A GENTLE INTRODUCTION TO PROOF ASSISTANTS

BY
BENJAMIN CALDWELL

A Thesis

Submitted to the Department of Natural Sciences
New College of Florida
in partial fulfillment of the requirements for the degree
Bachelor of Arts
under the sponsorship of Patrick McDonald

Sarasota, Florida
May 2020

A Gentle Introduction to Proof Assistants

Benjamin Caldwell

New College of Florida, 2020

Abstract

The Curry-Howard Isomorphism is a powerful tool in automated and assisted theorem proving. We provide a development of the material required to understand the Curry-Howard Isomorphism and why it is useful. As a part of this program we provide a resource to quickly learn how to prove mathematical statements using a programming language. After establishing the theory we will look at examples of how the Curry-Howard isomorphism can be useful by looking into the programming language Idris.

Patrick McDonald, Professor of Mathematics
May 12, 2022

Contents

Acknowledgements

I would like to thank all those at New College who made my days brighter. I would also like to thank Professor Stuart Kurtz for introducing me to the material covered here. A last thanks goes to Professor McDonald for being a guiding advisor in both academics and life during my four years at New College. Everything I do from now onward builds upon what you all have taught me.

Chapter 1

Overview

There is a connection between functional programming and mathematical proof known as the Curry-Howard isomorphism. The Curry-Howard isomorphism allows mathematicians to verify their proofs with the assistance of computers. Proof can also be assisted by the computer, rather than just verified. For the mathematician, computer assisted proof can offer some insight to what assumptions or definitions might be needed in order to prove theorems. Computer scientists can use the isomorphism to prove facts about their code, either on paper or with the assistance of computers. With our programming language able to implement the Curry-Howard Isomorphism we get more benefits than just being able to prove statements. We can automatically guarantee that our functions will terminate, preventing infinite loops. Programming languages capable of using the Curry-Howard isomorphism allow programmers to eliminate the possibility of unintended consequences of their code.

Mathematical logic underwent a revolution when Gottlob Frege attempted to write down a formal system describing the logic behind mathematics. Frege tried to build sets as a foundation for mathematics. Unfortunately for Frege, before he could publish his works a paradox was found in his logic by Bertrand Russell. Two solutions to Russell's paradox were formulated. The first was developed by Ernst Zermelo and Abraham Fraenkel and forms the basis for modern set theory. Russell proposed his own solution to the paradox, which helped develop the field of type theory. This split helped solidify that there were multiple approaches to constructing a formal basis for

mathematics. Many logicians tried to develop different foundations for mathematics. Creating a foundation involves formalizing propositions and deduction then developing a basis of how to construct objects as simply as possible. Some logicians, including Moses Schönfinkel, developed ways to deal with propositions simply including development of combinators. Combinators helped lay the groundwork for a functional way of approaching mathematics which helped form the basis for Alonzo Church's later work. Alonzo Church continued work on this functional view of mathematics. Alonzo Church wanted to develop a functional system which could handle proofs and propositions. His goal was to create a language for mathematics based on functions and computation. This would reduce proving a statement to applying functions. From this view, he developed the λ-calculus.

The λ-calculus describes functions as rules for computation. This varies from the normal view of functions as graphs or as ordered pairs of arguments and corresponding value. The untyped λ-calculus accomplishes this goal, however it was shown to be insufficient as a logical basis[1]. Church's solution was to develop a system of types for λ-expressions. From this type system we get the Curry-Howard isomorphism which will describe how we can use λ-calculus as a basis for mathematical logic which will in turn allow us to produce computer verified proofs.

Our principle goal is to explain the development and show an implementation of the Curry-Howard isomorphism. In order to build a common foundation for this discussion we will formulate the language mathematics is built upon, first order logic. Then we will formulate the language of λ-calculus which will give us the basis of any functional programming language. Then we will build the system of type theory. Type theory is a natural way to think. In our day-to-day life we clump things into types, apples are a type of fruit, squares are a type of shape. Type theory captures this notion. When we restrict the λ-calculus using type theory, we will get the Curry-Howard isomorphism, connecting functions and proof. After we discuss the theory, we

[1]The untyped λ calculus is insufficient as it has functions which can not be computed in a finite number of steps. This would be insufficient as our proofs should only have a finite number of steps. The goal is to create a system that mimics proof, and we cannot have infinite proof so this is insufficient.

will look at a practical example in the programming language/proof assistant Idris. Idris exists as both a programming language and a proof assistant, so it will help us illustrate that there are two ways to view functions thanks to the Curry-Howard Isomorphism. We can look at a function as something which generates outputs based on inputs, like a programmer. We can alternatively view a function as a way of constructing an object which satisfies some property, like how a mathematician may view a constructive proof. The properties we are trying to satisfy will be encoded in the type system of our language.

