	$\Box A$	got this somehow
2.		strict
3.	$\Box A$	strict-import-**4**(1)
4.	⋮	
5.	B	got this somehow
6.	$\Box B$	$\Box$I(2 - 5)

1.	$\Box A$	got this somehow
2.	⋮	
3.	A	strict-import-**T**(1)
4.		

We can now attempt to bring back some metaprogramming intuition for this Fitch-style system, even if it does not keep track of λ-terms. A strict subordinate derivation corresponds to working in a quasiquotation, while strict-import-**K** seems to correspond to splicing. However, note that a **K** import can only be performed on a formula exactly one subderivation outside. We saw in the derivation of $\Box A \to \Box\Box A$ using the system of Davies & Pfenning that we need to be able to splice from outside two quasiquotes. This is the effective contribution of the **4** import rule: it allows a $\Box A$ assumption to be imported arbitrarily many times without removing the $\Box$. Similarly, the **T** import rule allows top-level splices, which is necessary for evaluating code.

A disadvantage of this Fitch-style natural deduction using lists of formulas is that there is no longer a direct correspondence between the lambda terms and the structure of Fitch-style derivations. Thankfully, we can adapt the idea of opening strict subordinate derivations to the Gentzen-style calculus, an idea originating with Borghuis [47], Martini & Masini [48]. We present a more recent and streamlined version of the theory due to Clouston [49].

There is a correspondence between regular subordinate derivations and the structure of the context in Gentzen-style natural deduction. In particular, opening a subordinate proof with hypothetical assumption A corresponds to inserting A in the context. This suggests that adapting strict subordinate derivations will also require the insertion of some structure into the context.

Unlike regular subderivations, strict subderivations do not introduce new hypothetical assumptions, but rather prevent access to formulas inferred from earlier assumptions, unless they are of a certain form. With this in mind, in the Gentzen-style system for λ-calculus, we add a lock construction 🔒 to the context whenever we enter a quasiquotation using the ($\Box$I) rule. This lock construction restricts access to any hypothetical assumptions inserted before the lock, which is expressed in the (Ass) rule. Only terms that are being spliced can access locked assumptions, which is expressed by lock removal when we enter a splice, as expressed by the $\Box$ elimination rules. We present an elimination rule that models **K**, and another rule modelling **S4** - both based on the strict import rules for the Fitch-style deduction system. We call these new systems the $\lambda^{\mathbf{K}}$-calculus and $\lambda^{\mathbf{S4}}$-calculus, respectively. They correspond to the logics **IK** and **IS4**.

Definition 5.8 (Typing rules For $\Box$ In $\lambda^{\mathbf{K}}$ [49] and $\lambda^{\mathbf{S4}}$ [50])
The introduction and assumption rules are shared by both $\lambda^{\mathbf{K}}$ and $\lambda^{\mathbf{S4}}$.

$$\frac{\Gamma, 🔒 \vdash M : A}{\Gamma \vdash [\![M]\!] : \Box A}\ (\Box\text{I}) \qquad \frac{\Gamma = \Gamma_1, x : A, \Gamma_2 \qquad 🔒 \notin \Gamma_2}{\Gamma \vdash x : A}\ (\text{Ass})$$

Splices in $\lambda^{\mathbf{K}}$ may only access variables that are locked exactly once. In particular, this also means it cannot access variables that have not been locked.

$$\frac{\Gamma_1 \vdash M : \Box A \qquad 🔒 \notin \Gamma_2}{\Gamma_1, 🔒, \Gamma_2 \vdash \$(M) : A}\ (\Box\text{E-}\mathbf{K})$$

Splices in $\lambda^{\mathbf{S4}}$ are allowed to access all variables in the context, reflecting the strength of the **K**,

4 and **T** strict import rules. This is expressed by the removal of all locks in Γ, denoted by $\Gamma^{🔓}$. In particularly, splices are allowed even when Γ contains no locks.

$$\frac{\Gamma^{🔓} \vdash M : \Box A}{\Gamma \vdash \$(M) : A}\ (\Box\text{E-}\mathbf{S4})$$

The remaining rules for the other connectives remain exactly the same as for the simply typed lambda calculus.

As with Davies & Pfenning's system, we can show that the modal axioms are indeed validated. Note that the derivation for axiom (**K**) is applicable to both $\lambda^{\mathbf{K}}$ and $\lambda^{\mathbf{S4}}$.

$$\dfrac{\dfrac{\dfrac{}{x : \Box A \vdash x : \Box A}}{x : \Box A \vdash \$(x) : A}}{\vdash \lambda x.\ \$(x) : \Box A \to A} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : \Box A, \vdash x : \Box A}}{x : \Box A, 🔒, 🔒 \vdash \$(x) : A}}{x : \Box A, 🔒 \vdash [\![\$(x)]\!] : \Box A}}{x : \Box A \vdash [\![[\![\$(x)]\!]]\!] : \Box\Box A}}{\vdash \lambda x.\ [\![[\![\$(x)]\!]]\!] : \Box A \to \Box\Box A}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \Box(A \to B), y : \Box A \vdash x : \Box A \to B}}{x : \Box(A \to B), y : \Box A, 🔒 \vdash \$(x) : A \to B} \quad \dfrac{\dfrac{}{x : \Box(A \to B), y : \Box A \vdash y : \Box A}}{x : \Box(A \to B), y : \Box A, 🔒 \vdash \$(y) : A}}{x : \Box(A \to B), y : \Box A, 🔒 \vdash \$(x)\ \$(y) : B}}{\dfrac{\dfrac{x : \Box(A \to B), y : \Box A \vdash [\![\$(x)\ \$(y)]\!] : \Box B}{x : \Box(A \to B) \vdash \lambda y.\ [\![\$(x)\ \$(y)]\!] : \Box A \to \Box B}}{\vdash \lambda x.\ \lambda y.\ [\![\$(x)\ \$(y)]\!] : \Box(A \to B) \to (\Box A \to \Box B)}}$$

Unlike Davies & Pfenning's system which was built from the ground up for metaprogramming, $\lambda^{\mathbf{K}}$ and $\lambda^{\mathbf{S4}}$ are derived from a general deduction system for modal logic. Unfortunately, this means that the approach does not have much to say about reduction rules. We still expect the usual β-reduction rules for the other connectives as long as they occur outside of a $[\![\ ]\!]$, as well as the following β rule:

$$\frac{}{\$([\![M]\!]) \rightsquigarrow_\beta M}$$

but it is not clear what sort of congruence rule $[\![\cdot]\!]$ should have. Following Davies & Pfenning by eliminating the congruence rule for $[\![\cdot]\!]$ is only sensible for $\lambda^{\mathbf{S4}}$, where splices can occur outside of any quotes. In $\lambda^{\mathbf{K}}$, all splices occur inside a quote so the removal of the congruence rule simply means no β-reduction for splices/quotes can occur at all, which does not seem correct. On the other hand, having a general congruence rule violates the idea that a term under $[\![\cdot]\!]$ is code, and therefore sensitive to syntactic manipulations. A good middle-ground seems to be to allow only the parts of a term occuring immediately under a splice to reduce. In any case, we postpone the discussion of appropriate reduction rules for $\lambda^{\mathbf{K}}$ to the next chapter, where we can use the provability semantics to settle this ambiguity.

Regardless of this ambiguity with reduction rules, $\lambda^{\mathbf{K}}$ and $\lambda^{\mathbf{S4}}$ are still conceptually simpler to reason about than the Davies & Pfenning system, since the splicing operation is less verbose and requires less manipulations in the context.

5.4 The Incompatibility Between Provability and Metaprogramming

Having identified and examined the modal nature of both provability and metaprogramming, we may now consider the possibility of interpreting the type $\Box A$ more rigorously in terms of provability. The

type $\Box A$ necessarily contains two components: the code of some term, and a witness that the term is indeed of type A. At first thought, this is highly reminiscent of the provability type $\mathsf{Prov}_{\mathbf{MLTT}}(\ulcorner A \urcorner) \equiv \sum_{xy:\mathbb{N}} \overline{\mathsf{prf}}(x, y, \ulcorner A \urcorner)$, because an inhabitant of this type has the form $\langle M, N \rangle$ where $M : \mathbb{N}$ codes a term and $N : \sum_{y:\mathbb{N}} \overline{\mathsf{prf}}(M, y, \ulcorner A \urcorner)$ witnesses that the coded term indeed has type A. This suggests that we can interpret $\Box A$ as $\mathsf{Prov}_{\mathbf{MLTT}}(\ulcorner A \urcorner)$. Notice that the constructive nature of **MLTT** is essential here - a classical proof of an existential sentence does not constitute a pair.

On closer inspection however, there appears to be some wrinkles to this idea: the modal logics of provability **GL** and metaprogramming **S4** are different. While they share the common axioms (**K**) and (**4**), **GL** has axiom (**L**) and **S4** has axiom (T). Attempting to combine the two logics lead to severe issues with consistency. Working purely in the axiomatic deduction system for modal logic, we can easily derive an inconsistency from the combined use of axioms (**L**) and (**T**):

1.	$\Box(\Box p \to p) \to \Box p$	Axiom (**L**)
2.	$\Box p \to p$	Axiom (**T**)
3.	$\Box(\Box p \to p)$	Apply rule (**Nec**) to 2
4.	$\Box p$	Apply rule (**MP**) to 1 and 3
5.	p	Apply rule (**MP**) to 2 and 4
6.	$\bot$	Apply rule (**Sub**) to 5, substituting $[\bot / p]$

We obtain a less opaque explanation of the inconsistency [51] by considering that axiom (**T**) allows a derivation of $\Box\ \bot \to \bot$, which is equivalent to $\neg(\Box\ \bot)$. However, applying the proposed provability interpretation to obtain $\neg(\mathsf{Prov}_{\mathbf{MLTT}}(\ulcorner \bot \urcorner))$, it is clear that this is the internalised consistency statement of **MLTT**, which by the second incompleteness theorem cannot be proven unless **MLTT** is inconsistent. Hence, the provability interpretation cannot validate axiom (**T**).

A more computational explanation of the inconsistency can be given by instead attempting to integrate axiom (**L**) into $\lambda^{\mathbf{S4}}$ with the following rule, adapted from [52]:

$$\frac{\Gamma, 🔒, z : \Box A \vdash M : A}{\Gamma \vdash \mathsf{fix}\ z\ \mathsf{in}\ M : A}$$

Denoting the term for axiom (**T**) that we derived earlier as $\mathsf{eval} : \Box A \to A$, we may now derive the general fixpoint combinator

$$\Gamma \vdash \mathsf{fix}\ z\ \mathsf{in}\ (\lambda x.\ \mathsf{eval}\ z\ x) : (A \to A) \to A$$

which makes the system inconsistent as it allows a term of any type A to be derived by simply applying the above term to $\lambda x.\ x$.

All together, these explanations suggest that we must be conservative in attempting to interpret staged metaprogramming as provability. Because we cannot interpret the full **S4** modality as provability, we have to restrict it to just (**K**) and (**4**). This is a huge blow as it means provability cannot justify evaluation of code, which is crucial in the practice of metaprogramming for actually using the metaprograms, as we saw with the superpower example.

6 ⊢ The Provability Semantics of Metaprogramming in Martin-Löf's Type Theory

> Deep in the human unconscious is a pervasive need for a logical universe that makes sense, But the real universe is always one step beyond logic.
>
> –from *The Sayings of Muad'Dib* by the Princess Irulan

–from *Dune* by Frank Herbert

In the previous chapter, we investigated provability and metaprogramming under the unifying framework of modal logic, discovering that while the modal logics **GL** and **S4** (corresponding to provability and metaprogramming respectively) share some common features, they are ultimately incompatible.

In this chapter, we push forward anyway by restricting to the modal logic **K** which is a shared sublogic of both **GL** and **S4**. For **K**, instead of using the lock-based system $\lambda^{\mathbf{K}}$, we can consider a more specialised theory which annotates terms by their staging level [53]. Using these level annotations, we no longer need to insert locks in the context, so the structure of the context remains the same as the regular simply-typed λ-calculus. We then proceed to adapt this idea of level annotations to **MLTT**, obtaining the theory $\mathbf{MLTT}^{\mathbf{lvl}}$.

