Dissertation Type: Research

DEPARTMENT OF COMPUTER SCIENCE

Automated Theorem Proving in Category Theory and the λ-calculus

Alessio Zakaria

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Friday 10th May, 2019

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Alessio Zakaria, Friday 10th May, 2019

Contents

Acknowledgements

I would like to thank my supervisor, Dr. Nicolas Wu for his continued enthusiasm and inspiration. I would like to thank Dr. David Bernhard for enabling me to become a mathematician. I would like to thank Csongor Kiss and Jannis Limperg for their guidance though all things types and Agda. Thank you to my friends and family for being everything.

Executive Summary

The research hypothesis of this thesis is that there is a significant advantage to employing computer-aided proof assistants in the study of category theory and mathematical logic. This was done by implementing a theorem within the study of categorical logic in the theorem prover Agda. By doing this, the limitations and importance of theorem proving was explored in a practical way.

- I learnt the underlying theory of modern theorem provers
- I learnt the practical usage of the theorem prover Agda
- I formalised Lawvere's fixed point theorem within Agda
- I obtained a novel application of the theorem within the untyped λ-calculus
- I surveyed modern advancements in type theory that eliminate the limitations of modern theories.

Supporting Technologies

This thesis made use of two supporting technologies

- version 2.5.4.2 of the theorem prover Adga to formalise theorems within Category Theory
- the `cats` library by Jannis Limperg [30]

Chapter 1

Contextual Background

1.1 Introduction

This thesis is a small exploration into the interconnected and mysterious worlds of mathematical logic and computation. Since the advent of the fields in the early 20th Century, links have been found in unusual places and both fields have provided insight into the other. There are two primary focal points of this thesis: the unification of disparate areas in mathematical logic and computing via category theory, an offshoot of abstract algebra; and the rigorous formalisation of mathematics within modern theorem provers. More precisely, a particular theorem within category theory will be proven within the theorem prover Agda and its applications will be explored, with a new application provided in the context of the untyped λ-calculus. The theorem that will be explored within this thesis is Lawvere's fixed point theorem discovered by William Lawvere in 1969 in *Diagonal Arguments and Cartesian Closed Categories* [29]. Lawvere's fixed point theorem is a statement within the context of cartesian closed categories, categories which play a crucial role within the study of computation and logic. Interest was somewhat renewed in Lawvere's theorem in 2003 after a review paper published by Noson Yanofsky [46] detailed how many common paradoxes and results in computing and mathematical logic could be brought within the framework of the theorem. Lawvere's fixed point theorem is a categorical abstraction of the familiar class of *diagonal arguments* employed throughout computer science and mathematics. The decision to formalise Lawvere's fixed point theorem within a theorem prover is not incidental; many of the theorems abstracted by the theorem play an important role in the problem of providing a formal foundation in which to do mathematics.

The primary contributions of this thesis are: a formalisation of Lawvere's fixed point theorem within the theorem prover Agda with additional proofs in the theory of cartesian closed categories; a novel application of Lawvere's fixed point theorem within the context of the untyped λ-calculus; and a formalisation within Agda of a category of small types with presentations of Cantor's Theorem. Given the attempts of this thesis to incorporate a large amount of related yet distinct fields together the contextual history behind the primary areas will be outlined. What follows is a blurred and idealistic exposition of 20^{th} century mathematical logic.

1.2 Foundations of Mathematical Logic and Computation

1.2.1 Cantor's Theorem

The story begins slightly before the advent of the 20^{th} century with the German Mathematician Georg Cantor. Cantor (1845 - 1918) is considered the father of set theory with his proofs of the differing cardinalities of the real and natural numbers and his theory of ordinals. A major piece of work by Cantor was his eponymous theorem [3], published in 1892, that the cardinality of a set is strictly smaller than the cardinality of its powerset. Cantor's proof of this theorem made use of a so-called *diagonal argument*, one of the first of its kind which, repeated through time in different fields, was abstracted by Lawvere to give his fixed point theorem in category theory subsuming, previous instances. Cantor's theorem proceeds as follows:

Theorem (Cantor's Theorem). Let f be a function from a set A to its powerset $\mathcal{P}(A)$. Then f is not a surjective function.

Proof. Aiming for a contradiction assume f is surjective. Consider the set $B = \{a \in A \mid a \notin f(a)\}$. Therefore there exists a $b \in A$ such that $f(b) = B$. By definition $f(b) = B$ and therefore $b \notin B$. However, $b \notin f(b)$ implies $b \in B$. Contradiction. □

The name *diagonal theorem* comes from an earlier proof by Cantor of the uncountability of the real numbers. This was done by assuming an enumeration of the set of infinite sequences of binary digits which are in correspondence with the real numbers, and considering this as a table, see Figure 1.1. The proof then proceeds by going along the diagonal of the table flipping the n^{th} digit of the n^{th} number going down the table to produce a new real number not in the table. The repetition of the argument in two places in the definition yields a familiar structure repeated in many theorems in mathematical logic known as *diagonal arguments*.

$$\begin{array}{|lcl|}
\hline
s_1 & = & \textcolor{red}{0}\;0\;0\;0\;0\;0\;0\;0\;0\;0\;0 \ldots \\
s_2 & = & 1\;\textcolor{red}{1}\;1\;1\;1\;1\;1\;1\;1\;1\;1 \ldots \\
s_3 & = & 0\;1\;\textcolor{red}{0}\;1\;0\;1\;0\;1\;0\;1\;0 \ldots \\
s_4 & = & 1\;0\;1\;\textcolor{red}{0}\;1\;0\;1\;0\;1\;0\;1 \ldots \\
s_5 & = & 1\;1\;0\;1\;\textcolor{red}{0}\;1\;1\;0\;1\;0\;1 \ldots \\
s_6 & = & 0\;0\;1\;1\;0\;\textcolor{red}{1}\;1\;0\;1\;1\;0 \ldots \\
s_7 & = & 1\;0\;0\;0\;1\;0\;\textcolor{red}{0}\;0\;1\;0\;0 \ldots \\
s_8 & = & 0\;0\;1\;1\;0\;0\;1\;\textcolor{red}{1}\;0\;0\;1 \ldots \\
s_9 & = & 1\;1\;0\;0\;1\;1\;0\;0\;\textcolor{red}{1}\;1\;0 \ldots \\
s_{10} & = & 1\;1\;0\;1\;1\;1\;0\;0\;1\;\textcolor{red}{0}\;1 \ldots \\
s_{11} & = & 1\;1\;0\;1\;0\;1\;0\;0\;1\;0\;\textcolor{red}{0} \ldots \\
\vdots & & \vdots\;\vdots\;\vdots\;\vdots\;\vdots\;\vdots\;\vdots\;\vdots\;\vdots\;\vdots\;\vdots\;\ddots \\
\hline
\end{array}$$

$$\begin{array}{|lcl|}
\hline
s & = & \textcolor{blue}{1\;0\;1\;1\;1\;0\;1\;0\;0\;1\;1} \ldots \\
\hline
\end{array}$$

Figure 1.1: Cantor's Diagonal Argument Illustrated

1.2.2 Russell's Paradox

Cantor, in founding set theory as a field of study, introduced his naïve set theory [2] as a foundation in which to do mathematics. The naïvity of Cantor's foundations rested in the **Axiom of Unrestricted Comprehension**, adapted from Gottlob Frege's **Basic Law V** [15]. Where $w_1, \ldots, w_n$ ranges over the words of the language of sets, the axiom states that, for all predicates (functions that return a boolean value), φ, there exists a set, B, whose elements exactly satisfy φ.

Definition (Axiom of Unrestricted Comprehension).

$$\forall\varphi\;\forall w_1, \cdots, w_n\;\exists B. \text{ such that } \forall x\;(x \in B \text{ iff. } \varphi(x, w_1, \cdots, w_2))$$

This lead to perhaps the first significant result in the foundations of mathematics in the 20th century, Russell's paradox. Bertrand Russell (1872-1970), an English polymath, showed in *The Principles of Mathematics* [39] written with Alfred Whitehead in 1903 that, in classical logic, the axiom of unrestricted comprehension leads to a contradiction.

Theorem (Naïve set theory is inconsistent).

Proof. Consider the predicate

$$\varphi(x) = x \notin x$$

By the **Axiom of Unrestricted Comprehension** there is a set X such that all $x \in X$ satisfy φ. Aiming for a contradiction, assume $X \in X$. By definition of φ, $X \notin X$. Contradiction. □

This proof shares the similar diagonal style argument where x appears in the positive and negative positions of the set membership relation $\in$. In modern set theories, this is disposed of in favour of an axiom of restricted comprehension, where all predicates must be defined on an already existing set. For a set theory to be consistent it must therefore reject the **Axiom of Unrestricted Comprehension**. This in turn prevents the existence of a set of all sets from being considered as, by definition, this must contain itself, something which is not supported by restricted comprehension schemes. This result is relevant to both prongs of this thesis. Russell's paradox presented a blow to the hoped foundations of mathematics. Various systems were designed to provide a more secure foundation of mathematics as a result of Russell's paradox. Russell himself created the first theories of types in his *Principia Mathematica* [45] to combat this with various other foundational theories being presented, including set theoretic foundations such as Zermelo-Fraenkel set theory (ZFC) and Von-Neumann-Bernays-Gödel set theory (NBG) which reject the **Axiom of unrestricted Comprehension**. Yanofsky [46] shows how Russell's paradox can be brought within the framework of Lawvere's fixed point theorem.

1.2.3 Computability

Shortly after the work of Cantor and Russell, the field of theoretical computer science was birthed by attempts to answer the Entscheidungsproblem (decision problem), a challenge by David Hilbert [20] which required the formalisation of the notion of algorithm.

Entscheidungsproblem

Devised in 1928 by Hilbert, the Entscheidungsproblem asked for an effective procedure or algorithm which, on an input of a statement in first-order logic plus a finite numbers of axioms, could determine its validity in all structures satisfying the axioms. Before an answer could be given, the notion of algorithm had to be formalised. This was done in the 1930s independently by Alonzo Church and Alan Turing. Turing's formalisation [43] known as Turing Machines, aimed to capture directly the notion of algorithm. Turing provided a negative answer to this question by establishing the existence of undecidable problems such as the Halting problem and confirming the Entscheidungsproblem to be one. Turing's proof of the existence of undecidable problems and his proof of the Halting problem resemble closely the proof of Cantor's theorem being diagonalisation arguments. Yanofsky [46] also shows how both proofs can be understood through the lens of Lawvere's fixed point theorem by considering the category of computable universes or objects which "support computation"

λ-calculus

Instead of capturing directly the notion of computability, Alonzo Church's formalisation was focused on formalising more precisely the notion of a function. Church's formulation, the λ-calculus [4], is a formal system consisting of the operations of abstraction and application using variable binding and substitution. Church showed that the λ-calculus had an notion of λ-definability [5] which was equivalent to the class of general recursive functions [25] as defined by Gödel and clarified by Kleene, another model of computation. Church also provided a negative answer to the Entscheidungsproblem.

The λ-calculus consists of a set of inductively defined terms and rewriting rules governing how to compute with them. In the definitions that follow s and t range over λ-terms and x ranges over variable names.

Definition (The set Λ of λ-terms).

$$x \in \Lambda$$
$$s \in \Lambda \ \wedge \ t \in \Lambda \implies (s\ t) \in \Lambda$$
$$s \in \Lambda \implies \lambda x.s \in \Lambda$$

$(s\ t)$ denotes the application of s to t and $\lambda x.s$ denotes the *capture* or *binding* of the variable x in term s, known as a λ-abstraction. Intuitively, application of a term to a λ-abstraction, when considered as a computation, should replace all instances of the abstracted variable with the term being applied a process known as substitution. This computation is known as β-reduction.

Definition (β-reduction). A term t applied to λ-abstraction $(\lambda x.s)\ t$, *β-reduces* to $s[t/x]$ denoting the substitution of t for x in s written

$$(\lambda x.s)\ t \to_\beta s[t/x]$$

With this notion of reduction an equational theory, $\lambda\beta$ can be defined as the reflexive transitive closure of the relation that $\rightarrow_\beta$ defines.

The λ-calculus originally arose from Church's desire to provide a foundation for logic, a desire also shared by another American Logician, Haskell Curry. In the 1920s and 1930s, Curry worked on a foundation of logic extremely similar to the λ-calculus, his theory of combinators [11]. Combinatory logic, as defined by Curry, was in correspondence with the λ-calculus. In 1934 Curry observed that, if types were assigned to the domains and codomains of his combinators, the types matched the axioms of intuitionistic implicational logic [9]. This observation proved to be the beginning of theorem provers as they are in their current state.

1.2.4 Curry-Howard Correspondence

Curry's combinatory logic and Church's λ-calculus, when viewed as a system for logic, proved to be inconsistent as shown by Kleene and Rosser [24] and again by Curry using his fixed point combinator [10]. To remedy this inconsistency both Curry and Church restricted their logical systems with types, giving typed combinatory logic and the simply typed λ-calculus. Further to Curry's aforementioned observation made of typed combinatory logic, in the 1960s William Alvin Howard made a further observation that the typing rules of the simply λ-calculus corresponded to laws of inference for intuitionistic style natural deduction [21]. This lead to the conclusion that every system in formal logic has a corresponding computational calculi with a specific type system named the Curry-Howard correspondence. The relation of computational calculi to *intuitionistic* logic is an interesting one. Intuitionistic logics are a class of logics that reject common principles of classical logic, namely the Law of Excluded Middle (LEM), ($\vdash p \vee \neg p$), and Double Negation Elimination (DNE) ($\neg\neg p \vdash p$). Intuitionistic Logic is highly related to the philosophical position of mathematical constructivism. Mathematical constructivism places a higher burden on proof when positing the existence of mathematical structures and demands that a mathematical object be explicitly constructed. Both LEM and DNE are able to prove the existence of objects that have not been explicitly constructed through the use of proof by contradiction. Constructivism can be seen as a natural counterpart the philosophical position of intuitionism which asserts that mathematics is an entirely human activity that arises from our mental faculty, contrary to platonsism which makes stronger ontological claims about the objective status of mathematical structures. The effectiveness of the Curry-Howard correspondence in formalising and mechanizing mathematics gives some credence to this theory and the act of theorem proving can be seen as a continuation of the underlying causes of intuitionistic movements.

1.3 History of Theorem Proving

The Curry-Howard correspondence provided a computational interpretation of the notion of proof. Providing a proof of a theorem corresponded to a program that inhabited a given type. Type systems in computation had largely grown with the theory of programming languages. Types were intended to check that a program, to some degree, behaved as intended. It was understood that the comprehensive type systems and their associated checkers could be used to check the validity of proofs in mathematics. With the advent of physical computers this provided hope that a more methodical approach to mathematics that eliminated the uncertainty around new proofs would be achieved.

1.3.1 Automath

The first computational system to exploit the Curry-Howard correspondence to act as a theorem prover was Automath in 1967, slighty before Howard's explicit observation of the Curry-Howard correspondence. Automath (automating mathematics) was designed by Dutch Mathematician, Nicolaas Govert de Brujin (of index fame) [12], who independently observed the Curry-Howard correspondence. Automath was a typed programming language providing inbuilt support for variable binding, substitution and application of judgemental equalities. This has been a common feature of proof-assistants since and is the defining features of a set of computational calculi known as *logical frameworks*. Users could define their own logics and types and no method of introducing inductive types was provided. Typechecking the program corresponded to the verification of the proof.

1.3.2 Martin-Löf Type Theory

In the early 1970s Per Martin-Löf, a Swedish logician and mathematician, aimed to exploit the Curry-Howard correspondence to provide what he deemed as a better foundation for mathematics [34]. Per Martin-Löf, a constructivist, asserted that in order to know of the existence of a mathematical object it must be directly constructed. To this end Martin-Löf went about producing a type system to produce an intuitionistic type theory for proving within higher-order logics. Intuitionistic logics were precisely the logics that type systems corresponded to which matched Martin-Löf's constructivist agenda. Martin-Löf further aimed for his type system to replace set theory as a foundation for doing all of mathematics. Martin-Löf's type theory was incredibly successful and pioneered the propositions-as-types approach to theorem proving which is an incredibly common approach taken in modern theorem provers. An overview of his theory will be presented in the Section 2.1

Martin-Löf type theory was deeply influential in many theorem provers and logical frameworks designed from then on including the Edinburgh Logical Framework [19] and Agda, the proof-assistant used within this thesis. An analysis of the various theorem provers and their relative strengths with take place in Section 4.1.