Our goal is to provide a resource for a programmer to learn how to prove facts about their code or for a mathematician to find a helpful tool for proving statements. As such, we will cover all the background we need in this thesis. However, it may be beneficial in the later sections to work alongside the thesis in Idris[2].

[2]Instructions can be found at the start of Chapter 6 for how to run Idris.

Chapter 2

First Order Logic

2.1 Propositional Logic

To begin to talk about encoding mathematical logic into a computer, we must first have some agreed upon foundation for mathematical logic. Taking a reductive view of mathematics, we can say that mathematics is the process of making deductions about some agreed upon collection of axioms. This involves a discussion of propositional and predicate logic, the systems on which formal mathematics and its arguments are built. When developing propositional logic we try to capture the process of logical deduction. As such, we will need to define what a proposition is and describe the processes by which we can prove facts about propositions. The system we will be using is called Gentzen-style Natural Deduction which will follow our natural ideas of reasoning. We begin with the definition of a proposition given by Simon Thompson [10].

Definition 2.1. A *proposition* is either

- A *propositional variable*[1], typically notated by a capital letter.
- A *compound formula* made using one of the following:
  - $(A \wedge B)$, read as A and B.

[1]We will often drop the term "propositional"

- $(A \implies B)$, read as A implies B and meaning if A has a proof, then B has a proof.
- $(A \vee B)$, read as A or B.
- $\bot$, read as False typically, and is a proposition which has no proof.
- $(A \iff B)$, read as A if and only if B, meaning A is true only when B is true, and B is true only when A is true.
- $(\neg A)$, read as not A.

Now that we have our basic object, we will define two types of operations over propositions. They will be called introduction and elimination rules, and will allow us to piece together propositions and take them apart to perform deductions.

Definition 2.2. An *introduction rule* is a way to take multiple propositions as inputs and form a compound formula involving the propositions.

Introduction rules creating compound formulas will give propositions structure rather than just leaving them as single variables.

Definition 2.3. An *elimination rule* is a way to take a compound formula and form one or more propositions.

As introduction rules will allow us to make compound formulas, elimination rules will allow us to take compound formulas apart to get at the statements contained within. With these two types of rules, we will be able to operate on propositions in a way that mirrors our natural idea of deduction.

Definition 2.4 (Assumption Rule)**.** The assumption rule is the first introduction rule, and it states that any propositional formula A can be derived from the assumption of A.

$$\frac{A}{A}$$

This is very straightforward, and has no correlated elimination rule (unlike the other rules we will observe). From this rule, we have the standard way that we will

write these rules. Any propositions found above the line will be our assumptions, and below the line we will find our conclusion through application of some rule. These statements can stack, so we may find a very tall statement with many lines for applying many different rules, this will constitute a proof and should have clear assumptions found at the top with clear applications of rules at each step.

The first compound proposition we will define rules for will be $\wedge$.

Definition 2.5 ($\wedge$ introduction)**.** Given the assumptions A and B, we can form the compound proposition $A \wedge B$.

$$\frac{A \quad B}{A \wedge B}$$

Definition 2.6 ($\wedge$ elimination)**.** Given the assumption $A \wedge B$, we can form either the proposition A, the proposition B, or both propositions A and B for our next layer of deduction.

$$\frac{A \wedge B}{A} \quad \text{or} \quad \frac{A \wedge B}{B} \quad \text{or} \quad \frac{A \wedge B}{A \quad B}$$

These match our natural idea of what "and" means when discussing deduction, as it will give us both statements or just one of our choice. It's important to note that we want the possibility of dropping either side of our $\wedge$ statement, as we want to end up with a single conclusion at the end and may not need both sides of our assumption.