Next, we sketch a provability semantics which translates derivations of $\mathbf{MLTT}^{\mathbf{lvl}}$ into derivations of **MLTT**. Derivations of the type $\Box A$ are translated into derivations of $\mathrm{Prov}_{\mathbf{MLTT}}\ \ulcorner A \urcorner$. As a corollary of the provability semantics, we find that $\mathbf{MLTT}^{\mathbf{lvl}}$ is consistent, since **MLTT** is consistent as well [4]. This serves as a first check that $\mathbf{MLTT}^{\mathbf{K}}$ is a viable logical theory.

With this semantics in mind, we proceed to discuss the computation rules of $\mathbf{MLTT}^{\mathbf{lvl}}$ that will be sound under the semantics. We prove the type soundness of these definitional equality rules for $\mathbf{MLTT}^{\mathbf{K}}$, as a sanity check that $\mathbf{MLTT}^{\mathbf{K}}$ is also a viable typed programming language.

6.1 Staging Levels

Since we are interested in only the **K** modality, we may consider a simpler and more uniform representation of $\lambda^{\mathbf{K}}$ which avoids the use of locks. This representation is based on staging levels, which identifies at what stage a particular term is defined.

We first introduce the idea for simple types, and then provide a straightforward extension to dependent types in Martin-Löf's type theory.

6.1.1 Simple Types

Suppose we have a derivation of $\Gamma \vdash M : A$ in $\lambda^{\mathbf{K}}$, where Γ contains n locks. Then we recognize that M is occuring under $n + m$ quotes and m splices, since each quote adds a lock and each splice removes a lock. We identify this as the *staging level* of M [53], which is the number of quotes minus the number of splices surrounding M.

Consider now the term $[\![\$(\$(M))]\!]$ at level n. The quote clearly owns the outer splice, since the term inside the outer splice occurs at level n as well. However, the term in the inner splice occurs at level $n-1$, so it does not belong to this quote, but rather to some outer quote (not shown). This shows that quotes at level n capture all and only splices of terms at level n.

One way to make explicit this idea of staging level is to annotate the typing judgement with a level [53], instead of using locks. We call this theory with level annotations $\lambda^{\mathbf{lvl}}$. To identify which level a variable is bound at, we also annotate context variables by their level, leading to a typing judgement of the form:

$$x_1 : (A_1, n_1), \ldots x_k : (A_k, n_k) \vdash^n M : A$$

Here, the derivation as a whole occurs at level n. In the context, variable x_i has type A_i and has staging level n_i. This means x_i was bound by an abstraction at level n_i, and can only occur at level n_i inside M. These are expressed by the following rules

Definition 6.1 (Context-related Rules For $\lambda^{\mathbf{lvl}}$)

$$\frac{\Gamma, x : (A, n) \vdash^n M : B}{\Gamma \vdash^n \lambda x.\, M : A \to B}\ (\to \mathsf{I}) \qquad \frac{x : (A, n) \in \Gamma}{\Gamma \vdash^n x : A}\ (\mathsf{Ass})$$

Notice that we no longer have a need for locks, since the level annotation is now being used to restrict access to variables at a given level. The introduction and elimination rules for $\Box$ shift the levels up and down, rather than introducing and removing locks.

Definition 6.2 (Introduction & Elimination Rules For $\Box$ in $\lambda^{\mathbf{lvl}}$)

$$\frac{\Gamma \vdash^{n+1} M : A}{\Gamma \vdash^n [\![M]\!] : \Box A}\ (\Box\mathsf{I}) \qquad \frac{\Gamma \vdash^n M : \Box A}{\Gamma \vdash^{n+1} \$(M) : A}\ (\Box\mathsf{E})$$

In the context of metaprogramming, a term's level denotes the stage at which it is to be evaluated. A term at level 0 is evaluated at runtime, i.e. the current stage, while a term at positive levels are to be evaluated at a future stage. Negative levels are also commonly included, which denote terms to be evaluated at compile time [54]. However, having negative levels allow splices to occur at level 0 outside of any quotations, akin to in $\lambda^{\mathbf{S4}}$, which means negative levels are not valid under a provability interpretation. For this reason, we will consider only non-negative levels.

In addition to keeping explicit track of levels, $\lambda^{\mathbf{lvl}}$ more closely resembles the regular simply typed λ calculus as we do not have to worry about locks in the context. Both of these features make it easier to discuss the provability semantics of a level-annotated system than in a lock-based system. Of course, we want the semantics to apply to the lock-based system as well, so we need to establish that $\lambda^{\mathbf{lvl}}$ can prove anything $\lambda^{\mathbf{K}}$ can.

It is fairly straightforward to show that any typeable term in $\lambda^{\mathbf{K}}$ is also typeable in $\lambda^{\mathbf{lvl}}$. Since the syntax of types and terms remain the same, we only need to define a translation of contexts.

Definition 6.3 (Translation of $\lambda^{\mathbf{K}}$ Contexts Into $\lambda^{\mathbf{lvl}}$ Contexts)
We first define the translation indexed by a level $k \in \mathbb{N}$:

$$(\!|\, \Gamma, \text{🔒} \,|\!)_{k+1} \triangleq (\!|\, \Gamma \,|\!)_k$$
$$(\!|\, \Gamma, x : A \,|\!)_k \triangleq (\!|\, \Gamma \,|\!)_k, x : (A, k)$$

Denoting the number of locks in Γ by $||\Gamma||$, one can naturally see that $(\!|\, \Gamma \,|\!)_k$ is well-defined only when $k \geq ||\Gamma||$. Hence, a natural choice is to define the translation as

$$(\!|\, \Gamma \,|\!) \triangleq (\!|\, \Gamma \,|\!)_{||\Gamma||}$$

With this translation, we can establish that for any derivation in $\lambda^{\mathbf{K}}$, we can translate the context to obtain a derivation in $\lambda^{\mathbf{lvl}}$, and vice versa. We prove a slightly more general result amenable to a proof by induction.

Theorem 6.4

1. For all $k \in \mathbb{N}$, if $\Gamma \vdash_{\lambda\mathsf{K}} M : A$ then $(\!|\, \Gamma \,|\!)_{||\Gamma||+k} \vdash^{||\Gamma||+k}_{\lambda^{\mathbf{lvl}}} M : A$.
2. If $\Gamma \vdash_{\lambda\mathsf{K}} M : A$ then $(\!|\, \Gamma \,|\!) \vdash^{||\Gamma||}_{\lambda^{\mathbf{lvl}}} M : A$.

Proof.

1. By induction on M, generalising k, Γ and A. See Appendix 2.1 for the proof.
2. This is just the special case of 1. when $k = 0$. ⊣

Defining a translation in the opposite direction from $\lambda^{\mathbf{lvl}}$ to $\lambda^{\mathbf{K}}$ is more difficult as there's no guarantee that the context will be increasingly ordered by level. However, this is not an issue as our translation from $\lambda^{\mathbf{K}}$ to $\lambda^{\mathbf{lvl}}$ already ensures that any provability semantics for $\lambda^{\mathbf{lvl}}$ applies to both theories.

6.1.2 Dependent Types

We can fairly easily extend the level annotations to Martin-Löf's type theory, obtaining **MLTT**$^{\mathbf{lvl}}$. With dependent types, we must now also ensure that types in the context are well-formed at the level they are annotated at, which is expressed via the well-formed context judgement.

Definition 6.5 (Context-related Rules For MLTT$^{\mathbf{lvl}}$)

$$\frac{\Gamma, x : (A, n) \vdash^n M : B}{\Gamma \vdash^n \lambda x.\, M : \prod_{x:A} B}\ (\Pi\mathsf{I}) \qquad \frac{x : (A, n) \in \Gamma}{\Gamma \vdash^n x : A}\ (\mathsf{Ass}) \qquad \frac{\Gamma \vdash^n A : \mathcal{U}_i}{\Gamma, x : (A, n)\ \mathrm{ctx}}\ (\text{ctx-var})$$

The introduction and elimination rules for $\Box$ remain exactly the same as in $\lambda^{\mathbf{lvl}}$. However, we now also have to consider the formation rule, which also shifts the level up by one in accordance with the quote.

Definition 6.6 (Formation, Introduction & Elimination Rules For $\Box$ in MLTT$^{\mathbf{lvl}}$)

$$\frac{\Gamma \vdash^{n+1} A : \mathcal{U}_i}{\Gamma \vdash^n \Box A : \mathcal{U}_i}\ (\Box\mathsf{F}) \qquad \frac{\Gamma \vdash^{n+1} M : A}{\Gamma \vdash^n [\![M]\!] : \Box A}\ (\Box\mathsf{I}) \qquad \frac{\Gamma \vdash^n M : \Box A}{\Gamma \vdash^{n+1} \$(M) : A}\ (\Box\mathsf{E})$$

The remaining rules for the other connectives of **MLTT** remain the same, except that they will have to propagate the level unchanged.

We can prove the following standard lemmas for **MLTT**$^{\mathbf{lvl}}$. Due to the mutual recursion between the typing judgement, well-formed context judgement and definitional equality judgement, the proof relies on the definitional equality rules, which are given only in Subsection 6.3.1 after we discuss the provability semantics.

Lemma 6.7

1. If $\Gamma \vdash^n M : A$ then Γ ctx.
2. If $\Gamma \vdash^n M : A$ then $\Gamma \vdash^n A : \mathcal{U}_i$ for some universe level i.
3. (Weakening) If $\Gamma, \Delta \vdash^n M : B$ and $\Gamma \vdash^m A : \mathcal{U}_i$, then for any fresh variable x not occuring in Γ nor Δ, $\Gamma, x : (A, m), \Delta \vdash^n M : B$.
4. (Substitution) If $\Gamma, x : (A, m), \Delta \vdash^n M : B$ and $\Gamma \vdash^m N : A$ then $\Gamma, \Delta[N/x] \vdash^n M[N/x] : B[N/x]$.
5. (Exchange) If $\Gamma, x : (A, m), y : (B, n), \Delta \vdash^k M : C$ and $\Gamma \vdash^n B : \mathcal{U}_i$, then $\Gamma, y : (B, n), x : (A, m), \Delta \vdash^k M : C$.

Proof. All the proofs are done by mutual induction. See Appendix 2.2. ⊣

6.2 Provability Semantics

As explained in the previous chapter, the reduction rules for $\lambda^{\mathbf{K}}$, or in this case the definitional equality rules for **MLTT**$^{\mathbf{lvl}}$, are ambiguous. We can settle this ambiguity by considering what rules would be sound with respect to the provability semantics. Independent of this, the provability semantics serves to justify and explain the meaning of quotes and splices. It also reduces the consistency of **MLTT**$^{\mathbf{lvl}}$ to **MLTT**, which has already been proven by Martin-Löf [55].

6.2.1 A First Attempt

Recall from the previous chapter that the core idea behind the provability semantics is to translate the term $[\![M]\!] : \Box A$ into an **MLTT** term of type $\sum_{xy:\mathbb{N}}(\overline{\mathsf{prf}}\ x\ y\ \ulcorner A \urcorner)$. A canonical term of such type will be a triple[1] $\langle M_1, M_2, M_3 \rangle$ where M_1 codes a *closed* term of type A, M_2 codes a derivation of this term, and M_3 witnesses that M_2 is indeed such a derivation. While it is clear that M_1 should code the result of recursively translating M into **MLTT**, M_2 has to code an **MLTT** derivation of M_1, which we do not have as we are only tanslating terms.