1.3.3 Agda

The first version of Agda was designed at Chalmers university in 1999 by Caterina Coquand [7] based on the ALF logical framework [33] ultimately derived from Martin-Löftype theory. The second version of Agda was later designed by Ulf Norell during his Ph.D. also at Chalmers University in 2007 [37] based on Zhaohui Luo's *Unified Theory of Dependent Types* (UTT) [32] derived from Martin-Löf type theory. Agda is implemented as a functional programming language with dependent types, and features an interactive mode in the emacs text editor, allowing users to interact with the Agda intepreter to develop proofs. Agda is a total language, allowing the user to write only programs that are provably terminating. This enables type checking (and therefore proof checking) to be decidable. The standard Agda backend, *MAlonzo*, is written in the functional programming language Haskell and is still being extended and developed in 2019.

1.4 Category Theory and Mathematical Logic

Alongside the development and exploration of proof assistants, type theory and the Curry-Howard correspondence the field of category theory was birthed and matured. Category theory is an offshoot of abstract algebra and algebraic topology and geometry that aims to provide a general account of mathematical structure. A category is a mathematical structure consisting of a collection of objects and arrows between these objects with a small set of further axioms to which the structure must adhere. This description allows many common mathematical structures to be considered as a category such as the category of sets where the objects are sets and the arrows functions or the category of groups with the objects being groups and the arrows being group homomorphisms. The definition of category is sufficiently abstract that the majority of mathematical structures can be considered as one. The abstract and encompassing definition of categories provides a vehicle for the transposition of mathematical ideas from one field to another. Category theory also provides a framework in which to understand ubiquitous and reoccurring themes in mathematics such as free objects, products and function spaces.

1.4.1 Beginnings

Category theory was initially invented by American mathematicians Samuel Eilenberg and Saunders Mac Lane in 1945 in their paper *General Theory of Natural Equivalences* [13] which defined categories, structure-preserving maps between categories, known as functors, and structure preserving maps between functors, known as natural transformations. Eilenberg initially applied these to the field of algebraic topology and geometry to make certain constructions simpler [14]. This line was followed by Grothendieck and Kan who added further concepts such as adjoint functors and limits [23], and further revolutionised algebraic topology [18]. A marked addition to the study of category theory was made by one of Eilenberg's Ph.D students William Lawvere. Lawvere's work throughout the 1960s began to analyse the relations between logic, foundations of mathematics and category theory. His Ph.D. thesis explored model theory and introduced Lawvere's theories, the categorical counterpart to equational theories [27]. Lawvere soon provided a categorical and structural account of axiomatic set theory with his *Elementary Theory of the Category of Sets* [28].

1.4.2 Diagonal Arguments and Cartesian Closed Categories

Another staple in Lawvere's series of papers relating to category theory and logic and a focal point of this thesis was his 1969 paper *Diagonal Arguments and Cartesian Closed Categories* [29]. Cartesian closed categories are a conceptual class of categories that have close relations to type theory and logic. Centered around having an internal concept of function space, cartesian closed categories arose from the beginnings of the study of topoi. Joachim Lambek in 1985, extended the Curry-Howard correspondence by providing a correspondence between simply-typed lambda calculi and cartesian closed categories [26]. In *Diagonal Arguments and Cartesian Closed Categories*, Lawvere showed how the paradoxes in mathematical logic and set theory from the early 20$^{\text{th}}$ century could be unified under a single theorem in the theory of cartesian closed categories. Lawvere's fixed point theorem, when intepreted within the category of sets which is cartesian closed, produces Cantor's theorem. Lawvere's theorem has the structure of a classical diagonal theorem and has been used since to explain the structure and appearances of other paradoxes and phenomena in mathematical logic

1.4.3 Beyond Lawvere

In 2003, Noson Yanofsy released a review paper, *A Universal Approach to Self-Referential Paradoxes, Incompleteness and Fixed Points* [46], on Lawvere's 1969 paper. Yanofsky's paper, in an attempt to make Lawvere's paper more accessible, reframed Lawvere's theorem outside of category theory. Yanofsky extended Lawvere's theorem to explain several of the logical paradoxes that have existed for thousands of years including the Liar's paradox and Grelling's paradox. Yanofsky also provided accounts of the theorem's applicability to phenomena in computability theorem such as a derivation of Rice's theorem and the Halting problem in automata theory and Kleene's recursion theorem. Yanofsky emphasises early in his article that Lawvere's fixed point theorem can be viewed as the limitations of logical and computational systems and how paradoxes can be viewed as the consequences of violating these limitations.

1.4.4 Contributions

It has been noted (see Section 3.3.2 and 4.3.1) that the components of the proof of Lawvere's fixed point theorem indicate a connection to the untyped λ-calculus. Given the connection between cartesian closed categories and the untyped λ-calculus and the syntactic resemblance it has been assumed by some that there is a relation between the untyped lambda calculus and Lawvere's fixed point theorem. Given the theorems applications within the theory of computable universes and Turing machines, recursion theory and the recursion theorem and Rice's theorem, it seems that this would be likely. In spite of this the relation between the λ-calculus and Lawvere's fixed point theorem has never been fully developed. The primary contribution of this thesis is an explicit account of this relation, primarily that, when interpreted in the context of cartesian closed categories corresponding to models of the untyped λ-calculus, Lawvere's fixed point theorem is equivalent to the first fixed point theorem for the untyped λ-calculus. In the evaluation of this thesis, previous attempts to formalise the relation are provided and examined. The proofs in this thesis are done in the proof-assistant Agda version 2.5.4.2. This decision was made so as to explore the relation between type theory and mathematical logic but has provided other benefits. Through formalisation, a greater appreciation for the particularities of Lawvere's theorem can be understood making it easier to find further relations. The other primary contribution of this thesis is a collection of formalised proofs within cartesian closed categories and of Lawvere's theorem itself. The code featured within this thesis is literate Agda and has been type checked for correctness. All mechanised proofs within this thesis can be found at `https://github.com/alessio-b-zak/thesis/tree/master/agda`.

Chapter 2

Technical Background

This section will outline the essential technical details to understand the primary contributions of this thesis. This section will aim to to provide a working knowledge of the theory behind and the usage of the Agda theorem prover. Through Agda, the underlying theory of categories will be explained to sufficiently understand Lawvere's fixed point theorem. As explained prior, theorem provers often work by utilising the Curry-Howard correspondence to embed a logical framework within the type system of a programming language. This is the approach taken by the Agda theorem prover which makes use of a type system similar to that of Martin-Löf Type theory, a type theory which provides a logical framework for intutionistic higher-order logic.

2.1 Martin-Löf Type Theory

The approach to theorem proving taken in this thesis is a type theoretic approach exploiting the Curry-Howard correspondence. This will be done in the dependent type theory designed by Martin-Löf. Martin-Löf's type theory (MLTT) was intended to be an entirely constructive foundation in which mathematics could be done. Just as, inline with the Curry-Howard correspondence, the intutionistic fragment of natural deduction has an interpretation within the simply-typed λ-calculus, MLTT has a logical interpretation as first-order intuitionistic predicate logic, by including dependent types in the theory. The theory is outlined below, a more detailed examination can be found notes derived from Martin-Löf's lectures [34] or Chapter One of the *Homotopy Type Theory* book [44]. Martin-Löf's lecture notes discuss the philosophical ramifications of his theory and advocate for its use as a foundation for all of mathematics.

Within MLTT, typing takes the form of a judgement. Judgements are statements in the metatheory of the type theory that can either be derived from the deductive rules of the type theory or introduced independently. The judgement that a term a has type A is written

$$a : A$$

The other primary judgement of MLTT is equality. Equality equates two terms in the sense that one can be replaced with the other freely within the theory. This allows new constructions to be introduced within the theory through naming and functions. The judgement that two terms, a and b are equal (at type A) is written

$$a \equiv b : A$$

Types are introduced by introducing equalities that define how to form and use the types within the context of the theory. In the section that follows, only a limited number of types will be introduced so as to understand how the type theory can be used in theorem proving. Before defining types, however, what it means to be a type must be introduced. A universe is a type whose elements are themselves types. The version of MLTT presented here postulates an infinite hierarchy of universes, U_n which is an element of the universe at a higher level U_{n+1} i.e.

$$U_0 : U_1 : U_2 \cdots$$

A type is then an inhabitant of some universe. One might desire some finite number of universes in which to work and indeed, Martin-Löf's first presentation of his type theory featured only one universe, U, which was an element of itself.

$$U : U$$

Thierry Coquand, in 1992, showed that a Russell-style paradox could be embedded within a type theory with a single impredicative universe [8]. This mirrors the inconsistencies that develop when trying to postulate a set of all sets in naive set theory.

Working in a world with a hierarchy of universes, a type is then defined to be an element of one of the postulated universes. New types can be defined within any of the postulated universe. Introducing new types into the type theory involves explaining how to create objects of that type and how to compute with objects of the type, done by adding more definitional equalities to the system. The most basic components of MLTT is the dependent function or Π-type. Π-types represent functions where the output type of a function can depend on the argument to the function. If A is a type (i.e. $A : U$ for some universe U) and $B : U$ the Π-type,

$$\prod_{(x:A)} B$$

which binds the variable x in B, represents the dependent function which takes an argument $x : A$ and returns an element of type B with the free variable x replaced with the argument to the function. A dependent function, $f : \Pi_{(x:A)}B$, can be introduced via a set of defining equations or via a λ-abstraction. Given an expression $M : B$ where B may contain the variable $x : A$, f can be defined as

$$f(x) :\equiv M \text{ for } x : A$$

Another way to introduce a dependent function is to introduce a λ which takes an identifier and an expression and produces a dependent function i.e.

$$\lambda\, x.M : \prod_{(x:A)} B(x)$$

Computing with $\prod$-types occurs through application and substitution. Given a dependent function, $f(x) \equiv M : \prod_{(x:A)} B(x)$, and a value $a : A$, a can be applied to f to obtain a value of type $B(a)$ written

$$f(a) : B(a)$$

The expression that the application yields is the result of replacing all instances of x in M with a. When B returns a constant type for input types the normal function arrow, →, can be used. The other types to be introduced in this section can be written as inductive types. Inductive types can be introduced by supplying constructors with build elements of the type from other types. Inductive types can be computed with using induction principles which describe how to compute with arbitrary structures of the type based on the constituent parts. Induction principles can be seen as an alternative to pattern matching, a more common feature of functional programming languages. A simple type to illustrate this is the product type, which is the type theoretic analogue of the cartesian product. Given two types $A : U_n$ and $B : U_n$ the product type, $A \times B : U$ can be formed. Given $a : A$ and $b : B$ an element of the pair type $A \times B$ can be constructed as

$$(a\, ,\ b)\ :\ A \times B$$

Before introducing the induction principle for product types it is worth dwelling on some subtleties of induction principles. Induction principles are judgemental equalities that describe how to compute with newly introduced data types. Induction principles often provide a separate defining function for each constructor for a type however this is *not* pattern matching. A separate defining equation for each constructor often makes logical sense for the type to have good computational properties but there is an element of choice to designing induction principles as shall be seen in the discussion of the identity type. The induction principle for products is a function with the type

$$\mathsf{ind}_{A\times B} : \prod_{C:A\times B\to U} (\prod_{(x:A)}\prod_{(y:B)} C((x,y))) \to \prod_{x:A\times B} C(x)$$

Intuitively, this type can be read as, given a function $C : A \times B \to U$ for some universe, U, and given a dependent function which takes two arguments and returns the dependent function applied to the pair of the two arguments, a dependent function for pairs can be produced. More concisely, a function for pairs can be produced from a function that takes two arguments. Intuitively this can be done by taking the function with two arguments, deconstructing the pair and applying each component of the pair in turn. As a defining equation this is:

$$\mathsf{ind}_{A\times B}\,(C, g, (a, b)) :\equiv g(a)(b)$$

Other types that are integral to the theorem-proving effort are Σ-types or dependent pairs. Dependent pairs are product types where the type of the second argument can depend on the first. Given a type $A : U$ and a function $B : A \to U$ the dependent product type $\Sigma_{(x:A)}B(x)$ can be formed. Given an element $a : A$ and an element $b : B(a)$ the dependent pair $(a, b) : \Sigma_{(x:A)}B(x)$. The induction principle for Σ-types is similar to the induction principle for products.

$$\mathsf{ind}_{\Sigma_{x:A}B(x)} : \prod_{C:\Sigma_{x:A}B(x)\to U} \left(\prod_{(x:A)}\prod_{(y:B)} C((x,y))\right) \to \prod_{p:\Sigma_{x:A}B(x)} C(x)$$

with the same defining equation

$$\mathsf{ind}_{\Sigma_{x:A}B(x)}\,(C, g, (a, b)) :\equiv g(a)(b)$$

The unit type, $\mathbf{1} : U$ is inhabited by only a single element $\star : \mathbf{1}$.

The induction principle for the unit type captures the notion that, to compute with the unit type, it is only necessary to consider the element $\star$.

$$\mathsf{ind}_{\mathbf{1}} : \prod_{C:\mathbf{1}\to U} C(\star) \to \prod_{x:\mathbf{1}} C(x)$$

This is defined as

$$\mathsf{ind}_{\mathbf{1}}(C, c, \star) :\equiv c$$

The coproduct type is the typed variant of the disjoint union from set theory. For two types $A : U$ and $B : U$ coproduct of A and B is $A + B : U$. There are two constructors for the coproduct type $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$ The induction principle for the coproducts requires a function to eliminate the inl case and a function to eliminate the inr. ind_{A+B} therefore has type

$$\mathsf{ind}_{A+B} : \prod_{(C:(A+B)\to U)} \left(\prod_{(a:A)} C(\mathsf{inl(a)})\right) \to \left(\prod_{(b:B)} C(\mathsf{inr}(b)))\right) \to \prod_{(x:A+B)} C(x)$$

With a defining equation for each constructor

$$\mathsf{ind}_{A+B}(C, c, d, \mathsf{inl}(a)) :\equiv c(a)$$
$$\mathsf{ind}_{A+B}(C, c, d, \mathsf{inr}(b)) :\equiv d(b)$$

The void type, $\mathbf{0} : U$, has no inhabitants and therefore there is no way to introduce it. The induction principle has no defining equations as there is no inhabitants with which to provide it. It is the type equivalent of the trivial function from the empty set.

$$\mathsf{ind}_{\mathbf{0}} : \prod_{(C:\mathbf{0}\to U)}\prod_{(x:\mathbf{0})} C(x)$$

The final and key component of MLTT is the identity type, a type used to prove that two terms are equal. This is known as propositional equality and is internal to the theory. For a given type, $A : U$, the identity type is a family $\mathsf{Id}_A : A \to A \to U$ for a given $a : A$, $b : A$ written $a =_A b$. Within MLTT there is a single inhabitant of the identity type for a given $a : A$ which can only be introduced if the second elements of A are definitionally equal known as refl_a. refl is a constructor of type

$$\mathsf{refl} : \prod_{a:A} (a =_A a)$$

There are different choices for induction principle for the identity type. Choosing whether to use both or only one is major decision in modern type theories. One of the induction principles for the identity type is known as Axiom J. Axiom J is a function of type

$$\mathsf{ind}_{=_A} : \prod_{(C:\Pi_{(x,y:A)}(x=_A y)\to U)} \left(\prod_{(x:A)} C(x,x,\mathsf{refl}_x) \right) \to \prod_{(x,y:A)} \prod_{p:x=_A y} C(x,y,p)$$

Axiom J is defined as

$$\mathsf{ind}_{=_A}(C, c, x, x, \mathsf{refl}_x) :\equiv c(x)$$

The basis upon which MLTT can be used as a foundation for mathematics is via the Curry-Howard correspondence in a particular fashion known as propositions-as-types.