The next equally important conjunction in our logical toolkit is the $\vee$, which can be introduced and eliminated as follows.

Definition 2.7 ($\vee$ introduction)**.** Given the assumption A and any other proposition B, we can form the statement $A \vee B$.

$$\frac{A}{A \vee B}$$

Definition 2.8 ($\vee$ elimination)**.** Given the assumption $A \vee B$, a proof that derives

C from A, and a proof that derives C from B, we can derive C.

$$\dfrac{A \vee B \qquad \begin{array}{c}[A] \\ \vdots \\ C\end{array} \qquad \begin{array}{c}[B] \\ \vdots \\ C\end{array}}{C}$$

The idea of involving entire proofs of C from A may seem like a lot of work, so we wish to also include a shorthand for those proofs.

Definition 2.9 ($\implies$ introduction)**.** Given a proof that derives B from A, we can derive the compound formula $A \implies B$.

$$\dfrac{\begin{array}{c}[A] \\ \vdots \\ B\end{array}}{A \implies B}$$

To eliminate these proofs, we want to use a rule commonly known as "modus ponens".

Definition 2.10 ($\implies$ elimination)**.** Given A and $A \implies B$, we can derive B.

$$\dfrac{A \qquad (A \implies B)}{B}$$

The next rule will deal with $\bot$. If our statement somehow constructs a proof for $\bot$, we have a problem. By definition, it should be lacking proof. If we are left with $\bot$ by itself we must have made a contradiction somewhere in our argument. As such, we want to be able to produce a proof of any statement from $\bot$.

Definition 2.11 ($\bot$ elimination)**.** From $\bot$, we can deduce anything.

$$\dfrac{\bot}{A}$$

Now that we have falsehood absolute, we deal with $\neg$. To construct $\neg A$, we must derive a contradiction from A. Otherwise both A and $\neg A$ are equally valid.

Definition 2.12 ($\neg$ introduction)**.** Given a proof of B from A, and a proof of $\neg B$ from A, then we can derive $\neg A$.

$$\frac{\begin{array}{cc}[A] & [A] \\ \vdots & \vdots \\ B & \neg B\end{array}}{\neg A}$$

To eliminate $\neg$, we must have both the statement A and $\neg A$. This rule matches our idea of the law of contradiction. Similar to $\bot$, we only should run into A and $\neg A$ if something goes terribly wrong, so we want to be able to draw any conclusion from this statement.

Definition 2.13 ($\neg$ elimination)**.** Given the assumption A and $\neg A$, we can derive any proposition B.

$$\frac{A \quad \neg A}{B}$$

Using these rules, we can construct proofs for the two basic axioms of intuitionistic propositional calculus, which are

$$P \implies (Q \implies P) \tag{2.1}$$

which can be read "if P holds, then Q implies P" and

$$(P \implies (Q \implies R)) \implies ((P \implies Q) \implies (P \implies R)) \tag{2.2}$$

which can be read as "if P holding implies that Q implies R, then if P implies Q, P also implies R."

A construction of Axiom 2.1 is given by

$$\frac{\frac{\frac{\frac{P \text{ by supposition.}}{Q \text{ by supposition.}}}{P \text{ by Assumption Rule, Definition 2.4, on } P.}}{Q \implies P \text{ by } \implies \text{ introduction on lines 2 to 3.}}}{P \implies (Q \implies P) \text{ by } \implies \text{ introduction on lines 1 to 4.}}$$

A construction of Axiom 2.2 is given by

$$\frac{\frac{\frac{\frac{\frac{(P \implies (Q \implies R)) \text{ by supposition.}}{(P \implies Q) \text{ by supposition.}}}{P \text{ by supposition.}}}{Q \text{ by } \implies \text{ elimination on } P \implies Q.}}{Q \implies R \text{ by } \implies \text{ elimination on } P \implies (Q \implies R).}}{\frac{(P \implies Q) \implies (Q \implies R) \text{ by } \implies \text{ introduction on lines 2 to 5.}}{(P \implies (Q \implies R)) \implies ((P \implies Q) \implies (Q \implies R)) \text{ by } \implies \text{ introduction on lines 1 to 6.}}}$$

With these two axioms we have built a system for intuitionistic logic, which is a logical system weaker than the one normally used by mathematicians. The key difference is that intuitionistic logic does not have the law of the excluded middle. It is not taken as an axiom that $P \vee \neg P$. This has many far reaching consequences including the lack of double negation. Because intuitionist logic is weaker than classcal logic, any proof in intuitionistic logic is a valid proof in classical logic but the reverse does not hold. In practice, intuitionistic logic cares more about constructing examples from proof. This is why it lacks contradiction as a valid proof technique. Proof by contradiction can tell you something exists, but not tell you what structure it has. Intuitionistic logic is often referred to as constructive logic for this reason.