Due to this, we should rather consider a translation of derivations in **MLTT**$^{\mathbf{K}}$ into derivations in **MLTT**, proceeding recursively. Given a ($\Box$I) derivation (left), we first recursively translate δ to obtain δ' (right).

$$\dfrac{\begin{array}{c}\vdots\,\delta\\ \Gamma \vdash^{n+1} M : A\end{array}}{\Gamma \vdash^n [\![M]\!] : \Box A}\ (\Box\mathrm{I}) \quad \Longrightarrow \quad \begin{array}{c}\vdots\,(\!|\,\delta\,|\!)\\ \Gamma' \vdash M' : A'\end{array}$$

The code of $(\!|\,\delta\,|\!)$ can then be used in a derivation of $\sum_{xy:\mathbb{N}}(\overline{\mathsf{prf}}\ x\ y\ \ulcorner A \urcorner)$.

$$\dfrac{\begin{array}{c}\vdots\,\delta_1\\ \Gamma \vdash \ulcorner M' \urcorner : \mathbb{N}\end{array} \qquad \begin{array}{c}\vdots\,\delta_2\\ \Gamma \vdash \ulcorner (\!|\,\delta\,|\!) \urcorner : \mathbb{N}\end{array} \qquad \dfrac{???}{\Gamma \vdash ??? : \overline{\mathsf{prf}}\ \ulcorner M' \urcorner\ \ulcorner (\!|\,\delta\,|\!) \urcorner\ \ulcorner A \urcorner}}{\Gamma \vdash \langle \ulcorner M' \urcorner, \ulcorner (\!|\,\delta\,|\!) \urcorner, ??? \rangle : \sum_{xy:\mathbb{N}}(\overline{\mathsf{prf}}\ x\ y\ \ulcorner A \urcorner)}$$

[1] Or to be pedantic, a pair whose right element is also a pair.

The ???s are unknown because they depend on the particular definition of $\overline{\text{prf}}$. The construction of prf is likely to take significant time, so we will leave it unknown in this report. On the other hand, δ_1 and δ_2 should be easily obtainable as $\ulcorner M' \urcorner$ and $\ulcorner (\!| \delta |\!) \urcorner$ are just s repeatedly applied to z as they are natural numbers in canonical form.

This is quite a naive attempt at establishing a provability semantics for it has not accounted for splices at all. In particular, M may contain splices and it is not immediately clear what we should do with them when translating derivations, as splices represent code that is as of yet unknown.

6.2.2 Splice Environments

In order to deal with splices, we take inspiration from the splice environments of [54]. Splice environments provide yet another way of representing metaprograms. Rather than having splices inside a quote, splices are instead represented by a splice variable. The corresponding quote is then annotated with a splice environment, which keeps track of the splice variables inside the quote and the code terms to be spliced in place of each splice variable. As an example, the term on the left with regular quotes and splices can be represented using splice environments by the term on the right.

$$[\![M_1\ \$(M_2)\ [\![\$(M_3)]\!]]\!] \qquad \text{becomes} \qquad [\![M_1\ s'\ [\![s]\!]_{s \mapsto M_3}]\!]_{s' \mapsto M_2}$$

In [54], splice environments provide the advantage that terms inside quotes can be treated opaquely, since all the splices have been extracted. With splices, this is not possible as we must be able to inspect inside a quote in order to evaluate splices. The opacity provides greater freedom in designing the internal representation of code.

For the provability semantics, we can take a similar approach to translating splices by replacing them with splice variables. When we eventually get to the quote $[\![M]\!]$, we can translate the derivation δ of M to obtain δ' and M'. As before, we may take the code of M' and δ', but of course, M' is no longer a closed term because it now contains the splice variables. Hence, the code terms we extracted from each splice have to be substituted into the code of M'. Similarly, the derivation from each extracted code term has to be substituted into δ'.

In [54], an elaboration procedure translates terms in a level-annotated theory[2] with quotes and splices, to terms in a theory with quotes annotated by splice environments. The elaboration procedure returns both a term and a splice environment, which keeps track of splices that have not been captured by a quote yet. We do the same, except that our procedure works at the level of derivations not terms. First, we set up our own definition of splice environments. While splice environments in [54] map splices to terms, we have to also keep track of their derivations, since our elaboration procedure operates on terms.

Definition 6.8 (Splice Environments)
A splice environment is defined as a list of splice definitions. A splice definition consists of a *splice variable* s, its expected **MLTT** type A, the **MLTT** term M that is meant to be spliced in, and an **MLTT** derivation δ of M. It is also annotated by a level n, denoting the level at which the splice occured. We denote splice environments as ϕ.

$$\begin{array}{rcll} \phi & ::= & * & \textit{Empty Environment} \\ & | & \phi, s : A \overset{n}{\mapsto} M; \delta & \textit{Splice Definition} \end{array}$$

Given a splice environment $\phi, s : A \overset{n}{\mapsto} M; \delta$, M is intended to be a term of type $\sum_{xy:\mathbb{N}} (\overline{\text{prf}}\ x\ y\ \ulcorner A \urcorner_\phi)$. Here, $\ulcorner A \urcorner_\phi$ denotes the term where the previous splices in ϕ are substituted into the code of A. The substitution is necessary since A is a dependent type so it may itself contain other splice variables.

[2] Specifically, System F.

Definition 6.9 (Splice Substitutions)
If $\phi = s_1 : A_1 \overset{n}{\mapsto} M_1; \delta_1, \ldots, s_k : A_k \overset{n}{\mapsto} M_k; \delta_k$, then $\ulcorner M \urcorner_\phi$ denotes the splice substituted term

$$\text{subst-term}^* \; \pi_1(M_1) \; \ulcorner s_1 \urcorner \; (\ldots \; (\text{subst-term}^* \; \pi_1(M_k) \; \ulcorner s_k \urcorner \; \ulcorner M \urcorner) \; \ldots)$$

Here, subst-term(x, y, z) is the primitive recursive function that substitutes the term coded by x for occurrences of the variable coded by y in the term coded by z. Recall that primitive recursive functions may be more naturally represented by a function from $\mathbb{N}$ to $\mathbb{N}$ in **MLTT**, rather than a relation. Additionally, we take the left projection of M_1 with the expectation that it codes the term to be spliced in.

We may define the same operation but for codes of derivations instead,

$$\ulcorner \delta \urcorner_\phi \equiv \text{subst-deriv}^* \; \pi_1(\pi_2(M_1)) \; \ulcorner s_1 \urcorner \; (\ldots \; (\text{subst-deriv}^* \; \pi_1(\pi_2(M_k)) \; \ulcorner s_k \urcorner \; \ulcorner \delta \urcorner) \; \ldots)$$

where subst-deriv(x, y, z) is the primitive recursive function that substitutes the derivation coded by x for occurrences of the variable coded by y in the derivation coded by z.

We only assume the existence of subst-term and subst-deriv but it is easy to intuitively see why they are primitive recursive, since substitution amounts to traversing a finitely-sized term/derivation and replacing subparts of it.

6.2.3 The Elaboration Procedure

We may now describe the elaboration procedure, which takes place recursively on derivations δ of $\mathbf{MLTT}^{\mathbf{lvl}}$, returning an **MLTT** derivation $(\!|\, \delta \,|\!)$ and a splice environment ϕ. We denote this as

$$\delta \Mapsto (\!|\, \delta \,|\!) \mid \phi$$

Due to time constraints, we provide only a sketch of the elaboration for the $\Box$ formation, introduction and elimination rules. In particular, we will ignore the construction of Γ ctx derivations when defining the elaboration procedure. Some additional work will be needed to flesh out the details of the provability semantics.

Definition 6.10 (Elaboration of $\Box$E)
Given a derivation ending with $\Box$E, we first recursively elaborate its subderivation δ.

$$\begin{array}{c} \vdots\,\delta \\ \Gamma \vdash^n M : \Box A \end{array} \quad \Mapsto \quad \begin{array}{c} \vdots\,(\!|\, \delta \,|\!) \\ \Gamma' \vdash M' : \sum_{x\,y:\mathbb{N}} (\overline{\text{prf}}\; x\; y\; \ulcorner A' \urcorner_{\phi_1}) \end{array} \;\Big|\; \phi_2$$

Γ' contains all the variables originally in Γ along with any newly added splice variables during the elaboration of δ. These splice variables in Γ' are the same as those in ϕ_2.

Rather than directly building on $(\!|\, \delta \,|\!)$, we elaborate the ($\Box$E) derivation to a derivation ending with the assumption (Ass) rule, applied to a fresh splice variable s that has not been used before. This effectively replaces the splice with a splice variable which acts as a placeholder. The splice is then added to the splice environment ϕ_2, stored there until we encounter the quote corresponding to this splice.

$$\dfrac{\begin{array}{c}\vdots\,\delta\\ \Gamma \vdash^{n} M : \Box A\end{array}}{\Gamma \vdash^{n+1} \$(M) : A}\ (\Box\mathsf{E}) \quad \Rightarrow\!\!\!\!\Rightarrow \quad \dfrac{\Gamma', \phi_1^{\Gamma}, s : A' \text{ ctx}}{\Gamma', \phi_1^{\Gamma}, s : A' \vdash s : A'}\ (\mathsf{Ass}) \quad \Big| \quad \phi_2, \phi_1, s : A' \overset{n}{\mapsto} M'; (\!|\, \delta \,|\!)$$

ϕ_1^{Γ} denotes the append ϕ_1 into a context. We have to add back the ϕ_1 splice variables so that A' will be well-formed under the given context.

While the elaboration of a splice inserts new splice variables into the splice environment, the elaboration of a quote removes splices in order to substitute them into the derivation. The elaboration of (□I) remains mostly the same as the naive first attempt, except that we now have to substitute in the splices captureed by the quote.

Definition 6.11 (Elaboration of (□I))
Given a derivation ending with □E, first recursively elaborate its subderivation δ.

$$\begin{array}{c}\vdots\,\delta\\ \Gamma \vdash^{n+1} M : A\end{array} \quad \Rightarrow\!\!\!\!\Rightarrow \quad \begin{array}{c}\vdots\,(\!|\, \delta \,|\!)\\ \Gamma' \vdash M' : A'\end{array} \quad \Big| \quad \phi$$

Then, the (□I) derivation is elaborated into a derivation of Prov $\ulcorner A' \urcorner$, using the code of $(\!|\, \delta \,|\!)$ and M' to construct the term.

$$\dfrac{\begin{array}{c}\vdots\,\delta\\ \Gamma \vdash^{n+1} M : A\end{array}}{\Gamma \vdash^{n} [\![M]\!] : \Box A}\ (\Box\mathsf{I}) \quad \Rightarrow\!\!\!\!\Rightarrow \quad \dfrac{\delta_1 \quad \delta_2 \quad \dfrac{???}{\Gamma' - \phi.n \vdash ??? : \overline{\mathsf{prf}}\ \ulcorner M' \urcorner_{\phi.n}\ \ulcorner (\!|\, \delta \,|\!) \urcorner_{\phi.n}\ \ulcorner A' \urcorner_{\phi.n}}}{\Gamma' - \phi.n \vdash \langle \ulcorner M' \urcorner_{\phi.n}, \ulcorner (\!|\, \delta \,|\!) \urcorner_{\phi.n}, ??? \rangle : \sum_{xy:\mathbb{N}} (\overline{\mathsf{prf}}\ x\ y\ \ulcorner A' \urcorner_{\phi.n})} \quad \Big| \quad \lfloor \phi \rfloor_n$$

In the elaboration, $\phi.n$ denotes the splice environment containing all & only level n splice variables in ϕ while $\lfloor \phi \rfloor_n$ denotes the splice environment containing all splices variables in ϕ except at level n. Essentially, we remove the level n splices and substitute them into the codes of M', $(\!|\, \delta \,|\!)$ and A', leaving the remaining untouched.

The derivations δ_1 and δ_2, given below due to space constraints, can be derived from the derivations of the terms being spliced in, obtained from $\phi.n$.

$$\begin{array}{c}\vdots\,\delta_1\\ \Gamma' - \phi.n \vdash \ulcorner M' \urcorner_{\phi.n} : \mathbb{N}\end{array} \qquad \begin{array}{c}\vdots\,\delta_2\\ \Gamma' - \phi.n \vdash \ulcorner (\!|\, \delta \,|\!) \urcorner_{\phi.n} : \mathbb{N}\end{array}$$

Finally, the elaboration of (□F) is similar to (□I) in that it takes the code of the type and substitutes in the captured splices.