2.1.1 Propositions-as-types

Propositions-as-types hinges on interpreting a proof of a proposition as an inhabitant of a corresponding type. The types presented in the previous section are the types with which higher order intutionistic logic can be interpreter. With types instead of sets, universal quantification over a type can be simulated using Π-types where the output type of the dependent function is the proposition being quantified over. Existential quantification can be interpreted as Σ types where the second argument is the proposition being quantified over and the first argument is the object that satisfies said proposition. Implication is a non-dependent function type, logical conjunction corresponds to product types, coproduct types to logical disjunction. Truth is inhabitation of the unit type and falsity as an inhabitant of the void type. The void type has no inhabitants and so any such inhabitant would constitute a proof of the inconsistency of the logic the type system represents. With this definition of falsity, negation is a function that takes a type (proposition) and produces and inhabitant of the void type, something that should not be possible. The intutionistic, constructive component of this logic comes from the fact that to prove a position an element of a type *must* be constructed. A proof of $A \wedge B$ consists of providing a proof of A *and* providing a proof of B. This constructive approach to logic weakens the deductive framework compared to classical logic.

The propositions-as-types interpretation of logic is proof-relevant, proving, within the type theory, that two things are equal uses the identity type. An inhabitant of the identity type is an object within the type-theory presenting a equality between two things.

2.2 Agda

Agda is a dependently typed functional language in which theorem can be done. Agda can be used as a proof assistant through the lens of the Curry-Howard correspondence given it's type system. Agda uses Haskell like syntax and its underlying type theory is based of that of Per Martin-Löf. The following describes features of Agda available in version 2.52, the version employed in this thesis. Agda uses a predicative hierarchy of universes, Set, indexed by a natural-number like type level. Set can be viewed as a type-family returning a universe.

A term t of a given type, A can be introduced and named in Agda as follows

```
name : A
name = t
```

where name : A indicates that the identifier name has type A and the second line assigns to name the value to the right hand sign of the equals sign, t. Unlike other typed programming languages, the types of identifiers cannot be elided and inferred by the compiler. In the presence of dependent types, the problem of type inference becomes undecidable and therefore it is preferable to explicitly annotate them for all identifiers. Agda provides a limited number of the constructions provided in Martin-Löf type theory and instead provides a method for defining inductive datatypes allowing inductive types such as dependent pairs, product and coproduct types to be defined. One of the constructions provided by are Pi types. A Pi type can be introduced with an identifier, foo, as follows

```
foo : (x : A) → B
foo x = M
```

where, in the type signature, x is an identifier of type A which may appear in the output type B. The x to the lefthand side of the equals sign in the function definition binds the identifier x as an argument to the function of type A which may be used in the term M of type B. Arguments to a function need not have the same names in the type and definition of the function e.g.

```
foo' : (y : A) → B
foo' x = M
```

is a valid definition.

As in MLTT, functions can also be introduced using a λ-abstraction. A downside of this is that the arguments of a λ-abstraction can not be pattern matched on in the body of a function.

Inductive data types can be introduced as follows.

```
data InductiveType (Parameter : Set) : (Index : ℕ) → Set where
  Constructor1 : InductiveType Parameter 0
  Constructor2 : InductiveType Parameter 1
```

The data keyword is followed by the name of the inductive data type. Before the colon in the first line are the parameters to the type. Parameters to a type appear as is in constructors to the type. They indicate that the type behaves parametrically with respect to them. This is why in the constructors Parameter appears as is. Indices, on the other hand, appear after the colon. Indices can change the shape of the type depending on the constructor. In the example given the index is an inhabitant of the natural number type and the first constructor dictates the inhabitant of the natural number type is 0 and in the second constructor that the inhabitant is 1. The final term before the where statement is the output universe of the type i.e. to which Set the type belongs. The constructors of the Inductive data type are the possible inhabitants of the type. When pattern matching onto an inductive data type, information is gained about the type based on the index corresponding to the constructors produced by the pattern match.

Many constructions within Agda are common enough that they are desirable at all levels of universe. Agda does not have cumulativity so, to assist universe generic programming, universe polymorphism can be used.

A more useful example to consider is the sized list type, or vector

```
data Vec {a : Level} (A : Set a) : ℕ → Set a where
  [] : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)
```

The curly braces in the type definition indicate implicit arguments, arguments Agda will try and infer from other arguments. Vec features set polymorphism, parameters and indices. The two constructors for Vec, are indexed by their size. The empty list has size zero to indicate its emptiness. _::_ takes an element of type a and a Vector of size n and appends the singleton to the beginning creating a vector of size suc n. When pattern matching onto a Vec, information is gained about the inductive argument to the type i.e for the empty list has size zero and in the inductive case that the list has size suc n for some n. The remaining important types of Martin-Löf type theory can now be introduced as inductive types beginning with Sigma types.

```
data Σ {n m : Level} (A : Set n) (B : A → Set m) : Set (m ⊔ n) where
  _,_ : (a : A) → (B a) → Σ A B
```

Names can be defined with underscores which are taken as positional arguments. The output universe for sigma must be the maximum of the levels of the first and second components of the sigma type which can be done using ⊔ which takes two Levels and returns the larger. The product type is defined similarly.

```
data _×_ {m n : Level} (A : Set m) (B : Set n) : Set (m ⊔ n) where
  _,,_ : A → B → A × B
```

and coproducts

```
data _⊎_ {m n} (A : Set m) (B : Set n) : Set (m ⊔ n) where
  inl : A → A ⊎ B
  inr : B → A ⊎ B
```

and the unit type

```
data ⊤ : Set where
  tt : ⊤
```

and the void type

```
data ⊥ : Set where
```

Negation takes the appropriate definition within the intuitionistic setting

```
¬_ : ∀ {l} → Set l → Set l
¬ P = P → ⊥
```

Instead of introducing induction and recursion principles for inductive data types, Agda instead opts for deep pattern matching whereby an inductive datatype can be expanded into its constituent components. This is justified by the fact that all inductive data types possible of being defined within (normal) Agda correspond to W-types i.e. types that admit a well-founded induction principle. An example of pattern matching can be used to define a projection out of product types.

```
projl : {m n : Level} → {A : Set m} → {B : Set n} → A × B → A
projl (x ,, x₁) = x
```

The final type needed before propositions-as-types can be employed is the equality type which is defined as follows

```
data _≡_ {m : Level} {A : Set m} (x : A) : A → Set m where
  refl : x ≡ x
```

For each inhabitant of A, x, there is unique inhabitant of the equality type parameterised by x which is when the second indexed argument to the equality type normalises to the first parameter. This restriction can only be made if the second argument is an index so we are able to restrict its shape.

Proving theorems with no additional features in the language would prove difficult. Mathematical structures would be a pain to define in the standard inductive style as often they will often consist of a single constructor with arguments that depend on each other. To assist with these types of structure record types exist. Records are extensions of Σ-types which have named fields to assist with referring to the individual components, an illustrative example is the definition of a monoid, $(S, \bullet, e)$, within Agda

```
record Monoid (a : Level) : Set (lsuc a) where
  field
    S : Set a
    _•_ : S → S → S
    e : S
  field
    •-assoc : (a b c : S) → ((a • b) • c) ≡ (a • (b • c))
    e-left-neutral : {a : S} → e • a ≡ a
    e-right-neutral : {a : S} → a • e ≡ a
```

where lsuc is the equivalent of suc but for levels. This is required due to • which forces the implicit constructor for the type to be of a higher sort than S itself, due to the predicativity of the underlying type theory. The identifiers to the left hand side of the colon under the field keyword in the above definitions define projections out of a Monoid object. The first three fields correspond to the elements of a tuple representing a monoid, the underlying set, binary operation and identity element.

Another limitation within the currently outlined framework with respect to proving theorem is the definition of the equality type. In a world where where Agda used induction principles instead of pattern matching, Axiom J, would, in some sense, not be strong enough to be useful when proving a significant number of theorems. The problem, being addressed can be introduced by considering a homomorphism type for the above definition of monoid

```
record MonHom {L L'} (M : Monoid L) (M' : Monoid L') : Set ( L ⊔ L') where
field
  f : S → S'
  e-preserved : f e ≡ e'
  •-preserved : (X Y : S) → (f (X • Y)) ≡ (f X •' f Y)
```

where the fields of the second monoid are postfixed with an apostrophe. Consider showing two monoids are propositionally equal. For records, this amounts to showing that its fields are equal i.e that the underlying functions are the same but *also* the proofs of preservation of identity and operation are the same. The proof-relevant nature of the underlying type theory enables two different proofs to be distinguished by their normal form. It would be unreasonable and beside-the-point to demand this when equating two monoids but it is *required* when equating using propositional equality. The approach taken in standard Agda is to employ an additional axiom on the identity type known as Streicher's Axiom K [41]. Axiom K is an axiom that enables it to be proven that all inhabitants of the identity type are refl. Introducing Axiom K into MLTT turns the theory into a proof-irrelevant one. The proofs for both structures can be reduced to refl and then equated propositionally and all that remains is showing equality of functions. Introducing Axiom K is not without its downsides however. Recent advancements in type theory [44] show that there are significant advantages to working within a proof-relevant setting which are inconsistent with Axiom K. These are discussed in Section 4.2.2.

The last limitation that must be addressed is pertinent to the goal of formalising category theory within type theory. Returning again to considering equality of monoid homomorphisms, by employing Axiom K, showing equality of monoid homomorphisms amounts solely to showing equality of functions. Within set theory, it is common to equate functions that are pointwise equal

$$(\forall x\ f(x) = g(x)) \implies f = g$$

This notion of equality ignores, for better or worse, the computational content of the individual functions. It does not matter if functions are operationally different but only that they are functionally different. This principle is not derivable within standard MLTT and therefore, if it is going to be used, it must be postulated as an axiom. For many, a distinct advantage of computer-aided theorem proving using types is its intrinsically constructive interpretation and therefore it is common to avoid axioms such as function extensionality within the type theory.

A common solution within the type theory of unmodified Agda is the setoid approach.

2.2.1 Setoids

A setoid is a set or type alongside an equivalence relation

```
record Setoid c l : Set (suc (c ⊔ l)) where
  field
    Carrier : Set c
    _≈_     : Rel Carrier l
    isEquivalence : IsEquivalence _≈_
```

Where equivalence relations are defined appropriately

```
record IsEquivalence {a l} {A : Set a}
  (_≈_ : Rel A l) : Set (a ⊔ l) where
  field
    refl  : Reflexive _≈_
    sym   : Symmetric _≈_
    trans : Transitive _≈_
```

Setoids can be used to work with and prove properties of extensional equalities without introducing a new axiom. This is pertinent to category theory as, often, structures like monoids and monoid homorphisms are the subject of examination and a useful notion of equality is essential for making progress. Proofs with respect to the equivalence relation can be done in a largely similar way to using propositional equality using the properties of reflexivity, transitivity and symmetry of the equivalence relation. Other situations in which it is preferable to use setoids is when working with algebraic structures where the underlying equality is not truly propositional equality for example the monoid consisting of the rational numbers under addition. Commonly the rational numbers are defined as a pair of integers however equality on fractions usually consists of equality of their reduced forms, generating an equivalence class of fractions. In general quotiented structures are not readily available within standard MLTT.

The primary limitation when opting for setoid equality over propositionally equality is the inability to employ indecernability of identicals or congruence a natural consequence of Axiom J. This is an advantage of propositionally equality since this property must be proven for each equivalence relation separately, in some circumstances adding on a significant amount of work. Recent advancements in type theory have produced a type theory in which function extensionality can be derived and has computational content, discussed in Section 4.2.2

2.3 Category Theory

2.3.1 Basic Definitions

Category theory is a unifying field of mathematics that examines abstract structure. A category, **C**, is a mathematical structure containing a class objects and a class of morphisms or directed relations between said objects. Many expositions of category theory are somewhat vague surrounding exactly mathematical structures that objects and arrows are. If objects and arrows are *sets* of things, swathes of mathematical objects are inaccessible to category theory because they are too large due to Russell's paradox style problems such as a category of all sets or a category of all categories. This was a main motivation behind Martin-Löf's theory of types by defining categories and the objects and arrows of categories as types. By paying closer attention to what is in the metatheory versus internal to theory i.e. set membership versus a typing judgement and by paying closer attention to the predicativity of the system in question, *larger* objects can safely be embedded in the system. In the informal presentation of category theory, as is prevalent in the literature, no firm foundations will be provided so as to aid in understanding. This will be followed by a formal presentation of the constructions within Agda. To repeat, a category is a collection of objects and a collection of arrows between objects.

$$\begin{aligned} \text{Obj} &: A \\ \text{Arr} &: A \to B \end{aligned}$$

In Agda, categories can be constructed in with relative simplicity. The initial concepts introduced here are taken from `cats` [30], a Category Theory library in Agda by Jannis Limperg. `lo`, `la` and `l≈` are used in the following section as variables of type Level.

Categories can be introduced as a record parameterised by the level of their objects, arrows and type of morphism equality respectively

```
record Category lo la l≈ : Set (suc (lo ⊔ la ⊔ l≈)) where
  field
    Obj   : Set lo
    _⇒_ : Obj → Obj → Set la
```

Categories are typed at a level above the largest of objects, arrows and equalities in order to present equality on morphisms as relations on types as per the Agda standard library. In addition to objects and arrows, there exists a binary operation on morphisms known as composition which takes two morphisms, $f : A \to B$ and $g : B \to C$ and produces a third morphism $g \circ f : A \to C$. Furthermore, for each object, A, in the category, there exists an identity arrow $id_A : A \to A$.

$$\begin{aligned} &\text{Identity} : \forall A \in \text{Obj}(\mathbf{C})\ \exists\ id : A \to A \\ &\text{Composition} : \text{Given } f : A \to B \text{ and } g : B \to C\, \exists\ g \circ f : A \to C \end{aligned}$$

In Agda, identity arrows and composition take their obvious definitions. The identity arrow provides a distinguished morphism for each object implicitly and composition is a function that takes two morphisms of the correct shape and returns the appropriate morphism

```
field
  id : {O : Obj} → O ⇒ O
  _∘_  : ∀ {A B C} → B ⇒ C → A ⇒ B → A ⇒ C
```

The current operations defined for a category must adhere to a few more axioms, namely

$$\begin{aligned} &\text{Neutrality of identity} : \forall\, g : A \to B \quad g \circ id_A = g \text{ and } \forall f : C \to D \quad id_D \circ f = f \\ &\text{Associativity of Composition} : \forall\, f\,, g\,,\, h \quad ((h \circ g) \circ f) = (h \circ (g \circ f)) \end{aligned}$$

As mentioned in the previous section, when codifying equalities on morphisms, such as associativity of composition and the neutrality of identity within Agda, it is often not practical to use propositional equality. It is common to work with categories with which the morphisms are functions between types equipped with additional structure. To work with these in standard Agda, either extensionality must be postulated or setoids must be used. There are other factors involved with the decision between equality on morphsims being propositional or setoid such as performance and ease-of-use, some of which is discussed in.

```
field
  _≈_ : ∀ {A B} → Rel (A ⇒ B) l≈
  equiv : ∀ {A B} → IsEquivalence (_≈_ {A} {B})
```

The IsEquivalence function establishes the appropriate proofs of reflexivity, transitivity and symmetry. With the notion of equality of morphisms in place it is possible to state the properties of composition and identity

```
field
  id-r : ∀ {A B} {f : A ⇒ B} → f ∘ id ≈ f
  id-l : ∀ {A B} {f : A ⇒ B} → id ∘ f ≈ f

  assoc : ∀ {A B C D} {f : C ⇒ D} {g : B ⇒ C} {h : A ⇒ B}
    → (f ∘ g) ∘ h ≈ f ∘ (g ∘ h)

  ∘-resp : ∀ {A B C}
    → (_∘_ {A} {B} {C}) Preserves₂ _≈_ ⟶ _≈_ ⟶ _≈_
```

The field ∘-resp is the result of the aforementioned lack of congruence for equivalence relations. Preserves2 indicates that composition is congruent in both of its arguments. This allows us to target individual compositions in a large categorical term to apply an equality. This is given for free when using propositional equality as functions are unable to distinguish terms with the same normal form. This can be seen as one of the downsides to using equivalence relations as congruence must be proven for every each individual equivalence relation.