Definition 6.12 (Elaboration of (□F))
As usual, first we elaborate the subderivation.

$$\begin{array}{c}\vdots\,\delta\\ \Gamma \vdash^{n+1} A : \mathcal{U}_i\end{array} \quad \Rightarrow\!\!\!\!\Rightarrow \quad \begin{array}{c}\vdots\,(\!|\, \delta \,|\!)\\ \Gamma' \vdash A' : \mathcal{U}_i\end{array} \quad \Big| \quad \phi$$

Then, the elaboration only uses the code of A' and not $(\!|\, \delta \,|\!)$, substituting in the captured splices as in (□I).

$$\dfrac{\begin{array}{c}\vdots\,\delta\\ \Gamma \vdash^{n+1} A : \mathcal{U}_i\end{array}}{\Gamma \vdash^{n} \Box A : \mathcal{U}_i}\,(\Box\mathsf{I}) \quad \mapsto \quad \dfrac{\dfrac{???}{\Gamma' - \phi.n, x : \mathbb{N}, y : \mathbb{N} \vdash \overline{\mathsf{prf}}\ x\ y\ \ulcorner A' \urcorner_{\phi.n} : \mathcal{U}_i}}{\Gamma' - \phi.n \vdash \sum_{xy:\mathbb{N}} (\overline{\mathsf{prf}}\ x\ y\ \ulcorner A' \urcorner_{\phi.n}) : \mathcal{U}_i} \;\Big|\; \lfloor \phi \rfloor_n$$

We omit the two derivations that show $\mathbb{N}$ is a well-formed type (arising from the type of x and y) since they are simply applications of $(\mathbb{N})$F.

The rules for the remaining connectives are elaborated in the obvious sense by re-applying the same rule and propagating the splice environments. For example, the elaboration of (ΠE) is

$$\dfrac{\begin{array}{c}\vdots\,\delta_1\\ \Gamma \vdash^{n} M : \prod_{x:A} B\end{array} \qquad \begin{array}{c}\vdots\,\delta_2\\ \Gamma \vdash^{n} N : A\end{array}}{\Gamma \vdash^{n} M\ N : B[N/x]}\,(\Pi\mathsf{E}) \quad \mapsto \quad \dfrac{\begin{array}{c}\vdots\,(\!|\, \delta_1 \,|\!)\\ \Gamma' \vdash M' : \prod_{x:A'} B'\end{array} \qquad \begin{array}{c}\vdots\,(\!|\, \delta_2 \,|\!)\\ \Gamma'' \vdash N' : A'\end{array}}{\Gamma' \cup \Gamma'' \vdash M'\ N' : B'[N'/x]}\,(\Pi\mathsf{E}) \;\Big|\; \phi_1, \phi_2$$

where $\Gamma' \cup \Gamma''$ denotes the union of the two contexts. This is necessary so that the splice variables from both derivations are combined. ϕ_1 and ϕ_2 are the splice environments obtained from elaborating δ_1 and ϕ_2 respectively.

Since this is just a sketch, it remains to be seen whether every elaboration produces a valid derivation in **MLTT**. If it did, we can establish the consistency of **MLTT**$^{\mathbf{lvl}}$ on the consistency of **MLTT**, since the consistency of **MLTT** means that there is no valid derivation of $* \vdash M : \mathbb{0}$.

6.2.4 A Simple Example

As a simple example of the elaboration, consider the function numeral below which computes the code of the canonical form of a natural number. It is the internalised version of the $\overline{n}$ operation which converts a natural number n in the metatheory into the term with s applied n times to z.

$$\begin{aligned}&\mathsf{numeral} : \mathbb{N} \to \Box\mathbb{N}\\ &\mathsf{numeral}\ \mathsf{z} \quad :\equiv [\![\mathsf{z}]\!]\\ &\mathsf{numeral}\ \mathsf{s}(x) :\equiv [\![\mathsf{s}(\$(\mathsf{numeral}\ x))]\!]\end{aligned}$$

Desugaring the pattern matching notation gives us

$$* \vdash^0 \lambda n.\ \mathsf{ind}_{\mathbb{N}}(x.\Box\mathbb{N}, [\![\mathsf{z}]\!], xp.[\![\mathsf{s}(\$(p))]\!], n) : \mathbb{N} \to \Box\mathbb{N}$$

We first elaborate the base case in order to demonstrate an example with no splices. The base case has the following derivation:

$$\delta_1 = \dfrac{\dfrac{n : (\mathbb{N}, 0)\ \mathsf{ctx}}{n : (\mathbb{N}, 0) \vdash^1 \mathsf{z} : \mathbb{N}}}{n : (\mathbb{N}, 0) \vdash^0 [\![\mathsf{z}]\!] : \Box\mathbb{N}}$$

Elaborating the subderivation δ_1 gives us an empty splice environment, since there are no splices.

$$(\!|\, \delta_1 \,|\!) = \dfrac{n : \mathbb{N}\ \mathsf{ctx}}{n : \mathbb{N} \vdash \mathsf{z} : \mathbb{N}} \;\Big|\; *$$

We then take the code of $(\!|\, \delta_1 \,|\!)$ in the elaboration of the whole derivation:

$$\left.\dfrac{\begin{array}{ccc}\vdots & \vdots & \dfrac{???}{n:\mathbb{N}\vdash ??? : \overline{\mathsf{prf}}\ \ulcorner M'\urcorner_{\phi.n}\ \ulcorner \langle\!|\ \delta\ |\!\rangle\urcorner_{\phi.n}\ \ulcorner A'\urcorner_{\phi.n}} \\ n:\mathbb{N}\vdash \ulcorner \mathsf{z}\urcorner : \mathbb{N} & n:\mathbb{N}\vdash \ulcorner \langle\!|\ \delta_1\ |\!\rangle\urcorner : \mathbb{N} & \end{array}}{n:\mathbb{N}\vdash \langle \ulcorner \mathsf{z}\urcorner, \ulcorner \langle\!|\ \delta_1\ |\!\rangle\urcorner, ??? \rangle : \textstyle\sum_{xy:\mathbb{N}}(\overline{\mathsf{prf}}\ x\ y\ \ulcorner\mathbb{N}\urcorner)}\right| *$$

Now, we move on to the inductive case, which demonstrates how a splice is handled. The derivation of the inductive case is

$$\delta_2 = \dfrac{n:(\mathbb{N},0), x:(\mathbb{N},0), p:(\Box\mathbb{N},0)\ \mathsf{ctx}}{n:(\mathbb{N},0), x:(\mathbb{N},0), p:(\Box\mathbb{N},0)\vdash^0 p:\Box\mathbb{N}}$$

$$\delta_3 = \dfrac{\dfrac{n:(\mathbb{N},0), x:(\mathbb{N},0), p:(\Box\mathbb{N},0)\vdash^1 \$(p):\mathbb{N}}{n:(\mathbb{N},0), x:(\mathbb{N},0), p:(\Box\mathbb{N},0)\vdash^1 \mathsf{s}(\$(p)):\mathbb{N}}}{n:(\mathbb{N},0), x:(\mathbb{N},0), p:(\Box\mathbb{N},0)\vdash^0 [\![\mathsf{s}(\$(p))]\!]:\Box\mathbb{N}}$$

For the elaboration of δ_2, we obtain

$$\langle\!|\ \delta_2\ |\!\rangle = \left.\dfrac{n:\mathbb{N}, x:\mathbb{N}, p:\textstyle\sum_{xy:\mathbb{N}}(\overline{\mathsf{prf}}\ x\ y\ \ulcorner\mathbb{N}\urcorner)\ \mathsf{ctx}}{n:\mathbb{N}, x:\mathbb{N}, p:\textstyle\sum_{xy:\mathbb{N}}(\overline{\mathsf{prf}}\ x\ y\ \ulcorner\mathbb{N}\urcorner)\vdash p:\textstyle\sum_{xy:\mathbb{N}}(\overline{\mathsf{prf}}\ x\ y\ \ulcorner\mathbb{N}\urcorner)}\right| *$$

$\langle\!|\ \delta_2\ |\!\rangle$ is then placed in the splice environment when we elaborate δ_3. The splice variable s replaces it as a placeholder.

$$\langle\!|\ \delta_3\ |\!\rangle = \left.\dfrac{\dfrac{n:\mathbb{N}, x:\mathbb{N}, p:\textstyle\sum_{xy:\mathbb{N}}(\overline{\mathsf{prf}}\ x\ y\ \ulcorner\mathbb{N}\urcorner), s:\mathbb{N}\ \mathsf{ctx}}{n:\mathbb{N}, x:\mathbb{N}, p:\textstyle\sum_{xy:\mathbb{N}}(\overline{\mathsf{prf}}\ x\ y\ \ulcorner\mathbb{N}\urcorner), s:\mathbb{N}\vdash s:\mathbb{N}}}{n:\mathbb{N}, x:\mathbb{N}, p:\textstyle\sum_{xy:\mathbb{N}}(\overline{\mathsf{prf}}\ x\ y\ \ulcorner\mathbb{N}\urcorner), s:\mathbb{N}\vdash \mathsf{s}(s):\mathbb{N}}\right| s:\mathbb{N}\overset{0}{\longmapsto} p; \langle\!|\ \delta_2\ |\!\rangle$$

Finally, we elaborate the overall derivation. Since the quote occurs at level 0, it captures all splices at level 0, including the one we just added to the splice environment. Hence, we remove it from the context and the splice environment, substituting the term p for s into the codes of $\mathsf{s}(s)$ and $\langle\!|\ \delta_3\ |\!\rangle$, and using $\langle\!|\ \delta_2\ |\!\rangle$ in the derivation of these substituted codes. Letting $\Gamma = n:\mathbb{N}, x:\mathbb{N}, p:\sum_{xy:\mathbb{N}}(\overline{\mathsf{prf}}\ x\ y\ \ulcorner\mathbb{N}\urcorner)$ and $\phi = s:\mathbb{N}\overset{0}{\longmapsto} p; \langle\!|\ \delta_2\ |\!\rangle$, the result of the elaboration becomes

$$\left.\dfrac{\begin{array}{ccc}\vdots & \vdots & \dfrac{???}{\Gamma\vdash ??? : \overline{\mathsf{prf}}\ \ulcorner \mathsf{s}(s)\urcorner_\phi\ \ulcorner \langle\!|\ \delta_3\ |\!\rangle\urcorner_\phi\ \ulcorner\mathbb{N}\urcorner_\phi} \\ \langle\!|\ \delta_2\ |\!\rangle & \langle\!|\ \delta_2\ |\!\rangle & \\ \Gamma\vdash \ulcorner \mathsf{s}\urcorner_\phi : \mathbb{N} & \Gamma\vdash \ulcorner \langle\!|\ \delta_3\ |\!\rangle\urcorner_\phi : \mathbb{N} & \end{array}}{\Gamma\vdash \langle \mathsf{subst\text{-}term}^*\ \pi_1(p)\ \ulcorner s\urcorner\ \ulcorner \mathsf{s}(s)\urcorner, \ulcorner \langle\!|\ \delta_3\ |\!\rangle\urcorner_\phi, ??? \rangle : \textstyle\sum_{xy:\mathbb{N}}(\overline{\mathsf{prf}}\ x\ y\ \ulcorner\mathbb{N}\urcorner_\phi)}\right| *$$

The substitution operation $\ulcorner \mathsf{s}\urcorner_\phi$ in the conclusion is unfolded to demonstrate the resulting splice substitution.

6.3 Type Soundness of MLTT$^{\mathsf{lvl}}$

6.3.1 Computation and Congruence Rules of $\Box$

We can determine how the definitional equality rules of **MLTT$^{\mathsf{lvl}}$** - including the computation rules for $\Box$ - should behave in order to be sound with respect to the provability semantics. Informally speaking,

soundness means that if $M \equiv N$ in **MLTT$^{\text{lvl}}$**, and they elaborate into M' N' in **MLTT**, then we expect $M' \equiv N'$ just as well.

Any term M at level greater than 0 is eventually contained in a quote, which as we've seen from the provability semantics means that it is eventually converted into a natural number coding the term. Since the code represents the particular syntactic form of M, it is not definitionally equal to the code of a syntactically different term N, even if M and N are themselves definitionally equal. Therefore, this suggests we can only allow computation rules at level 0.

Definition 6.13 (Computation Rules For MLTT$^{\text{lvl}}$ Connectives Except $\Box$)
The computation rules for the **MLTT$^{\text{lvl}}$** connectives remain the same, but they are only allowed at level 0. This is accomplished by annotating the definitional equality judgement with levels, as we did with the typing judgement. For example, the computational rule for Π becomes

$$\frac{\Gamma, x : (A, 0) \vdash^0 M : B \qquad \Gamma \vdash^0 N : A}{\Gamma \vdash^0 (\lambda x.\ M)\ N \equiv M[N/x] : B[N/x]}\ (\Pi\text{C})$$

The exception to this restriction are splices. All splices must occur at a level greater than 1 anyway, but at level 1, we see that the provability semantics elaborates it into a substitution operation *outside* of the code. Hence, we allow a splice to cancel out a quote at level 1.

Definition 6.14 (Computation Rule For $\Box$ in MLTT$^{\text{lvl}}$)

$$\frac{\Gamma \vdash^1 M : A}{\Gamma \vdash^1 \$([\![M]\!]) \equiv M : A}\ (\Pi\text{C})$$

terms at level 0 and splices at level 1 can still occur as a subterm under an arbitrary amount of quotes and splices, so we maintain the congruence rules for all term formers. In particular this includes quotes, contrary to the analysis of Pfenning & Davies in the previous chapter.

6.3.2 Progress & Preservation

Having established the computation rules of **MLTT$^{\text{lvl}}$**, we may now prove the type soundness of **MLTT$^{\text{lvl}}$**. Because definitional equality is a typed judgement, it is particularly easy to establish that definitional equality preserves types: if $M : A$ and $M \equiv N$, then $N : A$. In fact, we can prove a stronger result.

Theorem 6.15 (Preservation)
If $\Gamma \vdash^n M \equiv N : A$, then $\Gamma \vdash^n M : A$ and $\Gamma \vdash^n N : A$.

Proof. By induction on $\Gamma \vdash^n M \equiv N : A$. For the full proof, see Appendix 2.3. ⊣

While preservation ensures that computation rules respect typing, progress ensures that any closed term can continue to progress its computation until it becomes a value of the given type. This requires us to first describe what the values of **MLTT$^{\text{lvl}}$** are.

Typically, a term is a value if it is in *canonical form*, which means it is composed entirely of constructors with the idea that all the eliminators have been cancelled out so no more computation rules may be

applied. In **MLTT$^{\text{lvl}}$**, we restrict computation inside quotes since they are meant to be code objects, so a value is no longer necessarily canonical. The notion of a value changes depending on the level. An additional complication is that for abstraction $\lambda x.\ M$, M no longer has to be closed, so we need a more general description of terms that are "stuck" in computation, which we say is in *normal form*. A term is in canonical form if it is normal and closed.

Definition 6.16 (Normal Forms of MLTT$^{\text{lvl}}$)
We describe normal forms mutually along with neutral terms, which describe destructors that are stuck and not able to compute yet. They are defined as untyped, level-annotated, inductively defined relations $M\ \text{normal}_n$ and $M\ \text{neutral}_n$.

First of all, any neutral term is normal since it cannot compute further.

$$\frac{M\ \text{neutral}_n}{M\ \text{normal}_n}$$

Additionally, any term in constructor form is always in normal form as long as its subterms are, at all levels. This includes type formers, with the understanding that they are the constructors of $\mathcal{U}_i$. For quotes and $\Box$, we have to shift the level up by one.

$$\frac{M\ \text{normal}_{n+1}}{[\![M]\!]\ \text{normal}_n} \qquad \frac{A\ \text{normal}_{n+1}}{\Box A\ \text{normal}_n} \qquad \frac{M\ \text{normal}_n}{\lambda x.\ M\ \text{normal}_n} \qquad \frac{}{\mathsf{z}\ \text{normal}_n} \qquad \frac{M\ \text{normal}_n}{\mathsf{s}(M)\ \text{normal}_n}\ \dots$$

Variables & Destructors are also normal at any level above 1, where it becomes code. The exception to this is splices, which can only occur in a normal form at level 2 or more.

$$\frac{}{x\ \text{normal}_{n+1}} \qquad \frac{M\ \text{normal}_{n+1}}{\$(M)\ \text{normal}_{n+2}} \qquad \frac{M\ \text{normal}_n \qquad N\ \text{normal}_n}{M\ N\ \text{normal}_n}$$

$$\frac{A\ \text{normal}_n \qquad M_1\ \text{normal}_n \qquad M_2\ \text{normal}_n \qquad M_3\ \text{normal}_n}{\text{ind}_{\mathbb{N}}(x.A, M_1, xp.M_2, M_3)\ \text{normal}_n}\ \dots$$

The neutral terms describe when a destructor is stuck on a variable, but otherwise should be allowed to compute. Hence, we only have description for neutrals only at level 1 for splices and at level 0 for the other destructors, since the higher levels are already covered as normal forms.

$$\frac{}{x\ \text{neutral}_0} \qquad \frac{M\ \text{neutral}_0}{\$(M)\ \text{neutral}_1} \qquad \frac{M\ \text{neutral}_0 \qquad N\ \text{normal}_0}{M\ N\ \text{neutral}_0}$$

$$\frac{A\ \text{normal}_0 \qquad M_1\ \text{normal}_0 \qquad M_2\ \text{normal}_0 \qquad M_3\ \text{neutral}_0}{\text{ind}_{\mathbb{N}}(x.A, M_1, xp.M_2, M_3)\ \text{neutral}_0}\ \dots$$

With this definition, we can now state and prove the progress theorem.

Theorem 6.17 (Progress)
If $* \vdash^n M : A$, then either $M\ \text{normal}_n$ or there is a term N such that $* \vdash^n M \rightsquigarrow_\beta N : A$. Here, we use $\rightsquigarrow_\beta$ to refer to a version of the definitional equality judgement without the equivalence closure rules, only computation and congruence rules.

Proof. By induction on $* \vdash^n M : A$. For the full proof, see Appendix 2.4. ⊣

7 ⊢ Evaluation

7.1 Incompleteness of MLTT

The proof of **MLTT**'s incompleteness proceeds very similarly to the standard proof for **PA**. However, we had to make a change in the definition of ω-consistency. Even though the new and old definitions are classically equivalent, they are not intuitionistically equivalent.

One possible explanation as to why the original definition does not work in **MLTT** is that the intuitionistic character of the existential quantifier makes the original definition too trivial. A proof of $\sum_{x:\mathbb{N}} A\ x$ constitutes a pair $\langle M : \mathbb{N}, N : A\ M\rangle$. Since this is a proof from the empty context, we can argue that M has to be definitionally equal to a term in canonical form, i.e. some m such that $\overline{m}$. Hence, this means any proof of $\sum_{x:\mathbb{N}} A\ x$ must include a proof of some $A\ \overline{m}$, contradicting the assumption. In other words, the original definition is simply a re-statement of the intuitionistic character of the existential quantifier, as defined in the BHK interpretation. It does not state anything new or profound about the theory.

On the other hand, we cannot construct such an argument for the new definition since a proof of $\neg \prod_{x:\mathbb{N}} A\ x$ does not constitute any proof of $A\ \overline{m}$. This suggests that our new definition is indeed the better definition when working intuitionistically.

7.2 MLTT$^{\mathsf{lvl}}$

7.2.1 The Provability Semantics

The provability semantics provided some justification for the use of quasiquotes and splices in **MLTT**$^{\mathsf{lvl}}$ allowing the expression of staged metaprograms. However, due to the incompatibility between provability and axiom (**T**), **MLTT**$^{\mathsf{lvl}}$ is missing the ability to evaluate code. This is a crucial aspect of staged metaprogramming systems, as it allows the output of metaprograms to actually be used. As it stands, our theory can only express metaprograms and reason about their output (see the next subsection for an example), but not to use them.

Additionally, we ultimately had to quite heavily restrict the computation rules in the theory to only operate at level 0. The impact of this restriction can be demonstrated with an example that would otherwise have worked if we had computation rules at level 1.

We will attempt to use **MLTT**$^{\mathsf{lvl}}$ to prove a formalisation of Lemma 4.21, a metatheorem which we had earlier used to establish that **MLTT** represents all computable functions. Omitting the case $n = 0$ which we had to handle separately as a special case, we can re-state the lemma more cleanly: for every natural number n, there exists a term canon such that $* \vdash \text{canon} : \prod_{x:\mathbb{N}}(x < \overline{n+1} \to (x = \overline{0} + \ldots + x = \overline{n}))$.

Expressing the formalised version of the lemma requires the construction of two simple metaprograms, one which represents the $\overline{n}$ operation and one which represents the construction of $x = \overline{0} + \ldots + x = \overline{n}$. The former we have seen in the previous chapter as the example term that we used to demonstrate the provability semantics.

$$
\begin{array}{ll}
\text{numeral} : \mathbb{N} \to \square\mathbb{N} & \text{disjunct} : \mathbb{N} \to \square(\mathbb{N} \to \mathcal{U}) \\
\text{numeral z} :\equiv [\![\text{z}]\!] & \text{disjunct z} :\equiv [\![\lambda x.\ x = \text{z}]\!] \\
\text{numeral s}(n) :\equiv [\![\text{s}(\$(\text{numeral } n))]\!] & \text{disjunct s}(n) :\equiv [\![\lambda x.\ (\$(\text{disjunct } n)\ x) + (x = \$(\text{numeral s}(n)))]\!]
\end{array}
$$

Then, the statement of the formalised lemma can be expressed as

$$\prod_{n:\mathbb{N}} \square \prod_{x:\mathbb{N}} (x < \$(\text{numeral s}(n)) \to \$(\text{disjunct } n)\ x)$$

Notice that in expressing disjunct, we had to abstract over x to ensure that the term inside the quasiquote

is closed and does not refer to the free occurence of x. As a result, we then have to apply $(disjunct n) to x whenever we want to use disjunct, particularly in the recursive definition of disjunct itself and in the formalised lemma's statement. Now, to prove the formalised lemma, we would expect to perform induction on n. However, we encounter issues when attempting the inductive case, in which we must provide a term of type

$$n : (\mathbb{N}, 0), p : (\Box \prod_{x:\mathbb{N}} (x < \$(\text{numeral s}(n)) \to \$(\text{disjunct } n)\ x), 0) \vdash^1 \prod_{x:\mathbb{N}} (x < \$(\text{numeral s}(\text{s}(n))) \to \$(\text{disjunct s}(n))\ x)$$

If we allow the definition of disjunct to unfold, this would be definitionally equal to

$$n : (\mathbb{N}, 0), p : (\Box \prod_{x:\mathbb{N}} (x < \$(\text{numeral s}(n)) \to \$(\text{disjunct } n)\ x), 0) \vdash^1 \prod_{x:\mathbb{N}} (x < \$(\text{numeral s}(\text{s}(n))) \to (\$(\text{disjunct } n)\ x) + (x = \$(\text{numeral s}(n))))$$

which we can prove (with some effort) using the the inductive hypothesis p. However, this unfolding takes place at level 1, so requires the use of $\mathbb{N}$. and Π's computation rule at level 1, which is disallowed by the provability semantics.

With all the problems we have encountered between the incompatibility of provability and staged metaprogramming, and this restriction on computational rules, the conclusion appears to be that provability is not a good way to justify staged metaprogramming.

7.2.2 Expressivity of MLTT$^{\text{lvl}}$

We have so far restricted our work on equipping **MLTT$^{\text{lvl}}$** with staged metaprogramming capabilities. This is because staged metaprogramming has been a major focus in homogenous metaprogramming research, and has a straightforward modal interpretation as explained in Chapter 5. However, staged metaprogramming is purely generative [7]: it only provides primitives for the construction of code, but not the inspection of code. In **MLTT$^{\text{lvl}}$**, we can construct code using a quasiquotation, but we cannot inspect the contents of code, for example by pattern matching on the syntax. A theory capable of such inspection is likely to take us beyond modal logic, although it may still be possible to give it a provability interpretation.

In the introduction, we discussed tactics as a kind of metaprograms that aid in the construction of proofs. However, even the simplest tactics require some capability for inspection, since it has to inspect the overall form of the theorem statement in order to decide the appropriate proof to generate. Furthermore, the lack of evaluation prevents us from using the output of such tactics, even if we could express them. Hence, **MLTT$^{\text{lvl}}$** itself is still a far cry from providing a practical metaprogramming experience for theorem provers.