There are many examples of categories throughout mathematics and computing. The category of groups, **Group**, has as its objects groups and its morphisms group homomorphisms. The category of sets, **Set**, has as its objects sets and its morphsisms total functions. A category of types will be introduced in . . . which will be used to explore an application of Lawvere's theorem

2.3.2 Universal Constructions

A key idea of category theory are universal constructions. Universal constructions are common patterns that occur throughout mathematics that aim to capture the essence of these patterns at the categorical level. The universal constructions presented here are those that will be of use within the thesis.

Terminal Objects

Terminal objects are constructions that capture the minimal structure required to be an object within a category. They often correspond to the trivial examples of objects within the category. A terminal object of a category **C** is an object, T, such that, for all other objects, A in the category, there exists a unique arrow $!_A : A \to T$. This can be shown as a diagram where the dashed line indicates uniqueness.

$$\begin{tikzcd}
A \arrow[d, dashed, "!_A"] \\
T
\end{tikzcd}$$

As is common with universal constructions, terminal objects in categories are unique up to unique isomorphism. Examples of terminal objects in common categories include any singleton in **Set** and the one element group in **Group** Formalising universal constructions within Agda requires the notion of unique arrow

```
IsUniqueSuchThat : ∀ {lp A B}
  → (A ⇒ B → Set lp)
  → A ⇒ B
  → Set (la ⊔ l≈ ⊔ lp)
IsUniqueSuchThat P f = ∀ {g} → P g → f ≈ g

IsUnique : ∀ {A B} → A ⇒ B → Set (la ⊔ l≈)
IsUnique {A} {B} = IsUniqueSuchThat {A = A} {B} (λ _ → ⊤)
```

Uniqueness is often given with respect to a property (hence universal properties). In Agda this amounts to formulating the property as type parameterised by the property and an arrow satisfying the property. The type encodes a function which, given any other arrow satisfying the property expresses equality to the parameterised arrow. Using this a general unique arrow can be encoded using a trivial function that always returns the unit. Universal properties can now be given as an object, a proposition and a proof of uniqueness

```
record ∃!′ {lp A B} (P : A ⇒ B → Set lp) : Set (la ⊔ l≈ ⊔ lp) where
  field
    arr : A ⇒ B
    prop : P arr
    unique : IsUniqueSuchThat P arr
```

Agda has support for custom syntax directives which can be used to create a universal mapping type postulating the existence of a unique arrow. Below is an example of defining products using this where ∃!, desugars to the universal property type above.

```
IsTerminal : Obj → Set (lo ⊔ la ⊔ l≈)
IsTerminal One = ∀ X → ∃! X One
```

A category having a terminal object can now be encoded as a proposition which takes a category and provides an object alongside a proof that it is terminal

```
record HasTerminal {lo la l≈} (Cat : Category lo la l≈)
  : Set (lo ⊔ la ⊔ l≈) where
  field
    One : Obj
    isTerminal : IsTerminal One
```

Lawvere's fixed point theorem is a theorem about cartesian closed categories. Cartesian closed categories are categories with three specific universal properties, a terminal object, binary products and exponentials. Having a terminal object and binary products is also called having finite products. Therefore the definition of a CCC in Agda is appropriately.

```
record IsCCC {lo la l≈} (Cat : Category lo la l≈)
  : Set (lo ⊔ la ⊔ l≈) where
  field
    hasFiniteProducts : HasFiniteProducts Cat
    hasExponentials : HasExponentials Cat
```

Products

Products exemplify a common construction in categories of combining the structure of two objects (in some canonical way) within the category to produce an object of the same category. In more concrete terms the product of two objects A and B in the category **C** is an triple $(A \times B, \pi_1, \pi_2)$ where for all other objects C in **C** with projections $f : C \to A$ and $g : C \to B$ the unique arrow $\langle f, g \rangle : C \to A \times B$ can be formed such that the following diagram commutes:

$$\begin{tikzcd}
 & C \arrow[ld, "f"'] \arrow[d, dashed, "\langle f,g \rangle"] \arrow[rd, "g"] & \\
A & A \times B \arrow[l, "\pi_2"] \arrow[r, "\pi_1"'] & B
\end{tikzcd}$$

where the dashed arrow indicates uniqueness. This can be extended to n-ary products in the obvious way.

As with terminal objects, products are unique up to unique isomorphism. Examples of products within familiar categories include the cartesian product × in **Set**, defined as the set of all tuples of elements from two separate sets. If products can be formed for every finite set of objects in a category it is said to be cartesian.

Products have a slightly more involved definition than terminal objects. Beginning with the uniqueness principle for products

```
IsProduct : ∀ {li} {I : Set li} (O : I → Obj) P → (∀ i → P ⇒ O i)
  → Set (lo ⊔ la ⊔ l≈ ⊔ li)
IsProduct O P proj
  = ∀ {X} (x : ∀ i → X ⇒ O i) → ∃![ u ] (∀ i → x i ≈ proj i ∘ u)
```

IsProduct takes an indexing function, a product object, and some projections out of the product into the components of the indexing category. IsProduct returns a function type which, upon being supplied a set of projections from an object to the indexing set, returns a unique arrow from the object to the previously supplied product object satisfying the commuting diagrams for the product. A product object for a given indexed family of objects O can be defined as the product object itself, prod, the projections out of the product object, proj, and the proof of uniqueness, isProduct.

```
record Product {li} {I : Set li} (O : I → Obj) : Set (lo ⊔ la ⊔ l≈ ⊔ li) where
  field
    prod : Obj
    proj : ∀ i → prod ⇒ O i
    isProduct : IsProduct O prod proj
```

A binary product is a product where the function that indexes the family of objects is the boolean elimination function

```
Bool-elim : ∀ {a} {A : Bool → Set a} → A true → A false → (i : Bool) → A i
Bool-elim x y true = x
Bool-elim x y false = y


BinaryProduct : Obj → Obj → Set (lo ⊔ la ⊔ l≈)
BinaryProduct A B = Product (Bool-elim A B)
```

A category containing binary products can be encoded as a category equipped with a operation that, for every pair of objects, A and B, produces the product object for the pair.

```
record HasBinaryProducts {lo la l≈} (C : Category lo la l≈)
  : Set (lo ⊔ la ⊔ l≈)
  where
  field
    _×′_ : ∀ A B → BinaryProduct A B
```

For notational convenience a function that returns the object within the category from a particular product object is useful.

```
_×_ : Obj → Obj → Obj
```

Also useful are the projections out of the product

```
projr : ∀ {A B} → A × B ⇒ B
projl : ∀ {A B} → A × B ⇒ A
```

and the methods of forming the unique arrows

```
⟨_,_⟩ : ∀ {A B Z} → Z ⇒ A → Z ⇒ B → Z ⇒ A × B
⟨_×_⟩ : ∀ {A B A′ B′} → A ⇒ A′ → B ⇒ B′ → A × B ⇒ A′ × B′
```

Examples of products include the direct-product of groups in **Group** and the cartesian product of sets in **Set**. A category with a terminal object and products is said to have finite products and is a cartesian category.

Exponentials

Exponential objects are universal constructions that capture the notion of function spaces or higher order objects. The exponential, B^A, indicates the mappings from the object A to B. This is paired with the morphism $eval : B^A \times A \to B$ such that for any object Z and morphism $f : Z \times A \to B$ there exists a unique morphism $\tilde{f} : Z \to B^A$ such that the following diagram commutes:

$$\begin{tikzcd}
B^A \times A \arrow[r, "eval"] & C \\
A \times B \arrow[u, "\langle\, \hat{f} \times id \rangle"] \arrow[ur, "f"'] &
\end{tikzcd}$$

$\tilde{f}$, or transposition, can be thought of as currying in the functional programming sense, taking a function in multiple arguments to a sequence of functions in one argument.

In Agda, an exponential object for objects B and C consist of an object

```
record Exp (B C : Obj) : Set (lo ⊔ la ⊔ l≈) where
  field
    C^B : Obj
```

the evaluation map

```
    eval : C^B × B ⇒ C
```

and the uniqueness principle for products

```
    curry′ : ∀ {A} (f : A × B ⇒ C)
      → ∃![ f’ ∈ A ⇒ C^B ] (eval ∘ ⟨ f’ × id ⟩ ≈ f)
```

For ease of use the function producing the unique arrow can be extracted.

```
    curry : ∀ {A} → A × B ⇒ C → A ⇒ C^B
```

A category that has exponentials is one that where the exponential object can be formed for every pair of objects

```
record HasExponentials {lo la l≈} (Cat : Category lo la l≈) : Set (lo ⊔ la ⊔ l≈)
  where
```

```
field
  _⇝′_ : ∀ B C → Exp B C
```

A convenient exponential operation can be defined that extracts the object

```
_⇝_ : Obj → Obj → Obj
```

With a generic evaluation map

```
eval : ∀ {B C} → (B ⇝ C) × B ⇒ C
```

and curry and uncurry as isomorphisms

```
curry : ∀ {A B C} → A × B ⇒ C → A ⇒ B ⇝ C
uncurry : ∀ {A B C} → A ⇒ B ⇝ C → A × B ⇒ C
```

In **Set**, the exponential object for two sets is the set of all functions between them.

Chapter 3

Project Execution

In this section the main contributions of this thesis will be presented. Working towards a formalised version of Lawvere's fixed point theorem, definitions and properties of points to objects will be explored. Following this, a detailed proof of Lawvere's theorem will be presented in Agda. This will then be applied to two separate domains: a category of types, and models of the untyped λ-calculus. Within the category of types an analog of Cantor's theorem will be derived after establishing its cartesian closedness. After, the interplay between cartesian closed categories and models of the untyped λ-calculus will be examined and a novel result will be derived.

3.1 Points

Lawvere's fixed point theorem makes use of the notion of points and surrounding definitions. The necessary definitions will be presented in Agda. Points are a categorical abstraction that generalise the notion of elements of a set. A point is an arrow from the terminal object to any other object

```
Point : Obj → Set la
Point X = One ⇒ X
```

Given the name of Lawvere's theorem it makes sense to formalise the notion of a fixed point categorically

```
IsFixedPoint : {B : Obj} → (f : B ⇒ B) → (s : Point B) → Set l=
IsFixedPoint f s = f ∘ s ≈ s
```

A fixed point of a morphism is a point that is idempotent under composition to the right with the morphism. As is common within the `cats` library, this is then wrapped up in a record as a sigma type to express that a given function has a fixed point

```
record HasFixedPoint {B : Obj} (f : B ⇒ B) : Set (lo ⊔ la ⊔ l=) where
  field
    X : Point B
    isFixedPoint : IsFixedPoint f X
```

Now, the fixed point property, something which features in Lawvere's theorem, can be formalised as a predicate on an object in a category expressing that all endomorphisms on the object have a fixed point.

```
FixedPointProperty : Obj → Set (lo ⊔ la ⊔ l=)
FixedPointProperty B = ∀ f → HasFixedPoint {B} f
```

Another concept that needs to be understood is the notion of point surjectivity which itself requires some machinery. First, the notion of a solved equation with points is required

```
IsSolution : {A B : Obj} → (f : A ⇒ B) → (a : Point A) → (b : Point B) → Set l=
IsSolution f a b = f ∘ a ≈ b
```

Point surjectivity expresses the notion that given a point to B, $(b : 1 \to B)$, and a morphism $f : A \to B$, we can produce a point to A, $(a : 1 \to A)$, that satisfies the equation $f \circ a = b$.

Packaging this up into a sigma type which, given a morphism, `f`, from an object `A` to an object `B`, and a point to `B`, contains a point to `A` and a proof that the point to `A` constitutes a solution to the triple.

```
record HasSolution {A B : Obj} (f : A ⇒ B) (b : Point B) : Set (lo ⊔ la ⊔ l=) where
  field
    X : Point A
    isSolution : IsSolution f X b
```

A point surjective morphism is a function for which for every point to `B` there exists a solution

```
IsPointSurjective : {A B : Obj} → (f : A ⇒ B) → Set (lo ⊔ la ⊔ l=)
IsPointSurjective f = ∀ b → HasSolution f b
```

The formulation of the point surjectivity used in the theorem is as a record confirming the existence of a point surjective function between two objects

```
record PointSurjective (A : Obj) (B : Obj) : Set (lo ⊔ la ⊔ l=) where
  field
    arr : (A ⇒ B)
    isPointSurjective : IsPointSurjective arr
```

3.2 Lawvere's Fixed Point Theorem

It is now possible to state Lawvere's theorem precisely, working within a Cartesian Closed Category.

```
lawvere : {A B : Obj} → PointSurjective A (A ⇝ B) → FixedPointProperty B
```

Or mathematically that, in a cartesian closed category, if there exists a point-surjective function from some object A to the exponential object, B^A then every endomorphism on B has a fixed point.

The proof of the theorem will be developed line by line. The first step is to pattern match on the arguments to the proof and bring in the constructors for the output type.

```
lawvere record { arr = ϕ ;
                 isPointSurjective = isPointSurjective } f =
in record { X = {!!} ; isFixedPoint = {!!} }
```

The first argument to the proof is the point-surjective morphism constituting the underlying morphism and the proof of point-surjectivity, and the second argument, `f` is the endomorphism on `B`. The output requires a point to `B` alongside a proof that it is a fixed point of `f`.

To produce a fixed-point, the goal is to create a morphism, `g`, from `A` to `B` and then exploit the point-surjective morphism to find a point to `A`. With the correctly chosen `g` the composition of this point to A with `g` will be a fixed-point. `g` will be constructed such that is in some way "self-replicating".

```
let g = (f ∘ eval ∘ ⟨ ϕ × id ⟩ ∘ δ )
```

where δ is the unit of the diagonal-product adjunction and can defined simply as

```
δ : {A : Obj} → A ⇒ A × A
δ = ⟨ id , id ⟩
```

Categorically, `g` represents the following diagram

$$\begin{tikzcd}
B & B \arrow[l, "f"'] & B^A \times A \arrow[l, "eval"'] & A \times A \arrow[l, "\langle \phi \, , id \rangle"'] & A \arrow[l, "\delta"']
\end{tikzcd}$$

In order to push `g` morphism back through the point-surjective morphism it needs to be turned into a point to the exponential object. This can be achieved via two isomorphisms, $1 \times A \cong A$ and $\hom(A \times B\,,\,C) \cong \hom(A\,,\,C^B)$. The directions of these isomorphisms used in the proof are as follows

```
collapseToOne : ∀ {A B} → (One × A ⇒ B) → (A ⇒ B)
curry : ∀ {A B C} → A × B ⇒ C → A ⇒ B ⇝ C
```

By applying the first isomorphism followed by the second, a point to B^A can be acquired

```
g' = (curry (extendToOne (f ∘ eval ∘ ⟨ ϕ × id ⟩ ∘ δ )))
```

The point-surjectivity of ϕ can now be used to acquire the associated point to A with g'

```
ps = isPointSurjective g'
u = (HasSolution.X ps)
```

The fixed point construction can now be achieved by composing ϕ with u twice to obtain a point to B. After composing once with u a point to B^A is obtained. This must be pushed through the aforementioned isomorphisms to get a morphism, $A \to B$, to compose with u again.

```
ϕ∘u = ( collapseToOne (uncurry (ϕ ∘ u)))
fixedPoint = ϕ∘u ∘ u
```

Now it must be shown that f ∘ fixedPoint ≈ fixedPoint. This proof will be developed using equational reasoning. The proof starts with the word begin and the left-hand side of the equality, with expressions separated by equalities on morphisms, put inside ≈⟨ ⟩, and ends with the right-hand side of the equality followed by ∎. In the case of the required proof

```
proof
  = begin
      fixedPoint
    ≈⟨ {!!} ⟩
      f ∘ fixedPoint
    ∎
```

The first transformation in the proof is to use the point-surjectivity of ϕ to expand the ϕ ∘ u within the definition of fixedPoint to g'.