7.3 Ethical Considerations

This project has not involved any animals, human or otherwise, other than the author of this report and her supervisor. There is no data collected on these two persons either. Due to the project's very theoretical nature, we do not anticipate this project to have any substantial impact on developing countries or the environment. We also doubt that this project is likely to be of use to the military or terrorist organisations, not least because we have ended up with a negative result. Any autonomous robots based on the theory we developed are not likely to be very functional.

Finally, there are no concerns with software copyright since we have not used any software other than LaTeX and the packages on CTAN, which have both been made available for free use.

8 ⊢ Conclusion

8.1 Summary

We started with the intention of equipping intuitionistic type theory with metaprogramming primitives so that it may express its own metaprograms and more importantly, to reason about its own metaprograms. In order to justify the primitives, we interpreted them using notions from provability.

We began by exploring the central theorem in provability: Gödel's incompleteness theorems. Gödel's proof establishes that a sufficiently strong theory is already innately capable of a form of metaprogramming. In particular, Gödel encoded formulas and derivations of the formal theory of arithmetic **PA** as numbers, allowing **PA** to represent operations that manipulate its own formulas/derivations and to reason about them.

PA is formulated in classical logic, so the connection between provability and metaprogramming is initially not so clear, since classical logic has no computational interpretation. However, intuitionistic logic has a deep connection to computation via the Curry-Howard Correspondence. We introduced Martin-Löf's intuitionistic type theory (**MLTT**), which fully leverages the Curry-Howard correspondence between logical proofs and functional programs. Re-establishing Gödel's theorems in **MLTT** reveals the connection between provability and metaprogramming.

In the next chapter, we put this connection between provability and metaprogramming to the test. In the metaprogramming literature, it is quite well-known that staged metaprogramming has some modal qualities. Independently of this, the use of modal logic to study provability has also been established. Under the framework of modal logic, we compared and contrasted the two notions of provability and staged metaprogramming, discovering an incompatibility in the form of the evaluation principle, expressed modally as axiom (**T**) $\Box A \to A$. The evaluation principle is key as it allows metaprograms to be used, rather than just to be reasoned about. This means that to incorporate metaprogramming into **MLTT** and justify it with a provability interpretation, we will have to sacrifice the ability to actually use the metaprograms.

Our work culminates by showing that a restricted version of the staged metaprogramming primitives can already be simulated in terms of provability in **MLTT**. Hence, the primitives simply serve as a convenient way to access the provability constructions. Unfortunately, in order to remain faithful to the provability semantics, we had to bend over backwards by also heavily restricting the computation rules of the theory. Ultimately, this makes the theory very impractical and unergonomic. Together with the aforementioned incompatibility, this suggests that while provability and metaprogramming share some similarities, provability is not a good way to justify the principles of practical metaprogramming. Ultimately, it is best if the two concepts are kept distinct.

8.2 Future Work

Establishing The HBL Conditions For $\mathsf{Prov}_{\mathbf{MLTT}}$ In proving the second incompleteness theorem of **MLTT**, we simply assumed that $\mathsf{Prov}_{\mathbf{MLTT}}$ satisfies the Hilbert-Bernays-Löb conditions. It is not necessarily true that $\mathsf{Prov}_{\mathbf{MLTT}}$ satisfies these conditions, since a proof of the satisfaction entails working with the fine details of the encoding of derivations and the definition of prf. In fact, even in **PA** many alternate definitions of $\mathsf{Prov}_{\mathbf{PA}}$ fail to satisfy some of the HBL conditions (e.g. Rosser's definition [15, 56]). Some additional work is warranted to establish that the HBL conditions hold of $\mathsf{Prov}_{\mathbf{MLTT}}$.

Fleshing Out The Provability Semantics Due to time constraints, we could not properly flesh out the provability semantics. In particular, we are missing a treatment of how to elaborate contexts, and we did not at all explore the intricacies of how to build the proof of $\mathsf{prf}_{\mathbf{MLTT}}(\ulcorner M \urcorner, \ulcorner \delta \urcorner, \ulcorner A \urcorner)$.

Extending the Provability Semantics to Modal Axiom (**4**) In principle, provability also validates axiom (**4**) $\Box A \to \Box\Box A$. This suggests we can extend the provability semantics to cover this axiom as

well. In metaprogramming terms, axiom (**4**) corresponds to a relaxation of the level restriction, allowing variables from any lower level to be spliced, rather than only exactly one level below.

Contextual Modal Type Theory The requirement that a term of $\Box A$ be the code of a *closed* term with type A is an awkward one, as we saw in the example in the evaluation. Contextual Modal Type Theory (**CMTT**) [57, 58] explores a type $[\psi]A$, whose terms are codes of terms with type A under the context ψ. We may consider an extension of **CMTT** to dependent types, as we did with $\Box$. This can be done under the framework of graded modal dependent type theory [59], where a graded modality is a modality indexed by a monoid. At least superficially, $[\psi]A$ resembles a graded monoid since context lists are monoids[1].

Categorical Semantics In [60], Kavvos investigated a generalisation of Gödel codes to a categorical setting. We may consider a generalisation of the provability semantics to consider these categories with their own notion of code. This is not likely to work however, considering how difficult and finicky it was just to establish the provability semantics for **MLTT**.

The following two points are more speculative, pertaining to modal logic more generally.

Quasiquotes & Splices Beyond Metaprogramming We were able to derive a meaningful metaprogramming interpretation of Fitch's general system of natural deduction for modal logic. The opening of a strict subderivation corresponds to quasiquoting while splicing corresponds to import rules. From the viewpoint of the possible world semantics of modal logic, a strict subderivation corresponds to "lifting" the reasoning over an arbitrary possible world. This suggests quasiquotes take on the same role in the wider context of modal logic. It may be worth looking into other interpretations modal logic and importing them to the intuitionistic setting to observe the role played by quotes and splices.

Modal Possibility Our work is purely pre-occupied with the necessity modality $\Box$. However, modal logic is traditionally concerned also with the possibility modality $\Diamond$. In classical modal logic, $\Diamond$ can be defined as $\neg\Box\neg$. However, just as the quantifiers are no longer interdefinable in intuitionistic logic, $\Diamond$ and $\Box$ also lose their interdefinability. It would be interesting to identify whether $\Diamond$ can be incorporated into the modal lambda calculus. As a corollary of this, we may obtain a metaprogramming interpretation for $\Diamond$.

[1] The free monoid structure is a list.

A ⊢ Lemmas For Establishing The Representability of Recursive Functions

1.1 Proof of Lemma 4.21

In the following proof, we will utilise the fact that terms of the following types exist. Since they are not metatheorems, they can easily be proven in a theorem prover such as Agda. Hence, we will omit the proofs.

Lemma A.1

1. succ-cong : $\prod_{x\,y:\mathbb{N}}(\mathsf{s}(x) = \mathsf{s}(y) \to x = y)$
2. plus-symm : $\prod_{x\,y:\mathbb{N}} x + y = y + x$
3. plus-eq-zero : $\prod_{x\,y:\mathbb{N}} x + y = \mathsf{z} \to (x = \mathsf{z} \times y = \mathsf{z})$
4. split : $\prod_{x\,y:\mathbb{N}}(x < \mathsf{s}(y) \to (x < y + x = y))$

Assume x and y as implicit arguments since their values can be inferred from the third argument.

The proof is by induction (in the metatheory) on n. We consider two base cases $n = 0$ and $n = 1$, since the type is different for $n = 0$.

Base Case ($n = 0$) We do this by induction on the definition of $x < \mathsf{z}$. The definition of $x < \mathsf{z}$ constitutes a nonzero number k that acts as the difference between x and z. We case split on k - if k is zero then we have a contradiction since we assumed it to be nonzero. If it is a successor, then we also derive a contradiction since we have that x plus k equals zero, which means k must be zero.

$$\begin{aligned}
&\mathsf{canon}_0 : \textstyle\prod_{x:\mathbb{N}} x < \mathsf{z} \to \mathbb{0}\\
&\mathsf{canon}_0\ x\ \langle \mathsf{z}, \langle p_1, p_2\rangle\rangle \quad :\equiv p_1\ \mathsf{refl}_{\mathsf{z}}\\
&\mathsf{canon}_0\ x\ \langle \mathsf{s}(k), \langle p_1, p_2\rangle\rangle :\equiv \text{obs-of-id}\ (\pi_2(\text{plus-eq-zero}\ p_2))
\end{aligned}$$

Base Case ($n = 1$) For this case, we case split on x. When x is zero, then its trivial. When x is the successor, i.e. $\mathsf{s}(x)$, we have that p_2 : plus $\mathsf{s}(x)$ $k = \mathsf{s}(\mathsf{z})$. Since addition is symmetrical, we have plus k $\mathsf{s}(x) = \mathsf{s}(\mathsf{z})$, which is definitionally equal to $\mathsf{s}(\text{plus } k\ x) = \mathsf{s}(\mathsf{z})$. Hence, plus k x = z which means x = z. Here trans : $\prod_{x\,y\,z:\mathbb{N}}(x = y \to y = z \to y = z)$ is the transitivity of equality [31]. The first three arguments are assumed to be implicitly given since they can be inferred from the next two arguments.

$$\begin{aligned}
&\mathsf{canon}_1 : \textstyle\prod_{x:\mathbb{N}} x < \mathsf{s}(\mathsf{z}) \to x = \mathsf{z}\\
&\mathsf{canon}_1\ \mathsf{z}\ p \qquad\qquad\quad :\equiv \mathsf{refl}_{\mathsf{z}}\\
&\mathsf{canon}_1\ \mathsf{s}(x)\ \langle k, \langle p_1, p_2\rangle\rangle :\equiv \pi_2(\text{plus-eq-zero}\ (\text{succ-cong}\ (\text{trans}\ (\text{plus-symm}\ k\ (\mathsf{s}(x)))\ p_2))
\end{aligned}$$

Inductive Case ($n = n + 1$) We have here the inductive hypothesis $\mathsf{canon}_n : \prod_{x:\mathbb{N}}(x < \overline{n} \to (x = \overline{0} + \ldots + x = \overline{n-1}))$. For the inductive case, we simply have to case-split on $x < \overline{n+1}$. If $x < \overline{n}$, then apply the IH. Otherwise, if $x = \overline{n}$ we already have the answer.

$$\begin{aligned}
&\mathsf{canon}_{n+1} : \textstyle\prod_{x:\mathbb{N}}(x < \overline{n+1} \to (x = \overline{0} + \ldots + x = \overline{n-1} + x = \overline{n}))\\
&\mathsf{canon}_{n+1}\ x\ p :\equiv \mathsf{ind}_+(h_<.\mathsf{canon}_n\ h_<, h_=.\mathsf{in}_2(h_=), \text{split}\ p)
\end{aligned}$$

1.2 Proof of Lemma 4.22

As with Lemma 4.21, we have to assume some simple lemmas that are fairly tedious but may be proven in a theorem prover such as Agda.

Lemma A.2

1. zero-plus-id : $\prod_{x:\mathbb{N}} \mathsf{z} + x = x$
2. ap : $\prod_{f:\mathbb{N}\to\mathbb{N}} \prod_{xy:\mathbb{N}} x = y \to f\ x = f\ y$
3. succ-plus : $\prod_{xy:\mathbb{N}} \mathsf{plus}\ \mathsf{s}(x)\ y = \mathsf{s}(\mathsf{plus}\ x\ y)$

For 2. assume f, x and y as implicit arguments since their values can be inferred from the next arguments.

$$
\begin{aligned}
&\mathsf{trichotomy} : \textstyle\prod_{xy:\mathbb{N}}((y < x + x < y) + x = y)\\
&\mathsf{trichotomy}\ \mathsf{z}\ \mathsf{z} \quad :\equiv \mathsf{in}_2(\mathsf{refl}_{\mathsf{z}})\\
&\mathsf{trichotomy}\ \mathsf{z}\ \mathsf{s}(y) \quad :\equiv \mathsf{in}_1(\mathsf{in}_2(\langle \mathsf{s}(y), \langle \lambda r.\ \text{obs-of-id}\ r, \text{zero-plus-id}\ \mathsf{s}(y)\rangle\rangle))\\
&\mathsf{trichotomy}\ \mathsf{s}(x)\ \mathsf{z} \quad :\equiv \mathsf{in}_1(\langle \mathsf{s}(x), \langle \lambda r.\ \text{obs-of-id}\ r, \text{zero-plus-id}\ \mathsf{s}(x)\rangle\rangle)\\
&\mathsf{trichotomy}\ \mathsf{s}(x)\ \mathsf{s}(y) :\equiv \mathsf{ind}_+(\_.(\mathsf{s}(y) < \mathsf{s}(x) + \mathsf{s}(x) < \mathsf{s}(y)) + \mathsf{s}(x) = \mathsf{s}(y),\\
&\qquad\qquad p_1.\mathsf{in}_1(\mathsf{ind}_+(\_.\mathsf{s}(y) < \mathsf{s}(x) + \mathsf{s}(x) < \mathsf{s}(y),\\
&\qquad\qquad\qquad p_{11}.\mathsf{in}_1(\mathsf{helper}\ y\ x\ p_{11}),\\
&\qquad\qquad\qquad p_{12}.\mathsf{in}_2(\mathsf{helper}\ x\ y\ p_{12}),\\
&\qquad\qquad\qquad p_1)),\\
&\qquad\qquad p_2.\mathsf{in}_2(\mathsf{ap}\ p_2),\\
&\qquad\qquad \mathsf{trichotomy}\ x\ y)
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathsf{helper} : \textstyle\prod_{xy:\mathbb{N}}(x < y \to \mathsf{s}(x) < \mathsf{s}(y))\\
&\mathsf{helper}\ \langle k, \langle pk, py\rangle\rangle :\equiv \langle k, \langle pk, \mathsf{trans}\ (\text{succ-plus}\ x\ k)\ (\mathsf{ap}\ py)\rangle\rangle
\end{aligned}
$$

In the last case, the proof is simply splitting the recursive case trichotomy $x\ y$. If it is $y < x$ then we prove $\mathsf{s}(y) < \mathsf{s}(x)$. If it is $x < y$ then we prove $\mathsf{s}(x) < \mathsf{s}(y)$. Finally, if $x = y$ then we prove $\mathsf{s}(x) = \mathsf{s}(y)$.

B ⊢ MLTT$^{\mathsf{lvl}}$

2.1 Proof of Theorem 6.4

The proof proceeds by induction on M, generalising Γ, k and A. For the inductive cases, the inductive hypothesis is therefore:

> (**IH**) For all Γ, k, A, if $\Gamma \vdash_{\lambda^{\kappa}} M : A$ then $(\!|\, \Gamma \,|\!)_{||\Gamma||+k} \vdash^{||\Gamma||+k}_{\lambda^{\mathsf{lvl}}} M : A$.

We show some of the important non-trivial cases of the proof. As we shall see, the proof is rather formulaic.

1. **Base Case** (x) Suppose $\Gamma \vdash x : A$. Then this means $\Gamma = \Gamma_1, x : A, \Gamma_2$ where 🔒 $\notin \Gamma_2$. Hence, $||\Gamma_1|| = ||\Gamma||$ and $||\Gamma_2|| = 0$. From this, we may infer that $(\!|\, \Gamma \,|\!)_{||\Gamma||+k} = (\!|\, \Gamma_1 \,|\!)_{||\Gamma||+k}, x : (A, ||\Gamma||+k), (\!|\, \Gamma_2 \,|\!)_{||\Gamma||+k}$, and so may apply the (Ass) rule to derive $(\!|\, \Gamma \,|\!)_{||\Gamma||+k} \vdash^{||\Gamma||+k} x : A$.

2. **Inductive Case** $(\lambda x.\ M)$ Suppose $\Gamma \vdash \lambda x.\ M : A$. Then we can infer that $A = A_1 \to A_2$ and $\Gamma, x : A_1 \vdash M : A_2$. Applying the **IH**, we find that $(\!|\, \Gamma, x : A_1 \,|\!)_{||\Gamma||+k} \vdash^{||\Gamma||+k} M : A_2$ which is equivalent to $(\!|\, \Gamma \,|\!)_{||\Gamma||+k}, x : (A_1, ||\Gamma||+k) \vdash^{||\Gamma||+k} M : A_2$. Applying $(\to \mathrm{I})$, we obtain $(\!|\, \Gamma \,|\!)_{||\Gamma||+k} \vdash^{||\Gamma||+k} \lambda x.\ M : A$.

3. **Inductive Case** $[\![M]\!]$ Suppose $\Gamma \vdash [\![M]\!] : A$. Then we can infer that $A = \Box A_1$ and $\Gamma, 🔒 \vdash M : A_1$. Applying the **IH**, we find that $(\!|\, \Gamma, 🔒 \,|\!)_{||\Gamma||+k+1} \vdash^{||\Gamma||+k+1} M : A_1$ which is equivalent to $(\!|\, \Gamma \,|\!)_{||\Gamma||+k} \vdash^{||\Gamma||+k+1} M : A_1$. Applying $(\Box \mathrm{I})$, we obtain $(\!|\, \Gamma \,|\!)_{||\Gamma||+k} \vdash^{||\Gamma||+k} [\![M]\!] : A$.

4. **Inductive Case** $\$(M)$ Suppose $\Gamma \vdash \$(M) : A$. Then we can infer that $\Gamma = \Gamma_1, 🔒, \Gamma_2$ where 🔒 $\notin \Gamma_2$, and that $\Gamma_1 \vdash M : \Box A$. Applying the **IH**, we find that $(\!|\, \Gamma_1 \,|\!)_{||\Gamma_1||+k} \vdash^{||\Gamma_1||+k} M : \Box A$ which we can apply $(\Box \mathrm{E})$ to, obtaining $(\!|\, \Gamma_1 \,|\!)_{||\Gamma_1||+k} \vdash^{||\Gamma_1||+k+1} \$(M) : A$. We can weaken this into $(\!|\, \Gamma_1 \,|\!)_{||\Gamma_1||+k}, (\!|\, \Gamma_2 \,|\!)_{||\Gamma_1||+k+1} \vdash^{||\Gamma_1||+k+1} \$(M) : A$, which is equivalent to $(\!|\, \Gamma_1, 🔒, \Gamma_2 \,|\!)_{||\Gamma_1||+k+1} \vdash^{||\Gamma_1||+k+1} \$(M) : A$. But of course, $||\Gamma_1|| + k + 1 = ||\Gamma|| + k$ since Γ contains one more lock than Γ_1.

5. etc.

2.2 Proofs for Lemma 6.7

Lemma 6.7.1

For the full proof, we actually have to prove the following two lemmas by mutual induction on the typing and definitional equality judgement.

Lemma B.1
1. If $\Gamma \vdash^n M : A$ then Γ ctx.
2. If $\Gamma \vdash^n M \equiv N : A$ then Γ ctx.

All the inductive cases for our newly added rules are actually trivial since they don't manipulate the context. This means we can simply apply the inductive hypothesis to obtain the goal.

Lemma 6.7.2

As with Lemma 6.7.1, we actually have to perform mutual induction.

Lemma B.2

1. If $\Gamma \vdash^n M : A$ then $\Gamma \vdash^n A : \mathcal{U}_i$.
2. If $\Gamma \vdash^n M \equiv N : A$ then $\Gamma \vdash^n A : \mathcal{U}_i$.

The case for ($\Box$F) is trivial, since $\mathcal{U}_i$ is always a well-formed type. For ($\Box$I), we simply need to apply the inductive hypothesis to obtain $\Gamma \vdash^{n+1} A : \mathcal{U}_i$ and apply ($\Box$F). For ($\Box$E), the inductive hypothesis gives $\Gamma \vdash^n \Box A : \mathcal{U}_i$, so we can infer that $\Gamma \vdash^{n+1} A : \mathcal{U}_i$.

For the congruence rules of $[\![\,]\!]$, $\$()$ and $\Box$, the proof follows closely from ($\Box$I), ($\Box$E) and ($\Box$F) respectively, while the computation rule for $\Box$ trivially follows from the inductive hypothesis.

Lemma 6.7.3 (Weakening)

The proof is by mutual induction over all three typing judgements. We have to

Lemma B.3

1. If $\Gamma, \Delta \vdash^n M : B$ and $\Gamma \vdash^m A : \mathcal{U}_i$, then for any fresh variable x not occuring in Γ nor Δ, $\Gamma, x : (A, m), \Delta \vdash^n M : B$.
2. If $\Gamma, \Delta \vdash^n M \equiv N : B$ and $\Gamma \vdash^m A : \mathcal{U}_i$, then for any fresh variable x not occuring in Γ nor Δ, $\Gamma, x : (A, m), \Delta \vdash^n M \equiv N : B$.
3. If Γ, Δ ctx and $\Gamma \vdash^m A : \mathcal{U}_i$, then for any fresh variable x not occuring in Γ nor Δ, $\Gamma, x : (A, m), \Delta$ ctx.

As with Lemma 6.7.1, this trivially follows from the inductive hypothesis for all the newly added rules in **MLTT**$^{\text{lvl}}$ since none of them manipulate the context.

Lemma 6.7.4 (Substitution)

Lemma B.4

1. If $\Gamma, x : (A, m), \Delta \vdash^n M : B$ and $\Gamma \vdash^m N : A$ then $\Gamma, \Delta[N/x] \vdash^n M[N/x] : B[N/x]$.
2. If $\Gamma, x : (A, m), \Delta \vdash^n M_1 \equiv M_2 : B$ and $\Gamma \vdash^m N : A$ then $\Gamma, \Delta[N/x] \vdash^n M_1[N/x] \equiv M_2[N/x] : B[N/x]$.
3. If $\Gamma, x : (A, m), \Delta$ ctx and $\Gamma \vdash^m N : A$ then $\Gamma, \Delta[N/x]$ ctx.

For our new rules, this also trivially follows from the inductive hypothesis, and an unfolding of the definition of substitution. For example, for ($\Box$E), the inductive hypothesis gives us $\Gamma, \Delta[N/x] \vdash^n M[N/x] : (\Box B)[N/x]$. However we know $(\Box B)[N/x]$ is by definition equivalent to $\Box(B[N/x])]$, so we can simply re-apply ($\Box$E) to obtain $\Gamma, \Delta[N/x] \vdash^n \$(M[N/x]) : B[N/x]$. Once again, $\$(M[N/x])$ is equivalent to $\$(M)[N/x]$.

We note here that weakening is needed to prove the cases for some of the other rules.

Lemma 6.7.5 (Exchange)

Lemma B.5

1. If $\Gamma, x : (A, m), y : (B, n), \Delta \vdash^k M : C$ and $\Gamma \vdash^n B : \mathcal{U}_i$, then $\Gamma, y : (B, n), x : (A, m), \Delta \vdash^k M : C$.

2. If $\Gamma, x : (A, m), y : (B, n), \Delta \vdash^k M \equiv N : C$ and $\Gamma \vdash^n B : \mathcal{U}_i$, then $\Gamma, y : (B, n), x : (A, m), \Delta \vdash^k M \equiv N : C$.
3. If $\Gamma, x : (A, m), y : (B, n), \Delta$ ctx and $\Gamma \vdash^n B : \mathcal{U}_i$, then $\Gamma, y : (B, n), x : (A, m), \Delta$ ctx.

As with Lemma 6.7.3, this trivially follows from the inductive hypothesis for all the newly added rules in **MLTT**$^{\text{lvl}}$ since none of them manipulate the context.

We note here that weakening is needed to prove the cases for some of the other rules.

2.3 Proof of Theorem 6.15 (Preservation)

The proof is by induction on $\Gamma \vdash^n M \equiv N : A$. We prove only the cases for the newly added rules, since the proof will not have changed for the remaining rules.