The proof of point-surjectivity is extracted as follows

```
ps-proof = HasSolution.isSolution ps
```

This cannot be used directly due to the application of curry and extendToOne to the expression. The usage of equivalence relations means that congruence must be proved separately for every function on morphisms. These two proofs have the following types but the proofs are elided due to unnecessary complexity.

```
uncurry-resp : ∀ {A B D} {f g : A ⇒ B ⇝ D} → f ≈ g → uncurry f ≈ uncurry g
collapseToOne-resp : ∀ {A B} {u v : One × A ⇒ B} → (u ≈ v) → (collapseToOne u) ≈ (collapseToOne v)
```

To make ps-proof work within the nested function applications they are wrapped in the two proofs of congruence necessary

```
col-unc-ps-proof = collapseToOne-resp ( uncurry-resp ps-proof )
```

This can then be used by targeting the lefthand morphism of the outermost composition to change this to g'. This is done using ∘-resp-l which allows a proof to be applied to change the lefthand side of a morphism composition.

```
  fixedPoint
≈⟨ ∘-resp-l col-unc-ps-proof ⟩
  (collapseToOne (uncurry g')) ∘ u
```

The next transformation is accomplished by utilising that curry is an isomorphism with respect to uncurry, and that collapseToOne is an isomorphism with respect to extendToOne. With this g' ∘ u is obtained, followed by expanding g',

```
  (collapseToOne (uncurry g')) ∘ u
≈⟨ ∘-resp-l (∘-resp-l uncurry∘curry) ⟩
  (collapseToOne (extendToOne ( f ∘ eval ∘ ⟨ ϕ × id ⟩ ∘ δ))) ∘ u
≈⟨ ∘-resp-l (collapseExtendIso) ⟩
```

```
  (f ∘ (eval ∘ (⟨ ϕ × id ⟩ ∘ δ))) ∘ u
```

Before being able to manipulate this expression the expression must be reassociated. This is particularly tedious.

```
  (f ∘ (eval ∘ (⟨ ϕ × id ⟩ ∘ δ))) ∘ u
≈⟨ ∘-resp-l unassoc ⟩
  ((f ∘ eval) ∘ ((⟨ ϕ × id ⟩) ∘ δ)) ∘ u
≈⟨ ∘-resp-l unassoc ⟩
  (((f ∘ eval) ∘ (⟨ ϕ × id ⟩)) ∘ δ) ∘ u
≈⟨ ∘-resp-l (∘-resp-l assoc) ⟩
  ( (f ∘ (eval ∘ ⟨ ϕ × id ⟩)) ∘ δ) ∘ u
≈⟨ assoc ⟩
  ( f ∘ eval ∘ ⟨ ϕ × id ⟩) ∘ (δ ∘ u)
```

Once this has been achieved, definitions can be expanded and applied. Intuitively, it can be seen that the precomposition of a morphism by an arrow to a product object can fused to push the precomposed morphism into each branch of the product object.

```
  ( f ∘ eval ∘ ⟨ ϕ × id ⟩) ∘ (δ ∘ u)
≈⟨ ≈.refl ⟩
  ( f ∘ eval ∘ ⟨ ϕ × id ⟩) ∘ ⟨ id , id ⟩ ∘ u
≈⟨ ∘-resp-r ⟨,⟩-∘ ⟩
  ( f ∘ eval ∘ ⟨ ϕ × id ⟩) ∘ (⟨ (id ∘ u) , (id ∘ u) ⟩)
≈⟨ ∘-resp-r (⟨,⟩-resp id-l id-l) ⟩
  ( f ∘ eval ∘ ⟨ ϕ × id ⟩) ∘ ⟨ u , u ⟩
≈⟨ ∘-resp-l unassoc ⟩
  ((f ∘ eval) ∘ ⟨ ϕ × id ⟩) ∘ ⟨ u , u ⟩
≈⟨ assoc ⟩
  (f ∘ eval) ∘ (⟨ ϕ × id ⟩ ∘ ⟨ u , u ⟩)
```

To reproduce the original fixed point, ϕ should be composed with u followed by another composition with u. To do this, a corollary of the universal property for exponentials must be employed

```
eval-curry : ∀ {A} {f : A × B ⇒ C}
  → eval ∘ ⟨ curry f × id ⟩ ≈ f
```

Actually applying this corollary requires more work as each transformation of product objects must be made explicit

```
  (f ∘ eval) ∘ (⟨ ϕ × id ⟩ ∘ ⟨ u , u ⟩)
≈⟨ ∘-resp-r ⟨×⟩-∘-⟨,⟩ ⟩
  (f ∘ eval) ∘ ⟨ ϕ ∘ u , id ∘ u ⟩
≈⟨ ∘-resp-r (⟨,⟩-resp (≈.sym id-r) ≈.refl) ⟩
  (f ∘ eval) ∘ ⟨ (ϕ ∘ u) ∘ id , id ∘ u ⟩
≈⟨ ∘-resp-r (≈.sym ⟨×⟩-∘-⟨,⟩) ⟩
  (f ∘ eval) ∘ (⟨ (ϕ ∘ u) × id ⟩ ∘ ⟨ id , u ⟩)
≈⟨ unassoc ⟩
  ((f ∘ eval) ∘ ⟨ (ϕ ∘ u) × id ⟩) ∘ ⟨ id , u ⟩
```

One u must be brought into the left-hand product without the right-hand one in order to match the universal property of exponentials. Another requirement for eval-curry is that $\phi \circ$ u must be wrapped inside curry. This can be done by applying the curry∘uncurry isomorphism

```
  ((f ∘ eval) ∘ ⟨ (ϕ ∘ u) × id ⟩) ∘ ⟨ id , u ⟩
≈⟨ ∘-resp-l (∘-resp-r (⟨×⟩-resp (≈.sym curry∘uncurry) ≈.refl )) ⟩
  ((f ∘ eval) ∘ ⟨ (curry (uncurry (ϕ ∘ u))) × id ⟩) ∘ ⟨ id , u ⟩
```

Now, the universal property can be applied to extract uncurry (φ ∘ u) from the product object

```
    ((f ∘ eval) ∘ ⟨ (curry (uncurry (φ ∘ u))) × id ⟩) ∘ ⟨ id , u ⟩
  ≈⟨ ∘-resp-l assoc ⟩
    (f ∘ (eval ∘ ⟨ (curry (uncurry (φ ∘ u))) × id ⟩)) ∘ ⟨ id , u ⟩
  ≈⟨ ∘-resp-l (∘-resp-r eval-curry) ⟩
    (f ∘ (uncurry (φ ∘ u))) ∘ ⟨ id , u ⟩
```

The end is in sight and all that remains is to extract the second u. This is done by collapsing $A \times 1$ to A, which can be achieved by inserting the identity for $A \times 1$ and deconstructing this into the morphisms comprising the isomorphism, oneIso and otherIso, and seeing what happens

```
    (f ∘ (uncurry (φ ∘ u))) ∘ ⟨ id , u ⟩
    ≈⟨ assoc ⟩
      (f ∘ ((uncurry (φ ∘ u)) ∘ ⟨ id , u ⟩))
    ≈⟨ ∘-resp-r (∘-resp-l (≈.sym id-r)) ⟩
      (f ∘ (((uncurry (φ ∘ u)) ∘ id) ∘ ⟨ id , u ⟩))
    ≈⟨ ∘-resp-r (∘-resp-l (∘-resp-r (≈.sym One×A⇒A))) ⟩
      (f ∘ (((uncurry (φ ∘ u)) ∘ oneIso ∘ otherIso) ∘ ⟨ id , u ⟩))
```

The isomorphisms happen to precisely be what is needed to recover the fixed point

```
      (f ∘ (((uncurry (φ ∘ u)) ∘ oneIso ∘ otherIso) ∘ ⟨ id , u ⟩))
    ≈⟨ ∘-resp-r (∘-resp-l unassoc) ⟩
      (f ∘ ((((uncurry (φ ∘ u)) ∘ oneIso) ∘ otherIso) ∘ ⟨ id , u ⟩))
    ≈⟨ ≈.refl ⟩
      (f ∘ (((collapseToOne (uncurry (φ ∘ u))) ∘ otherIso) ∘ ⟨ id , u ⟩))
    ≈⟨ ∘-resp-r assoc ⟩
      (f ∘ (((collapseToOne (uncurry (φ ∘ u)))) ∘ (otherIso ∘ ⟨ id , u ⟩)))
    ≈⟨ ≈.refl ⟩
      (f ∘ (((collapseToOne (uncurry (φ ∘ u)))) ∘ (projr ∘ ⟨ id , u ⟩)))
```

Applying projr to the product object extracts u giving us the fixed point

```
    ≈⟨ ∘-resp-r (∘-resp-r ⟨,⟩-projr) ⟩
      (f ∘ (((collapseToOne (uncurry (φ ∘ u)))) ∘ u ))
    ≈⟨ ≈.refl ⟩
      f ∘ fixedPoint
    ∎
```

The proof can be finished off by filling in the holes in the record

```
  in record { X = fixedPoint ; isFixedPoint = ≈.sym proof }
```

The contrapositive of the statement is worth defining as it is useful for some of the applications.

```
  cantor : {A B : Obj} → ¬ FixedPointProperty B → ¬ PointSurjective A (A ⇝ B)
  cantor = contraposition lawvere
```

3.3 Applications

Lawvere's fixed point theorem is an incredibly broad ranging theorem that generalises many important theorems in mathematical logic and foundational computer science. This thesis will formalise and axiomatize two specific instances, Cantor's theorem and the first fixed point theorem in the untyped λ-calculus. An analog to Cantor's theorem will be introduced alongside a category of small types in place of the category of sets. After, categorical models of the λ-calculus will be explored and the consequences of Lawvere's fixed point theorem in these models.

3.3.1 Cantor's Theorem

An analogue of Cantor's theorem can be constructed using a category of small types (i.e. all types that belong to a given universe). A category can be constructed from the elements of any Agda universe of a particular level by setting the objects of the category to be the types of the universe and the morphisms to be the functions between them.

```
instance Sets : ∀ l → Category (suc l) l l
Sets l = record
  { Obj = Set l ;
    _⇒_ = λ A B → A → B ;
```

Equality of morphisms is extensional propositional equality or functions.

```
_≈_ : (f g : A → B) → Set l
f ≈ g = ∀ x → f x ≡ g x
```

Morphism composition is function composition, identity morphsism are identity functions.

```
_≈_ = _≈_ ;
id = id ;
_∘_ = _∘_ ;
```

The remaining axioms are trivially satisfied by the above definitions. Each instance of Sets forms a cartesian closed category. The remainder of this section will be done with the lowest of these

```
Sets1 = Sets lzero
```

The cartesian closedness of Sets1 will be established by providing a terminal object, products and exponentials. Proving that Sets1 is cartesian requires showing that, for every pair of types, a product object can be formed.

```
set-product : {A B : Set} → BinaryProduct A B
set-product {A} {B} = record { prod = Pair A B ; proj = proj-pair ; isProduct = {!!} }
```

Product objects consist of an object, projections out of the object and a proof that the objects and arrows satisfy the universal property of products. Products correspond to the pair type

```
data Pair (A : Set) (B : Set) : Set where
  mkPair : A → B → Pair A B


proj-pair : ∀ {A B} i → Pair A B → Bool-elim A B i
proj-pair false (mkPair x x₁) = x₁
proj-pair true (mkPair x x₁) = x
```

All that remains is to prove that these constitute the product object i.e. that for each pair of arrows from a type to each component of the pair there exists a (extensionally propositionally) unique function from the type to the pair object that satisfies the definition of the product.

```
proj-uniqueness : ∀ {A B X} (p : ∀ i → X → Bool-elim A B i) →
  ∃![ u ] ( ∀ i (b : X) → p i b ≡ proj-pair i (u b))
```

Given the indexed-family p, the unique arrow u can be produced by extracting the values from the indexed-family and creating a pair from them

```
proj-uniqueness {A} {B} = λ p →
  let u = (λ x → mkPair (p true x) (p false x))
  in Unique.Build.∃!-intro u {!!}
       {!!}
```

Where Unique.Build.∃!-intro is the constructor for a universal mapping property.

The next field of the universal mapping type is the proof that the provided arrow satisfies the definition required from the product i.e.

```
proj-sat-univ : {A B X : Set} → {x₁ : X}{i : Bool}
  → {x : (j : Bool) → X → Bool-elim A B j}
  → x i x₁ ≡ proj-pair {A} {B} i (mkPair (x true x₁) (x false x₁))
```

Or in straightforward mathematical terms that the constructed arrow, u, satisfies (with abuse of notation)

$$\pi_1 \circ \mathtt{u} = f \wedge \pi_2 \circ \mathtt{u} = g$$

Where f and g are the two morphisms underlying p. This proves to be trivially true given the definition of proj-pair and can be directly placed as an argument to the universal mapping property.

```
Unique.Build.∃!-intro
  u
  (λ i b → proj-sat-univ {A} {B} {_} {b} {i} {p})
  {!!}
```

The final field of the universal mapping constructor is the proof that u is unique which will be elided for brevity but can be found in associated proofs.

```
proj-unique : {A B X : Set} {x : ∀ i → X → Bool-elim A B i}
              {g : X → Pair A B} →
              (∀ i (x₁ : X) → x i x₁ ≡ proj-pair i (g x₁)) →
              (x₁ : X) →
              mkPair (x true x₁) (x false x₁) ≡ g x₁
```

For a category to have finite products it must also have a terminal object. The terminal object in the category Sets1 is the unit type

```
data ⊤ : Set where
  tt : ⊤
```

To prove that ⊤ is in fact the terminal object the universal property must be proven

```
terminal-property : (X : Set) → ∃! X ⊤
terminal-property X =
  Unique.Build.∃!-intro {!!} _ {!!}
```

The first argument to the constructor is the unique arrow and the last the proof of uniqueness. The middle argument ordinarily corresponds to the property the arrow must satisfy but the property here is existence and so can be inferred automatically using an underscore as it is trivially true that the type can be inhabited.

For a given type the function to the terminal object is the function that constantly returns the single inhabitant of ⊤, tt.

```
terminal-arrow : {X : Set} → X → ⊤
terminal-arrow x = ⊤.tt
```

The final component of the terminal object is the proof of uniqueness i.e. that every function from a type to the terminal object is propositionally (extensionally) equal. This is trivially true as there is a single inhabitant of the unit type and therefore only one place for to which all functions can map.

```
terminal-unique : {X : Set} {g : X → ⊤} → ⊤ → (x : X) → ⊤.tt ≡ g x
terminal-unique x x₁ = refl
```

With this the universal mapping property can be completed

```
Unique.Build.∃!-intro terminal-arrow _ terminal-unique
```

And it can be established that Sets1 has a terminal object

```
⊤-isTerminal : IsTerminal ⊤
⊤-isTerminal = terminal-property
```

The last requirement for a cartesian closed category is exponentials. For every pair of types an exponential object must be produced consisting of a type, an evaluation map and transposition (currying). The exponential object for two types `A` and `B` is the function type between the two

```
set-exponential : {A B : Set} → Exp A B
set-exponential {A} {B} = record { C^B = A → B ; eval = set-eval ; curry′ = set-curry′}
```

The evaluation map takes a pair containing a function from a type `A` to a type `B` and term of type `A` and returns a `B`. All that is required here is to unpack the pair and apply the function to the value.

```
set-eval : ∀ {B C} → Pair (B → C) B → C
set-eval (mkPair f x) = f x
```

The last component that needs to be provided for exponentials is the curry function which for the category of small types takes the form

```
set-curry′ : ∀ {A B C} (f : Pair A B → C) →
                  ∃![ f' ∈ A ⇒ (B → C) ] ((x : Pair A B)
                    → (set-eval ∘ (λ y → mkPair (f' (fst y)) (Function.id (snd y)))) x ≡ f x)
```

Which returns a universal mapping property. The first argument of the universal mapping property, as is usual, is the mapping itself which is the curry function.

```
sets-curry : {A B C : Set} → (Pair A B → C) → (A → B → C)
sets-curry f = λ x y → f (mkPair x y)
```

The second argument is a proof that sets-curry satisfies the universal property.

```
sets-curry'-sat : ∀ {A B C} (f : Pair A B → C) → (x : Pair A B)
    → (set-eval ∘ (λ y → mkPair ((sets-curry f) (fst y)) (Function.id (snd y)))) x ≡ f x
```

By making use of the universal property for products

```
pairPrf' : {A B : Set} → {g : Pair A B}
          → mkPair (proj-pair true g) (proj-pair false g) ≡ g
pairPrf' {A} {B} {mkPair x x_1} = refl
```

The proof can be completed trivially by reducing the left hand side of the propositional equality in the above type

```
sets-curry'-sat = λ f x → begin
                              f (mkPair (proj-pair true x) (proj-pair false x))
                            ≡⟨ cong f pairPrf' ⟩
                              f x
                            ∎
```

The last component of the unique mapping property is the proof of uniqueness of the map the type of which is

```
sets-curry-unique : {A B C : Set} →
                    {f : Pair A B → C} →
                    {g : A → B → C} →
                    ((x : Pair A B) → g (proj-pair true x) (proj-pair false x) ≡ f x) →
                    (x : A) → (λ y → f (mkPair x y)) ≡ g x
```

This proof proves to be difficult to complete. This is an extensional proof equating the two desired structures however, within MLTT, function extensionality is not derivable. Function extensionality does not lead to inconsistencies within MLTT when postulated and therefore is done here to allow the proof to proceed.

```
postulate
  ext : Extensionality lzero lzero
```

Where extensionality equates functions that are pointwise equal

```
Extensionality : (a b : Level) → Set _
Extensionality a b = {A : Set a} {B : A → Set b} {f g : (x : A) → B x} →
                     ----------------------------
                     (∀ x → f x ≡ g x) → f ≡ g
```

With extensionality the proof can be completed without significant difficulty

```
= let tproof = λ y → fprf (mkPair x y)
  in (begin
      (λ y → f (mkPair x y))
    ≡⟨ ext (λ t → sym (tproof t)) ⟩
      (λ y → g x y)
    ≡⟨ refl ⟩
      g x
    ∎ )
```

The above defined function complete the definition of set-curry'

```
set-curry′ f = Unique.Build.∃!-intro (sets-curry f) (λ x → sets-curry'-sat f x) sets-curry-unique
```

Sets1 can now be defined to be closed

```
instance
  setsHasExponentials : HasExponentials Sets1
  setsHasExponentials = record { _⇝′_ = λ B C → set-exponential }
```

With these definitions the CCCness of Sets1 can be trivially established.

The category that has been is unfortunately not a valid model of set theory and is only a toy language for reasons that can be found in Section 4.2.1. A type theoretic analogue of Cantor's theorem can be established showing there is no point-surjection from a type to the predicates on the type. In set theory, for a given set, the predicates on the set can be considered as subsets placing the elements of the powerset of a set in correspondence with the predicates on the set. With no coherent notion of membership relation or subset relation (by design) there is no way to faithfully model the Cantor's theorem within the category produced. Even so, the proof shows, with enough similarity, how the argument would proceed in the category of sets.

Within Sets1, predicates are functions from a type A to Bool type.

```
data Bool : Set where
  true : Bool
  false : Bool
```

With Bool, Cantor's theorem can be reframed within Sets1 as

```
cantorsDiagonalTheorem : ∀ {A} → ¬ PointSurjective A (A → Bool)
```

Or that, in English, for all types (in Set) there does not exist a point-surjection from the type to the predicates on the type. It is unclear to the author as to whether point-surjectivity has a more coherent interpretation within Sets1.

To prove this the contrapositive of lawvere, cantor will be used. As a reminder

```
cantor : {A B : Obj} → ¬ FixedPointProperty B → ¬ PointSurjective A (A ⇝ B)
cantor = contraposition lawvere
```

To make use of this it is necessary to show that Bool does not have the fixed point property.

```
noFixPtBool : ¬ FixedPointProperty Bool
```

Recall that the definition of ¬ is

```
¬_ : ∀ {l} → Set l → Set l
¬ P = P → ⊥
```

Therefore, given a function which finds the fixed point of any function from Bool to Bool, an inhabitant of void must be provided. The observation that results in Cantor's theorem being an application of Lawvere's theorem is that there is a function from Bool to Bool that does not have a fixed point, the familiar function from classical logic, negation or not.

```
not : Bool → Bool
not false = true
not true = false
```

In fact, with respect to sets, every set with more than one element has at least one function to itself without a fixed point leading to an extension of Cantor's theorem i.e. that for all sets A and B with cardinality greater than one there does not exist a surjective function from $A \to B^A$. Before returning to noFixPtBool, a proof that not is needed.

```
not-fx-pt : ∀ {x} → (not x) ≢ x
```

not x reduces to a different normal form to x for both true and false and therefore the absurd pattern can be introduced in both cases introduced using () which indicate that there is no constructor that is valid for the argument,

```
not-fx-pt {false} ()
not-fx-pt {true} ()
```

Here the absurd patterns are used, given that the not-fx-pt is now used to derive noFixPtBool through a contradiction. ¬noFixPtBool is a function which, given a proof that Bool has a fixed property, can derive false. The proof that Bool has the fixed point property can be used to derive a fixed point for not which we have already proven does not have a fixed point. This is done through the with construct within Agda. with allows an intermediate computation to be pattern matched on. In the case of noFixPtBool it is the result of applying the proof of the fixed proof property Y, a fixed point combinator, to the not function.

```
noFixPtBool Y with (Y not)
```

Through this, a fixed-point, X of not can be extracted alongside a proof of X fixed pointedness.

```
... | record { X = X ; isFixedPoint = isFixedPoint } = {!!}
```

Now, within the scope of the noFixPtBool, there exist proofs that not both does and doesn't have a fixed point. not-fx-pt has takes an equality of not x and x and returns an element of İt is clear that by the proof of the existence of a fixed point to this will derive ⊥ as needed.

```
... | record { X = X ; isFixedPoint = isFixedPoint } = not-fx-pt (isFixedPoint ⊤.tt)
```

cantorsDiagonalTheorem can now be derived as a direct application of cantor

```
cantorsDiagonalTheorem : ∀ {A} → ¬ PointSurjective A (A → Bool)
cantorsDiagonalTheorem = cantor Sets1 noFixPtBool
```

3.3.2 The λ-Calculus

There is much to suggest that a coherent interpretation of Lawvere's theorem exists in the λ-calculus. The untyped λ-calculus is famous for Curry's fixed point combinator of which a consequence is that every λ-term has a fixed point under application (known as the first fixed point theorem) indicating that perhaps a direct application in an appropriate category could yield this result. In addition to this, significant results in different models of computation are given as an application of the theorem as outlined in Section 1.4.3. This observation has not gone unnoticed by others. nLab [35], an online encyclopedia for category theory, hosts a webpage for Lawvere's fixed point theorem [36] which states in **Remark 2.6**.

> *Many applications of Lawvere's fixed point theorem are in the form of negated propositions, e.g., there is no surjection from a set to its power set, or Peano arithmetic cannot prove its own consistency. However, there are positive applications as well, e.g., it implies the existence of fixed-point combinators in untyped lambda calculus.*

This claim affirms the notion that Curry's fixed combinator could be derived as an instance of Lawvere's theorem. Despite this remark, the realities of the situation are not so simple. The `nLab` page does not provide a source for their remark and, within the literature, there are only informal proofs of this claim. Upon further examination the informal proofs do not truly reflect Lawvere's theorem in the context of the untyped λ-calculus. These previously presented proofs will be examined more closely in Section 4.3.1. In this thesis, a precise account of the relationship indicated within the remark on `nLab` will be presented, which does not quite extend to establishing the existence of fixed point combinators within the λ-calculus but a straightforward corollary - the first fixed point theorem.

The proof that Lawvere's theorem implies the existence of fixed points for all λ-terms rests on the observation first made by Dana Scott in his development of domain theory, that λ-terms can also be considered as mappings between λ-terms. This naturally engenders a desire for some object D that is isomorphic to D^D, the function space on D. This is impossible for any set within a set theory that rejects unrestricted comprehension. Various models were constructed through the latter half of the 20th century including Scott's D_∞ and Plotkin's $P\omega$. These ideas were later generalised by identifying that all models of the λ-calculus arise from cartesian closed category with an object D that has a retraction to its own function space, known as a reflexive object.

The relationship between these constructions and Lawvere's theorem can be understood by observing that in any CCC with a reflexive object D there is a point surjective morphism from D to D^D, precisely the retraction. Formalising this in Agda first requires a formalisation of a reflexive object and retractions. In an arbitrary category a retraction between two objects `A` and `B` is

```
record Retract (A B : Obj) : Set (lo ⊔ la ⊔ l≈) where
  field
    forth : A ⇒ B
    back : B ⇒ A
    forth-back : forth ∘ back ≈ id
```

i.e. A pair of arrows in both directions between `A` and `B` and a proof that the composition of the two form the identity in a given direction. A reflexive object is simply some object `D` alongside a retraction, Retract D $\mathtt{D}^{\mathtt{D}}$. Now the relevant corollary to Lawvere's fixed point theorem can be stated precisely, working in a CCC

```
corollary : {X : Obj} → Retract X (X ⇝ X) → FixedPointProperty X
```

The proof can be created by creating a point-surjective function from `X` to $\mathtt{X}^{\mathtt{X}}$ and applying the earlier proof of lawvere. First, the retraction can be pattern matched on and Lawvere introduced with the arrow from `X` to $\mathtt{X}^{\mathtt{X}}$ with only a proof of point surjectivity required.

```
corollary {X} record { forth = forth ; back = back ; forth-back = forth-back }
  = lawvere C {X} (record { arr = forth ; isPointSurjective = λ b → {!!} })
```

With `b` being a general point to $\mathtt{X}^{\mathtt{X}}$, a point to `X` needs to be provided alongside a proof that `b` is the solution to the point-surjective equation. The point to `X` is found by using the retraction. The proof of equality is simply achieved by exploiting the definition of a retraction to collapse the identity.

```
record { X = back ∘ b ; isSolution = begin
                                   forth ∘ (back ∘ b)
                                 ≈⟨ unassoc ⟩
                                   (forth ∘ back) ∘ b
                                 ≈⟨ ∘-resp-l forth-back ⟩
                                   id ∘ b
                                 ≈⟨ id-l ⟩
                                   b
                                 ∎ } })
```

The implications of this corollary can be understood from this excerpt from Section 5 of Barendregt's *The Lambda Calculus: Its Syntax and Semantics* [1] on models.

> *"...for the construction of a λ-calculus model it is sufficient to have an object D in a CCC such that D^D is a retract of D."*

From the above quote and the earlier corollary to Lawvere's theorem it can be concluded that the categorical interpretation of every model of the λ-calculus has an object with the fixed point property.

In the section that follows, several are results are stated without proof for brevity. The relevant theorems and definitions from *The Lambda Calculus: Its Syntax and Semantics* will be pointed to for reference. Models, in the model-theoretic sense, are helpful for exploring properties of the λ-calculus that are not immediate from the equational theory and syntax itself. The general class of structures that are models of the λ-calculus are known as λ-algebras. The definition of λ-algebras are predicated on that of an applicative structure. An applicative structure is a tuple $M = (A\,,\,\bullet)$ where A is a set and $\bullet$ is a binary operation on A.

A useful type of models to be considered are the class of syntactic models. A syntactic applicative structure adds a method of interpreting terms in the λ-calculus, elements of the set Λ, into the applicative structure. In other words a syntactic applicative structure is a triple $M = (A\,,\,\bullet\,,\,[\![\,]\!])$ of an underlying set A, a binary operation $\bullet : A \to A$ and a syntactical interpretation function $[\![\,]\!]$. See **Definitions 5.3.1** and **5.3.2** for a precise definition of $[\![\ ]\!]$ and an appropriate definition of satisfaction, $\vDash$.

The constraint which turns a syntactic applicative structure, P, into a syntactic λ-algebra is

$$\lambda\beta \vdash M = N \Rightarrow P \models [\![M]\!] = [\![N]\!]$$

Or that every two λ-terms that are equal under $\lambda\beta$ are equal under their interpretation within the λ-algebra. There is a close relationship between λ-algebras and CCCs. Every λ-algebra can be transformed into a CCC with a reflexive object via a process known as the *Karoubi Envelope*, see **Definition 5.5.11.**. Furthermore, every CCC with a reflexive object can be turned into a λ-algebra such that taking a λ-algebra to a CCC and back to λ-algebra again produces an isomorphic λ-algebra, established in **Theorem 5.5.13**. This indicates that every λ-algebra can be obtained by a CCC with a reflexive object. The results of applying corollary can be interpreted in both directions of this transformation. Interpreting within the context of the transformation from *Karoubi envelope* to CCC yields no sensical results. The construction of the *Karoubi envelope* and the investigation of its relationship with Lawvere's theorem are detailed in Appendix A. Interpreting in the other direction, however, is more useful.

A locally-small CCC with a reflexive object, D, with arrows $F : D \to D^D$ and $G : D^D \to D$ can be turned into a λ-algebra as follows. The underlying set of the λ-algebra are the points to D, written $|D|$. The binary operation, $\star$, of the generated λ-algebra that operates on points, a, b to D is as follows:

$$a \star b = eval \circ \langle\, F \times id \,\rangle \circ \langle a\,,\,b\,\rangle$$

Definition 5.5.3. defines a semantic interpretation function for λ-terms for which the triple of $(\,|D|\,,\,\star,\,[\![]\!]\,)$ is shown to be a λ-algebra in **Theorem 5.5.6.**.