1. **Base Case** $\dfrac{\Gamma \vdash^1 M : A}{\Gamma \vdash^1 \$(\llbracket M \rrbracket) \equiv M : A}$ ($\Box$C)

   We immediately already have $\Gamma \vdash^1 M : A$ for the LHS. From this, we can simply construct the following derivation for the RHS

   $$\dfrac{\dfrac{\Gamma \vdash^1 M : A}{\Gamma \vdash^0 \llbracket M \rrbracket : \Box A}}{\Gamma \vdash^1 \$(\llbracket M \rrbracket) : A}$$

2. **Inductive Case** $\dfrac{\Gamma \vdash^{n+1} A \equiv A' : \mathcal{U}_i}{\Gamma \vdash^n \Box A \equiv \Box A' : \mathcal{U}_i}$

   Simply apply the inductive hypothesis followed by the ($\Box$F) rule to both the LHS and RHS.

3. **Inductive Case** $\dfrac{\Gamma \vdash^{n+1} M \equiv M' : A}{\Gamma \vdash^n \llbracket M \rrbracket \equiv \llbracket M' \rrbracket : \Box A}$

   Simply apply the inductive hypothesis followed by ($\Box$I) to both the LHS and RHS.

4. **Inductive Case** $\dfrac{\Gamma \vdash^n M \equiv M' : \Box A}{\Gamma \vdash^{n+1} \$(M) \equiv \$(M') : A}$

   Apply the inductive hypothesis followed by ($\Box$E) to both the LHS and RHS.

2.4 Proof of Theorem 6.17 (Progress)

The proof is by induction on $* \vdash^n M : A$. For brevity, we show only the cases for the newly added rules of **MLTT**$^{\text{lvl}}$.

1. **Inductive Case** $\dfrac{\Gamma \vdash^{n+1} A : \mathcal{U}_i}{\Gamma \vdash^n \Box A : \mathcal{U}_i}$ ($\Box$F)

   By the inductive hypothesis, either A normal$_{n+1}$ or $* \vdash^n A \rightsquigarrow_\beta B : \mathcal{U}_i$. In the former case, it immediately follows that $\Box A$ normal$_n$. In the latter case, we can simply apply the congruence rule for $\Box$.

2. **Inductive Case** $\dfrac{\Gamma \vdash^{n+1} M : A}{\Gamma \vdash^n \llbracket M \rrbracket : \Box A}$ ($\Box$I)

   By the inductive hypothesis, either M normal$_{n+1}$ or $* \vdash^n M \rightsquigarrow_\beta N : A$. In the former case, it immediately follows that $\llbracket M \rrbracket$ normal$_n$. In the latter case, we can simply apply the congruence rule for quotes.

3. **Inductive Case** $\dfrac{\Gamma \vdash^{n} M : \Box A}{\Gamma \vdash^{n+1} \$(M) : A}\ (\Box\mathsf{E})$

   By the inductive hypothesis, either M normal_n or $* \vdash^{n} M \rightsquigarrow_{\beta} N : \Box A$. In the latter case, then we can simply apply the congruence rule for splices. In the former case, we need to do a further case split. Due to its type, we know that either $M = [\![M']\!]$ or some destructor term. It cannot be a variable since M is well-typed under the empty context.

   If $M = [\![M']\!]$, and $n = 0$, then we can use computation rule for $\Box$ to show that $\$([\![M']\!])$ reduces. Otherwise, $\$([\![M']\!])$ normal_{n+1}, which works since $n > 0$.

   If M is some destructor, then $n > 0$ since it cannot be a neutral term without a freely occuring variable. Hence, $\$(M)$ normal_{n+1} again.

4. etc.

References

[1] Leonardo de Moura et al. “The Lean Theorem Prover (System Description)”. In: *Automated Deduction - CADE-25*. Springer International Publishing, 2015, pp. 378–388.

[2] Ulf Norell. “Towards a practical programming language based on dependent type theory”. PhD thesis. Chalmers University of Technology and Göteborg University, 2007.

[3] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. en. Springer Science & Business Media, Mar. 2013.

[4] Per Martin-Löf. “An intuitionistic theory of types”. In: *Twenty-five years of constructive type theory (Venice, 1995)*. Ed. by Giovanni Sambin and Jan M Smith. Vol. 36. Oxford Logic Guides. Oxford University Press, 1998, pp. 127–172.

[5] Philip Wadler. “Propositions as types”. In: *Commun. ACM* 58.12 (Nov. 2015), pp. 75–84.

[6] Gabriel Ebner et al. “A metaprogramming framework for formal verification”. In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017), pp. 1–29.

[7] Tim Sheard. “Accomplishments and Research Challenges in Meta-programming”. In: *Semantics, Applications, and Implementation of Program Generation*. Springer Berlin Heidelberg, 2001, pp. 2–44.

[8] Richard Zach. “Hilbert's Program”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N Zalta. Fall 2019. Metaphysics Research Lab, Stanford University, 2019.

[9] Kurt Gödel. “Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I”. In: *Monatshefte für Mathematik und Physik* 38.1 (Dec. 1931), pp. 173–198.

[10] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. en. College Publications, 2012.

[11] Gerhard Gentzen. “Untersuchungen über das logische SchlieSSen. I”. In: *Math. Z.* 39.1 (Dec. 1935), pp. 176–210.

[12] Gerhard Gentzen. “Untersuchungen über das logische SchlieSSen II”. In: *Math. Z.* 39 (1935), pp. 405–431.

[13] Giuseppe Peano. *Arithmetices Principia, Nova Methodo Exposita*. Libreria Bocca, 1889.

[14] Richard Zach. *Incompleteness and Computability: An Open Introduction to Gödel's Theorems*. Open Logic Project, 2019.

[15] Barkley Rosser. “Extensions of some theorems of Gödel and Church”. In: *J. Symbolic Logic* 1.3 (Sept. 1936), pp. 87–91.

[16] Peter Smith. *Introduction to Godel's Theorems, An. Cambridge Introductions to Philosophy*. en. Cambridge University Press, May 2014.

[17] D Hilbert and P Bernays. “Grundlagen der Mathematik II”. In: *J. Symbolic Logic* 39.2 (1974), pp. 357–357.

[18] M H Löb. “Solution of a Problem of Leon Henkin”. In: *J. Symbolic Logic* 20.2 (1955), pp. 115–118.

[19] Mark van Atten. “The Development of Intuitionistic Logic”. In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N Zalta. Summer 2022. Metaphysics Research Lab, Stanford University, 2022.

[20] A M Turing. “On computable numbers, with an application to the entscheidungsproblem”. en. In: *Proc. Lond. Math. Soc.* s2-42.1 (1937), pp. 230–265.

[21] Alonzo Church. “A Set of Postulates for the Foundation of Logic”. In: *Ann. Math.* 33.2 (1932), pp. 346–366.

[22] Morten Heine Sørensen and Pawe Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. en. Elsevier, 2006.

[23] Arend Heyting. “Die intuitionistische Grundlegung der Mathematik”. In: *Erkenntnis* 2 (1931), pp. 106–115.

[24] A Heyting. *Mathematische Grundlagenforschung Intuitionismus Beweistheorie*. Springer Berlin Heidelberg, 1934.

[25] A Kolmogoroff. "Zur Deutung der intuitionistischen Logik". In: *Math. Z.* 35.1 (Dec. 1932), pp. 58–65.

[26] Steffen van Bakel. *Type Systems for Programming Languages*. Online. 2016.

[27] Alonzo Church. "A formulation of the simple theory of types". In: *J. Symbolic Logic* 5.2 (June 1940), pp. 56–68.

[28] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Éditeur inconnu, 1972.

[29] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013.

[30] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. "Observational equality, now!" In: *Proceedings of the 2007 workshop on Programming languages meets program verification*. Freiburg Germany: ACM, Oct. 2007.

[31] Egbert Rijke. *Introduction to Homotopy Type Theory*. Online. 2019.

[32] Martin Hofmann. "Extensional concepts in intensional type theory". en. PhD thesis. July 1995.

[33] Thomas Streicher. "Investigations Into Intensional Type Theory". PhD thesis. Ludwig Maximilian University of Munich, 1993.

[34] Jesper Cockx, Dominique Devriese, and Frank Piessens. "Pattern matching without K". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. ICFP '14. Gothenburg, Sweden: Association for Computing Machinery, Aug. 2014, pp. 257–268.

[35] *Cubical compatible — Agda 2.6.3 documentation*. en. https://agda.readthedocs.io/en/latest/language/cubical-compatible.html. Accessed: 2022-6-3.

[36] James Garson. "Modal Logic". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N Zalta. Summer 2021. Metaphysics Research Lab, Stanford University, 2021.

[37] Rasmus Rendsvig and John Symons. "Epistemic Logic". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N Zalta. Summer 2021. Metaphysics Research Lab, Stanford University, 2021.

[38] Paul McNamara and Frederik Van De Putte. "Deontic Logic". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N Zalta. Spring 2022. Metaphysics Research Lab, Stanford University, 2022.

[39] Patrick Blackburn, Maarten de Rijke, and Yde Venema. *Modal Logic*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.

[40] Sergei N Artemov and Lev D Beklemishev. "Provability Logic". In: *Handbook of Philosophical Logic, 2nd Edition*. Ed. by D M Gabbay and F Guenthner. Dordrecht: Springer Netherlands, 2005, pp. 189–360.

[41] Robert M Solovay. "Provability interpretations of modal logic". In: *Israel J. Math.* 25.3 (Sept. 1976), pp. 287–304.

[42] W V Quine. *Mathematical Logic*. Cambridge: Harvard University Press, 1940.

[43] Rowan Davies and Frank Pfenning. "A modal analysis of staged computation". en. In: *J. ACM* 48.3 (May 2001), pp. 555–604.

[44] Frederic Brenton Fitch. *Symbolic Logic: An Introduction*. en. Ronald Press Company, 1952.

[45] Frederic B Fitch. "Natural Deduction Rules for Obligation". In: *Am. Philos. Q.* 3.1 (1966), pp. 27–38.

[46] Melvin Fitting. "Basic modal logic". In: *Handbook of logic in artificial intelligence and logic programming (vol. 1)*. USA: Oxford University Press, Inc., Aug. 1993, pp. 368–448.

[47] Vaj Tijn Borghuis. "Coming to terms with modal logic : on the interpretation of modalities in typed lambda-calculus". en. PhD thesis. Technische Universiteit Eindhoven, 1994.

[48] Simone Martini and Andrea Masini. "A Computational Interpretation of Modal Proofs". In: *Proof Theory of Modal Logic*. Ed. by Heinrich Wansing. Vol. 2. Applied Logic Series. Springer Dordrecht, Dec. 1995, pp. 213–241.

[49] Ranald Clouston. “Fitch-Style Modal Lambda Calculi”. In: *Lecture Notes in Computer Science*. Lecture notes in computer science. Cham: Springer International Publishing, 2018, pp. 258–275.

[50] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. “Implementing a modal dependent type theory”. In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019), pp. 1–29.

[51] Sergei N Artemov. “Explicit Provability and Constructive Semantics”. In: *Bull. Symbolic Logic* 7.1 (2001), pp. 1–36.

[52] G A Kavvos. “Intensionality, Intensional Recursion, and the Gödel-Löb axiom”. In: *ArXiv* (2017).

[53] Walid Taha and Tim Sheard. “Multi-stage programming with explicit annotations”. In: *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation - PEPM ’97*. Amsterdam, The Netherlands: ACM Press, 1997.

[54] Ningning Xie et al. “Staging with class: a specification for typed template Haskell”. en. In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022), pp. 1–30.

[55] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[56] Toshiyasu Arai. “Derivability conditions on Rosser’s provability predicates”. en. In: *Notre Dame Journal of Formal Logic* 31.4 (Sept. 1990), pp. 487–497.

[57] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. “Contextual modal type theory”. en. In: *ACM Trans. Comput. Log.* 9.3 (June 2008), pp. 1–49.

[58] Aleksandar Nanevski. “Meta-programming with names and necessity”. In: *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*. ICFP ’02. Pittsburgh, PA, USA: Association for Computing Machinery, Sept. 2002, pp. 206–217.

[59] Benjamin Moon, Harley Eades III, and Dominic Orchard. “Graded Modal Dependent Type Theory”. In: *Programming Languages and Systems*. Springer International Publishing, 2021, pp. 462–490.

[60] G A Kavvos. “On the Semantics of Intensionality”. en. In: *Foundations of Software Science and Computation Structures: 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Springer Berlin Heidelberg, Mar. 2017, pp. 550–566.