With corollary and the fact that the above transformation yields all λ-models, Lawvere's fixed point theorem can be used to prove the first fixed point theorem in all λ-models. The proof proceeds as follows. In some cartesian closed category with a point surjective arrow, PS.arr, the operation constituting $\star$ in the generated λ-algebra can be written as follows

```
_⋆_ : (A : Point X) → (B : Point X) → (Point X)
a ⋆ b = eval ∘ ⟨ PS.arr × id ⟩ ∘ (⟨ a , b ⟩)
```

Proving the first fixed point theorem in every λ-algebra amounts to showing that the following type is inhabited

```
first-fixed-point-theorem : (f : Point X) → Σ (Point X) (λ x → f ⋆ x ≈ x)
```

Or that every point to the reflexive object has a fixed point under $\star$. To prove this, the fixed point must be provided and a proof that it is a fixed point. The fixed point can be constructed as follows beginning by precomposing the point-surjective arrow with the given point to `f`

```
first-fixed-point-theorem f
  = let x = (PS.arr) ∘ f
```

This gives a point to $\mathtt{D}^{\mathtt{D}}$. This can be turned into a endomorphism on `D` as follows by passing it through two familiar isomorphisms

```
y = collapseToOne (uncurry (x))
```

Lawvere's fixed point theorem can now be used to find a fixed point for y

```
z = lawvere C ps y
```

The fixed point and proof can be extracted as follows

```
fixedPoint = HasFixedPoint.X z
fixedPointProof = HasFixedPoint.isFixedPoint z
```

fixedPoint is the fixed point for ⋆. Now, it needs to be shown that fixedPoint is in fact a fixed point for f under ⋆ i.e.

```
proof = begin
          f ⋆ fixedPoint
        ≈⟨ {!!} ⟩
          fixedPoint
        ∎
```

The definition of ⋆ can be expanded to

```
  f ⋆ fixedPoint
≈⟨ ≈.refl ⟩
  eval ∘ ⟨ PS.arr × id ⟩ ∘ ⟨ f , fixedPoint ⟩
```

From this point the proof proceeds in a similar fashion to the proof of lawvere's fixed point theorem. The key observation here is that any point to X can composed with Ps.arr to give a point to $\mathtt{X}^{\mathtt{X}}$ which can be pushed through familiar isomorphisms to give a endomorphism on X. The application of the generated applicative structure amounts to converting the left hand point of the operation into an endomorphism and then composing with the right hand point.

```
≈⟨ ∘-resp-r ⟨×⟩-∘-⟨,⟩ ⟩
  eval ∘ ⟨ (PS.arr ∘ f) , (id ∘ fixedPoint) ⟩
≈⟨ ∘-resp-r (⟨,⟩-resp (≈.sym id-r) ≈.refl) ⟩
  eval ∘ ⟨ (PS.arr ∘ f) ∘ id , id ∘ fixedPoint ⟩
≈⟨ ∘-resp-r (≈.sym ⟨×⟩-∘-⟨,⟩) ⟩
  eval ∘ ⟨ (PS.arr ∘ f) × id ⟩ ∘ ⟨ id , fixedPoint ⟩
≈⟨ unassoc ⟩
  (eval ∘ ⟨ (PS.arr ∘ f) × id ⟩) ∘ ⟨ id , fixedPoint ⟩
≈⟨ ∘-resp-l (∘-resp-r (⟨×⟩-resp (≈.sym curry∘uncurry) ≈.refl )) ⟩
  (eval ∘ ⟨ (curry (uncurry (PS.arr ∘ f))) × id ⟩) ∘ ⟨ id , fixedPoint ⟩
≈⟨ ∘-resp-l eval-curry ⟩
    (uncurry (PS.arr ∘ f)) ∘ ⟨ id , fixedPoint ⟩
≈⟨ ∘-resp-l (≈.sym id-r) ⟩
  (uncurry (PS.arr ∘ f) ∘ id) ∘ ⟨ id , fixedPoint ⟩
≈⟨ ∘-resp-l (∘-resp-r (≈.sym One×A⇒A)) ⟩
  (uncurry (PS.arr ∘ f) ∘ oneIso ∘ otherIso) ∘ ⟨ id , fixedPoint ⟩
≈⟨ ∘-resp-l unassoc ⟩
  ((uncurry (PS.arr ∘ f) ∘ oneIso) ∘ otherIso) ∘ ⟨ id , fixedPoint ⟩
≈⟨ assoc ⟩
  (uncurry (PS.arr ∘ f) ∘ oneIso) ∘ (otherIso ∘ ⟨ id , fixedPoint ⟩)
≈⟨ ≈.refl ⟩
  (collapseToOne (uncurry (PS.arr ∘ f))) ∘ (projr ∘ ⟨ id , fixedPoint ⟩)
≈⟨ ∘-resp-r ⟨,⟩-projr ⟩
  (collapseToOne (uncurry (PS.arr ∘ f))) ∘ fixedPoint
```

The right hand side of the outermost composition now matches the constructed `y` from earlier

```
  (collapseToOne (uncurry (PS.arr ∘ f))) ∘ fixedPoint
≈⟨ ≈.refl ⟩
  y ∘ fixedPoint
```

`fixedPointProof` can now be applied to make use of lawvere's fixed point theorem.

```
  y ∘ fixedPoint
≈⟨ fixedPointProof ⟩
  fixedPoint
∎
```

The above theorem shows that the syntactic applicative structure generated from any CCC with a point surjective morphism has a fixed point theorem. To connect this result back the terms of the λ-calculus and the equational theory $\lambda\beta$, **Theorem 5.2.18** from [1] must be considered which states

Theorem. For all $M, N \in \Lambda \quad \lambda\beta \vdash M = N \Leftrightarrow M = N$ is true in all λ-models

Given the earlier established fact that all λ-models can be obtained from CCCs with a reflexive object and that all CCCs with a reflexive object induce a syntactic applicative structure with a fixed point theorem it is possible to conclude, therefore, that the λ-calculus has a fixed point theorem under application. Whilst not the briefest or most insightful proof, it raises further questions concerning categorical interpretations of the untyped λ-calculus. λ-algebras are all the models that, for their interpretation of λ-terms, satisfy the equations of $\lambda\beta$. Other structures exist that aim to characterise all models for which other equational theories hold of their interpretation of λ-terms. For instance, λ-models are an extension of λ-algebras which also satisfy the Meyer-Scott axiom of weak-extensionality.

The class of CCCs that give rise to the λ-models are those with a reflexive object that *has enough points*. An object, A, has enough points if for all f, $g : A \to A$

$$f \neq g \implies \exists\, x : 1 \to X \quad f \circ x \neq g \circ x$$

Furthermore, there are the extensional λ-algebras that correspond the equational theory which satisfies the η rule i.e.

$$P \vDash \forall x(\lambda x.Mx) = M$$

The class of CCCs that give rise to these structures are those with an object D that *has enough points* but, instead of D being reflexive, is instead isomorphic to its exponential object i.e. $D \equiv D^D$.

Given that point surjectivity gives a fixed point theorem for any applicative structure generated for it a natural question that arises is to what underlying structure it might correspond. The notion is certainly weaker than that of certainly weaker than that of a λ-algebra as the arrow from $D^D \to D$ for a reflexive object is made use of when defining the semantic interpretation function giving rise to λ-algebras. This thesis does not provide a concrete answer to this question (see Section 4.4.1) but provides some combinators that are derivable in any applicative structure derived from a CCC with a point surjective object. The derivations of the combinators will be presented informally due to their similarity to the above proofs.

These combinators utilise the fact that $a \star b = \overline{uncurry\,(\varphi \circ a)} \circ b$. If a can be picked then, any endomorphism on D can be recovered. More precisely, for any $f : D \to D$, this can be turned into a point to D^D by pushing through the other way in the $1 \times A \cong A$ isomorphism and exploiting the other direction of the adjunction, here given the name curry i.e. $f' = curry\,(\,\underline{f}\,) : 1 \to D^D$. The point-surjectivity of φ can now be used to find the equivalent u such that $\varphi \circ u = f'$. Considering $u \star b$ for any b

$$\begin{aligned} u \star b &= \overline{uncurry\,(\varphi \circ u\,)} \circ b \\ &= \overline{uncurry\,(\,f'\,)} \circ b \\ &= \overline{uncurry\,(\,curry\,(\,\underline{f}\,)\,)} \circ b \\ &= f \circ b \end{aligned}$$

A fairly easy combinator to construct is the identity combinator $\mathbf{I}\,x = x$ by taking f in the above construction to be id and calculating the equivalent u.

Another useful combinator is the mockingbird or self-application operator, $\mathbf{M}x = xx$ by taking f to be the following morphism of type $D \to D$, **M’** $= eval \circ \langle \varphi \times id \rangle \circ \delta$ and finding the equivalent u and setting it to be **M**.

The final combinator a derivation is given for was found whilst attempting to recover the **K** combinator and requires slightly more machinery.

Let $x = id : D \to D$, $y = curry\,(\,\underline{x}\,) : 1 \to D^D$ pick $z : 1 \to D$ s.t. $\varphi \circ z = y$ from the point-surjectivity of φ let $q = z \circ !_D : D \to D$ where $!_D$ is the terminal arrow from D. Taking f as q and deriving the appropriate u an interesting combinator is derived. Calling the appropriate u, **F**

$$\begin{aligned}
\mathbf{F} \star a \star b &= (\mathbf{F} \circ a) \star b \\
&= eval \circ \langle \varphi \times id \rangle \circ \langle q \circ a, b \rangle \\
&= eval \circ \langle \varphi \times id \rangle \circ \langle z \circ !_D \ \circ\ a,\ b \rangle \\
&= eval \circ \langle \varphi \circ z \times id \rangle \circ \langle !_D \ \circ\ a\ ,\ b \rangle \\
&= \overline{uncurry\,(\varphi \circ z\,)} \circ b \\
&= id \circ b \\
&= b
\end{aligned}$$

i.e. **F** selects the second of its two arguments.

Chapter 4

Critical Evaluation

4.1 Alternative Theorem Provers

A relatively arbitrary decision was the choice of proof-assitant to use. Of the plethora that exist, the two most popular in use today are Coq [42] and Agda. The reasons for choosing Agda were largely convenience. Agda closely resembles strongly typed functional programming languages like Haskell in syntax and therefore is more likely to resemble a more familiar programming experience to computer scientists. The propositions-as-types approach is taken to proving where proof objects are easily manipulable. Coq is a dependently typed functional programming language supported by INRIA and originally developed by Gerard Huet and Thierry Coquand based on an alternative to MLTT called the Calculus of Inductive Constructions [38]. Proving in Coq does not consist of pattern matching and manipulating proof objects but instead through the refinement of goals through *tactics*. Tactics are procedures which take a goal (proof) and apply a deductive step in reverse to produce subgoals which themselves must be solved. This procedure continues until all goals have been satisfied and the proof is complete. In this way, the direct manipulation of terms can be avoided and large steps of proofs can be automated. Users are able to define their own tactics allowing for faster and more automated proofs. This style is not without its disadvantages. Coq proofs consist of a list of the tactics used. These can be difficult to read and can require more cognitive effort to understand. The timeframe for this project favoured the more intuitive and familiar interface Agda offered. Declarative style proofs i.e. begin-style, however, are not well supported in Agda, with no editor integration.

4.2 Limitations

As mentioned earlier in Section 2.2.1, a setoid based approach was taken towards modelling categories where categories were parameterised by an equivalence relation on morphisms. This limitation arose due to the inability to represent quotient sets within the type theory of plain Agda. This was a limitation in several different ways. As previously mentioned, for arbitrary equivalence relations, there is no support for indiscernibility of identicals and congruence must therefore be proved for every type individually. This did not prove to be a particularly annoying component of proving within the scope of this thesis. This, however, is primarily due to not working in any instances of categories that required working with an equivalence relation of morphisms. It would be difficult to take the version of Lawvere's theorem defined within this thesis and apply it to a non-trivial category.

4.2.1 Constructive Category of Sets

Another, more restrictive limitation is the lack of a category of sets to work within. In Section 3.3.1, rather than working in a real category of sets, an analog to the theorem was proven in a category of small types. This was not a faithful interpretation as, within type theory, the notion of typing is *external* to the theory, whereas a key notion within set theory is that the membership relation is internal to the theory. This is what allows the notion of powerset to be formalised and thus takes the formulation presented in Section 3.3.1 to a faithful interpretation of Cantor's theorem. To construct a category of sets it is necessary to work within a different type theory.

A type theory which is supported by Agda and provides a solution to both of the problems outlined above is the new and radically different type theory known as homotopy type theory.

4.2.2 Homotopy Type Theory

Homotopy type theory (HoTT) [44] is an extension of MLTT in which the higher dimensional structure of the equality type is embraced. homotopy type theory rejects Axiom K which implies uniqueness of identity proofs. By doing so the so-called higher homotopy structure of the identity type can be exploited. This is achieved by viewing equality between two types as a path in space between two points. Two different paths can be considered as well as homotopies (continuous deformations) between them. Through rejecting Axiom K, it is no longer possible to prove that refl is the only inhabitant of the identity type. An upside of this is that new inhabitants of the equality type for a given type can now be introduced without inconsistency. It is possible to imagine that, to create a quotient type, new equalities can be introduced corresponding directly to the equivalence classes desired. Types with these non-trivial equalities are known as higher inductive types.

Rejecting Axiom K by itself provides benefits to the type theory but another significant advantage can be gained, namely the introduction of the univalence axiom. The univalence axiom, inconsistent with Axiom K, is an axiom which states that equivalence is equivalent to equality.

$$(A = B) \simeq (A \simeq B)$$

Here, equivalence is meant in the category theoretic sense in that two types A and B belonging to some universe U are equivalent if there exists an arrow $f : A \to B$ with both a left inverse and right inverse. The univalence axiom captures an informal mathematical practice of equating two isomorphic structures, allowing the informal reasoning in mathematics to be done within the type theory. Univalence simplifies the task of working with higher inductive types and allows mathematics such as Category theory (very much dependent on classifying structures up to isomophism) and homotopy theory to be embedded succinctly within type theory. By postulating univalence it is possible to recover a classical principle of function extensionality without postulation.

Within homotopy type theory it is possible, as mentioned prior, to quotient your types by abitrary equivalence relations as a higher inductive type. Axiom J without Axiom K ensures that these higher inductive types behave the same as refl, meaning that indecernability of identicals is preserved. This allows categories to be defined with propositional equality without the additional requirement of congruence for the equivalence relation. Within homotopy type theory it is possible to define a set type which corresponds to the rules of ZFC with a coherent notion of set membership. This is outlined in **Chapter 10** of the HoTT book [44] . Using this construction, a coherent category of sets could be produced within the type theory and Lawvere's fixed point theorem could be applied appropriately.

Univalence is posited as an axiom within homotopy Type theory but work has been done recently to provide a constructive interpretation of the univalence axiom in a type theory known as Cubical type theory [6]. Support for Cubical type theory exists in Agda as of version 2.6 providing support for higher inductive types and an equality path type.

From the above remarks it is clear that almost all the limitations of the current formalisation of categories would be solved by moving towards a homotopy type theory setting, hopefully resulting in a simpler and more satisfying proving experience.

4.3 Importance of Theorem Proving

Part of the work done during this thesis was to investigate how computer-aided mathematics is done and whether it is worth being pursued in mainstream mathematics. With respect to the latter question, throughout this thesis it has become clear that, although there are certainly limitations and problems, computer-aided theorem proving is a positive step forward for mathematics. Much mathematical reasoning is done informally and even in published proofs many steps are left for the reader to internally fill in. There is no shortage of mathematical proofs that have been published only for mistakes to have been found. The Russian Mathematician Vladimir Voevodsky, a central figure behind the development of homotopy type theory, began his career in algebraic topology. In an open letter entitled *The Origins and Motivations of Univalent Foundations* as part of the Institute for Advanced Study Letter published in 2014 [40], Voevodsky outlined his motivations for creating more advanced tools in which to *do* mathematics. He outlines the mistakes and fear at discovering mistakes in theorems he had published and minor conflicts between mathematicians over the existence of counter examples. The motivations behind the programme of theorem proving can be neatly summed up by Voevodsky:

> *And I now do my mathematics with a proof assistant. I have a lot of wishes in terms of getting this proof assistant to work better, but at least I don't have to go home and worry about having made a mistake in my work. I know that if I did something, I did it, and I don't have to come back to it nor do I have to worry about my arguments being too complicated or about how to convince others that my arguments are correct. I can just trust the computer.*

In attempts to investigate the relationship between the λ-calculus and Lawvere's fixed point theorem, two informal attempts to outline the relation were found. These attempted proofs did not fully link the untyped λ-calculus to Lawvere's theorem. Through working in a highly rigorous setting, there is no reason to doubt that the relation ship outlined within this thesis is correct. Although not all the relationships have been investigated end-to-end, the tricky aspect of most proofs (i.e. the algebraic manipulation) has been formalised and checked. The work formalising Lawvere's theorem was vital to observing its connection to the λ-calculus. In theorem proving, none of the details can be elided so a subtle and deep understanding of the intricacies of the theorem is developed. Below are the two proofs that were found, with commentary on their aims and developments.

4.3.1 Prior Unifications of the Untyped λ-calculus and Lawvere's Theorem

The CCC generated from a simply-typed λ-calculus

A recent Bachelors thesis, *Category Theory and the Lambda Calculus* by Mario Román Garcia [17] provides an attempt to integrate Lawvere's theorem into the theory of the untyped λ-calculus.

Garcia uses Joachim Lambek's observation that every simply-typed λ-calculus has an associated cartesian closed category which is obtained by considering the objects of the category to be the types of the λ-calculus and the morphisms to be the functions between types. Lambek showed in [26] that these categories were cartesian closed. Through the familiar argument that the untyped λ-calculus can be viewed as a simply typed calculus i.e. that untyped is uni-typed - the untyped λ-calculus can be viewed as a simply typed λ-calculus with one type, D. Garcia uses this argument and also provides a retraction pair $r : \mathsf{D} \to (\mathsf{D} \to \mathsf{D})$, $s : (\mathsf{D} \to \mathsf{D}) \to \mathsf{D}$. Following this line of reasoning, Garcia establishes that considering the untyped-as-unityped λ-calculus as a cartesian closed category yields a category with a reflexive object, D where terms can be considered as morphisms via the retraction map. By Lawvere's fixed point theorem every λ-term, $d : \mathsf{D}$, has a point $p : 1 \to \mathsf{D}$ such that $d \circ p = p$. Garcia claims in **Corollary 4.20** that this proves that every term in the untyped λ-calculus has a fixed point. This argument fails on a crucial point. The cartesian closed category generated from the untyped-as-unityped λ-calculus has a single object. By the definition of a cartesian closed category this object *must* be the terminal object. To recap, the definition of a terminal object is that for all objects in the category there is a unique arrow from the object to the terminal object. This condition applies to the terminal object itself and therefore there is a unique arrow from the terminal object to the terminal object. By the definition of a category every object must have a terminal object and therefore the only arrow from the terminal object to itself can be the identity arrow. With this reading it is clear that there can be no syntactic interpretation of this arrow that yields the terms of the untyped λ-calculus. A key component of result in Section 3.3.2 was that it gave a fixed point theorem in every λ-algebra which allowed a route back to the untyped λ-calculus.

Interpretation in the reflexive cpo in the CCC - CPO$\perp$

The argument that follows is not incorrect but is derivable as a corollary via the result in this thesis.

The argument from a note published by Dan Frumin and Guillaume Massas [16]. Their arguments draw on material from throughout the Barendregt [1] which will be made clear when necessary. They begin their argument by considering the category $\mathbf{CPO}_\perp$. The objects of this category are directed complete partial orders (cpos) with a least element and arrows are scott-continuous functions. **Theorem 1.2.16** in the Barendregt shows that this category is cartesian closed. Frumin and Massas begin by stating without proof that there is a reflexive object in $\mathbf{CPO}_\perp$. Using Lawvere's theorem, Frumin and Massas conclude that every endomap on this object has a fixed point in the categorical sense. **Section 5.4** of the Barendregt outlines how every reflexive cpo with maps $F : D \to (D \to D)$ and $G : (D \to D) \to D$ induces a λ-model. The underlying set of the λ-model is the underlying set of the cpo and the binary operation $\bullet : D \times D \to D$ is defined on $x, y \in D$ as

$$x \bullet y = F(x)(y)$$

Frumin and Massas argue to the effect that, given that every endomap has a categorical fixed point, that scott-continuous function $f : D \to D$ must have a fixed point. This is valid reasoning as the points to a cpo correspond to the elements of the cpo and therefore every endomorphism on D having a fixed point $p : 1 \to D$ corresponds to every scott-continuous function on D having a fixed point. Whilst the reasoning of this argument is correct what has been proven is that a particular model of the λ-calculus has fixed points. Whilst suggestive there is no guarantee that it will be possible to translate this back to the syntax of the untyped λ-calculus. The extension to work with abstract cartesian closed categories with a reflexive object is a key component of relating the result back the syntax of the untyped λ-calculus. This could instead be considered an application of Lawvere's theorem to domain theory.

4.4 Future Work

There is much scope to extend the work done within this thesis. There are a significant number of applications of Lawvere's theorem which have yet to be explored within a computational setting. The formalised proof of Lawvere's theorem could be used to instantiate these paradoxes if the appropriate categories are constructed within the type theory. Many of the applications and paradoxes are within the category of sets or a set-like category.

With this in mind the most practical next step would be to reformalise the work done within this thesis within a language that supports homotopy type theory taking advantage of univalence and higher inductive types. There is much groundwork in place to assist in this endeavour, **Chapter 9** of the HoTT book details how category theory can be constructed within HoTT and there exist implementations of this within Cubical Agda [22]. The process of reimplementing the proofs within this thesis in a cubical setting would be an interesting process itself to examine. Working within Agda during this thesis was at times tedious but ultimately an intellectually tractable process. HoTT represents a significant shift in the viewpoint of theorem proving and its practicality is still worth being assessed. Does the cognitive overhead outweigh the practical advantages that univalence provides. Once reformalised, a it would be possible to work within a category of sets where the aforementioned questions could be formalised.

As was mentioned at the end of Section 3.3.2 to which computational calculi point surjectivity corresponds is left as an open question. The structure in question yields a fixed point theorem and several interesting combinators. Whether or not point-surjectivity yields a functionally complete set of combinators in its generated applicative structure is predicated on the categorical analog of models of combinatory logic, combinatory algebras.

4.4.1 Combinatory Algebras and Lawvere's Theorem

Longo and Moggi [31] provide the categorical analogue of the combinatory models, the class of models of combinatory logic. A combinatory algebra is a 4-tuple $(A, \bullet, k, s)$ where A is a set, $\bullet$ is a binary operation on A, and k and s are distinguished elements of A that satisfy

$$\begin{aligned} &k \bullet x \bullet y = x \\ &s \bullet x \bullet y \bullet z = x \bullet z \bullet (y \bullet z) \end{aligned}$$

An interesting observation is that, to model combinatory algebras, the category does not have to be cartesian closed. This is noteworthy as Lawvere's theorem also has an interpretation solely within a cartesian category. Longo and Moggi establish that all combinatory algebras can be obtained from a cartesian category with an object U such that there exist arrows $f : U \times U \to U$ and $g : U \to U \times U$ such that $g \circ f = id \textit{ and}$ that there exists a K-universal arrow $u : U \times U \to U$.

In a cartesian category, an arrow $u : X \times Y \to Z$ is K-universal if for all $f : X \times Y \to Z$ there exists a unique $s : X \to X$ such that $f = u \circ \langle\, s \times id \,\rangle$.

By transporting the definition of Lawvere's theorem outlined within this thesis through the adjunction between the functors $- \times Y$ and $-^Y$ a statement is obtained within cartesian categories. The equivalent theorem proceeds as follows and is outlined in **Theorem 2.2.** of [29]. In a cartesian category if there exists some $g : A \times A \to Y$ such that, for all $f : A \to Y$ there exists an $x : 1 \to A$ such that for all $a : 1 \to A$

$$f \circ a = g \circ \langle a, x \rangle$$

The interplay between the two definitions in this section will do much to enlighten the relationship between functional completeness and Lawvere's fixed point theorem and is a topic for future study.

Chapter 5

Conclusion

The motivating force behind this thesis was an investigation into mathematical logic with a particular focus in theorem proving and category theory.

Within the thesis an account of category theory has been presented within the theorem prover Agda. Lawvere's fixed point theorem, a significant theorem within the study of mathematical logic and category theory has been formalised. Proofs in the theorem prover Agda are detailed step by step and the subtleties of the language explained.

Two applications have been provided: The first, a type-theoretic analogue of Russell's paradox within a constructed category of small types. The second, a more in depth attempt, was a novel application of Lawvere's fixed point theorem to the untyped λ-calculus. This was achieved by examining the relationship between models of the λ-calculus and the theory of cartesian closed categories and resulted in the observation that the Lawvere's fixed point theorem could be used to derive the first fixed point theorem in the untyped λ-calculus.

An analysis of the limitations within theorem proving have been provided and explained alongside a presentation of modern advancements within type theory that rectify this.

The importance of theorem proving as a mathematical practice has been explained with evidence to suggest its worth. The interwoven historical context behind both theorem proving and Lawvere's fixed point theorem have been examined and the theory that underpins this has been presented.

This dissertation consists of the explanations herein and an online repository of fully mechanised proofs.

Bibliography

[1] Henk P Barendregt. Lambda calculi with types. 1992.

[2] G. Cantor. Über eine Eigenschaft des Inbegriffes aller reellen algebraischen Zahlen. *Crelle's Journal für Mathematik*, 77:258–263, 1874.

[3] G. Cantor. *Üeber eine elementare Frage der Mannigfaltigkeitslehre.* Druck und Verlag von Georg Reimer, 1892.

[4] Alonzo Church. A Set of Postulates for the Foundation of Logic. *Annals of mathematics*, pages 346–366, 1932.

[5] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[6] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. *arXiv preprint arXiv:1611.02108*, 2016.

[7] Catarina Coquand, Thierry Coquand, Ulf Nurell, et al. Agda. *WWW page*, 2000.

[8] Thierry Coquand. The paradox of trees in type theory. *BIT Numerical Mathematics*, 32(1):10–14, 1992.

[9] Haskell B Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584, 1934.

[10] Haskell B Curry. The paradox of kleene and rosser. *Transactions of the American Mathematical Society*, 50(3):454–516, 1941.

[11] Haskell Brooks Curry. Grundlagen der kombinatorischen logik. *American journal of mathematics*, 52(4):789–834, 1930.

[12] Nicolaas Govert de Bruijn. Automath, a language for mathematics. In *Automation of Reasoning*, pages 159–200. Springer, 1983.

[13] Samuel Eilenberg and Saunders MacLane. General theory of natural equivalences. *Transactions of the American Mathematical Society*, 58(2):231–294, 1945.

[14] Samuel Eilenberg and Norman E Steenrod. Axiomatic approach to homology theory. *Proceedings of the National Academy of Sciences of the United States of America*, 31(4):117, 1945.

[15] Gottlob Frege. *Die Grundlagen der Arithmetik: eine logisch mathematische Untersuchung über den Begriff der Zahl.* w. Koebner, 1884.

[16] Dan Frumin and Guillaume Massas. Diagonal Arguments and Lawvere's Theorem, September 2017. Self-published note.

[17] Mario Román García. *Category Theory and Lambda Calculus.* Bachelor's Thesis, Universidad de Granada, Granada, 2018.

[18] Alexandre Grothendieck. Sur quelques points d'algèbre homologique. *Tohoku Mathematical Journal, Second Series*, 9(2):119–183, 1957.

[19] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM (JACM)*, 40(1):143–184, 1993.

[20] D. Hilbert and W. Ackerman. *Grundzüge der theoretischen Logik*. Berlin, J. Springer, 1928.

[21] William A Howard. The formulae-as-types notion of construction.

[22] Frederik Iversen. cat: A formalization of category theory in cubical agda. `https://github.com/fredefox/cat`, 2018.

[23] Daniel M Kan. Adjoint functors. *Transactions of the American Mathematical Society*, 87(2):294–329, 1958.

[24] Stephen C Kleene and J Barkley Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, pages 630–636, 1935.

[25] Stephen Cole Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112(1):727–742, 1936.

[26] Joachim Lambek. Cartesian closed categories and typed λ-calculi. In *LITP Spring School on Theoretical Computer Science*, pages 136–175. Springer, 1985.

[27] F William Lawvere. *Functorial Semantics of Algebraic Theories: And, Some Algebraic Problems in the Context of Functorial Semantics of Algebraic Theories.*

[28] F William Lawvere. An elementary theory of the category of sets. *Proceedings of the National academy of Sciences of the United States of America*, 52(6):1506, 1964.

[29] William F Lawvere. Diagonal Arguments and Cartesian Closed Categories. In *Category theory, homology theory and their applications II*, pages 134–145. Springer, 1969.

[30] Jannis Limperg. cats: Category theory in agda. `https://github.com/JLimperg/cats/tree/master/Cats`, 2019.

[31] Giuseppe Longo and Eugenio Moggi. A category-theoretic characterization of functional completeness. *Theoretical Computer Science*, 70(2):193–211, 1990.

[32] Zhaohui Luo. A unifying theory of dependent types: the schematic approach. In *International Symposium on Logical Foundations of Computer Science*, pages 293–304. Springer, 1992.

[33] Lena Magnusson and Bengt Nordström. The alf proof editor and its proof engine. In *International Workshop on Types for Proofs and Programs*, pages 213–237. Springer, 1993.

[34] Per Martin-Löf and Giovanni Sambin. *Intuitionistic Type Theory*, volume 9. Bibliopolis Naples, 1984.

[35] nLab authors. HomePage. `http://ncatlab.org/nlab/show/HomePage`, May 2019. Revision 276.

[36] nLab authors. Lawvere's fixed point theorem. `http://ncatlab.org/nlab/show/Lawvere%27s%20fixed%20point%20theorem`, May 2019. Revision 10.

[37] Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Citeseer.

[38] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the calculus of constructions. In *International Conference on Mathematical Foundations of Programming Semantics*, pages 209–228. Springer, 1989.

[39] Bertrand Russell. *Principles of Mathematics*. Cambridge University Press, 1903.

[40] SVERKER SÖRLIN. Iasthe institute letter. *Institute for Advanced Study*, 2014.

[41] Thomas Streicher. Investigations into intensional type theory. *Habilitiation Thesis, Ludwig Maximilian Universität*, 1993.

[42] The Coq Development Team. The coq proof assistant, version 8.7.0, October 2017.

[43] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

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

[45] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1925–1927.

[46] Noson S Yanofsky. A universal approach to self-referential paradoxes, incompleteness and fixed points. *Bulletin of Symbolic Logic*, 9(3):362–386, 2003.

Appendix A

The Karoubi Map and Lawvere's Fixed Point Theorem

Below is a failed attempt to see the consequences of lawvere's fixed pont theorem after applying the Karoubi Envelope to the λ-algebra.

As stated earlier every λ-algebra can be turned into a ccc with a reflexive object via the karoubi envelope which turns any additive category into a pseudoabelian category. Let $C = (X, \cdot, k, s)$ be a λ-algebra the karoubi envelope is as follows where $a \circ b = \lambda x.a(bx)$.

$$\begin{aligned}
&\text{Objects : } \{x \in X \mid x \circ x = x\} \\
&\text{Arrows : } \text{Hom}(a,\, b) = \{f \in X \mid b \circ f \cdot a = f\} \\
&\text{Identity : } \text{id}_a = a \\
&\text{Composition : } f \circ g
\end{aligned}$$

This comes from considering C as a monoid which is then categorified as a category with a single object.

Proofs of the validity of all constructions can be found in (Koymans 1982). To show cccsness.

$$\text{Terminal : } t = \lambda xy.y$$

...

The reflexive object is $\mathbf{I}$ as $\mathbf{I}^{\mathbf{I}} = \mathbf{1}$ where $\mathbf{I} = \lambda x.x$ and $\mathbf{1} = \lambda x\,y.x\,y$ where $\mathbf{1}$ is itself the arrow both ways between $\mathbf{1}$ and $\mathbf{I}$ and $\mathbf{1} \circ \mathbf{1} = \text{id}_{\mathbf{1}}$.

Via the earlier corollary, every endomorphism on $\mathbf{I}$ has a fixed point. Endomorphisms on $\mathbf{I}$ are λ-terms, f, such that $\mathbf{I} \circ f \circ \mathbf{I} = f$. Expanding this out

$$\begin{aligned}
\mathbf{I} \circ f \circ \mathbf{I} &= \lambda x.\, \mathbf{I}\,(f\,(\mathbf{I}\,x)) \\
&= \lambda x.\, f\,x
\end{aligned}$$

For this to encompass all (any?) λ-terms our equational theory must include the η-rule. Given that this is a construction is for λ-algebras which need not have η this rains on the hope that Lawvere's fixed point theorem will correspond to the first recursion theorem for the untyped λ-calculus.

A point to $\mathbf{I}$, p, in the karoubi'd category corresponds to λ-terms such that $\mathbf{I} \circ p \circ t = p$

$$\begin{aligned}
\mathbf{I} \circ p \circ t &= p \circ t \\
&= \lambda x.p(tx) \\
&= \lambda x.p(\lambda y.y) \\
&= \lambda x.p\mathbf{I} \\
&= \mathbf{K}(p\mathbf{I})
\end{aligned}$$

i.e some constant function.

With this in mind, Lawvere's fixed point theorem ends up representing a reasonably strange theorem in the untyped λ-calculus i.e. for every λ-term, f, that satisfies extensionality there exists a constant λ-term, u, such that $\lambda x.f(ux) = u$. $u = \mathbf{K}(p\mathbf{I})$ for some p.

$$\begin{aligned} \lambda x.f(ux) &= \lambda x.f((\mathbf{K}(p\mathbf{I}))x) \\ &= \lambda x.f(p\mathbf{I}) \\ &= \mathbf{K}(f(p\mathbf{I})) \\ &= \mathbf{K}(p\mathbf{I}) \end{aligned}$$