2.2 Predicate Logic

Predicate logic is an extension of propositional logic which comes into play when we want to start writing propositions about theories that aren't just propositional logic, sometimes called a meta-theory. Propositions will be a tool for us to say certain objects have a property or are equal. These properties and objects are what make up our meta-theory. To allow ourselves to talk about the meta-theory we introduce objects through *terms* and give ourselves the power to talk about all terms or say a term exists through the predicates $\forall$ and $\exists$. This will allow us to talk about things that aren't propositons or predicates. We also include a primitive equality as it is not something we can define otherwise.

First, we will define what we mean by a *function symbol*. This will play a role in the definition of a *term*. This will give us the ability to talk about objects in our meta-theory. Terms can be variables or constants. Constants will be parts of our theory from which we can build other parts. As an example, in elementary number theory built from the Peano axioms, 0 will be a constant and everything else will be constructed by successor functions applied to 0.

Definition 2.14. A *function symbol* is a symbol which has a number of inputs and produces an output from those inputs. Each function symbol has an *arity*, or the number of inputs it takes. A *n-ary* function f_n takes n argument terms $t_1, \ldots, t_n$ in forming the term $f_n(t_1, \ldots, t_n)$. Function symbols represent functions in the typical understanding in mathematics.

Definition 2.15. A *term* is meant to denote objects in whatever theory we apply predicate logic to, and has one of the forms below

- *Individual variables*, or *variables*. We shall use x_1, $x_2, \ldots$ to indicate variables.
- *Individual constants*, or *constants*. We shall use $c_1, c_2, \ldots$ to indicate constants.
- *Composite terms* which are formed by applying function symbols to other terms.

Terms will be useful in proofs in our meta-theory, and will not come up in proofs purely about propositions. Instead, we will typically use variables. Now that we have the basic object of predicate logic, we can start discussing propositions.

Definition 2.16. A n-ary *predicate symbol* $P_n(t_1, \ldots, t_n)$ is a way of writing that some property P holds over the n terms $t_1, \ldots, t_n$.

In arithmetic we can look at $t_1 < t_2$ where $<$ is a predicate symbol with two inputs which expresses that t_1 is less than t_2. Relations in general are predicate symbols. The set theoretic definition of a relation ends up being the property our predicate symbol is meant to represent.

Definition 2.17. A *proposition* comes in one of three forms:

- *Atomic formulas* which are of two forms. The first is

  $$P_n(t_1 \ldots, t_n)$$

  for a predicate symbol P_n. The other atomic formula is the relation equality, we want this one to be a primitive so we define it separately. We take the class of formulas

  $$t_1 = t_2$$

  where t_1 and t_2 are terms to be our equality formulas.

- *Propositional combinations* which are combinations of formulas using the propositional connectives in **Definition 2.1**.

- *Quantified formulas* which are formulas involving $\forall$ and $\exists$, which come in the form

  $$\forall x.P \quad \text{and} \quad \exists x.Q.$$

  for arbitrary propositions P and Q.

Propositions defined this way give us access to arbitrary relations constructed from function symbols, basic equality, constants, and variables. We also have all of

the same propositional connectives from the last section and can use all the same introduction and elimination rules from Section 1.

Two predicates with different variable names should be treated as the same, so we have to find a way to codify this. We first have to acknowledge that not all variables are created equally. If a variable occurs in a $\forall$ or $\exists$ statement, it will have a different contextual meaning later on. We cannot switch one occurrence of this variable without switching all of them. Therefore, we call these variables bound. Any variable not bound by a $\forall$ or $\exists$ will therefore be a free variable.

Definition 2.18. An occurrence of a variable x within a sub-formula $\forall x.P$ or $\exists x.P$ is *bound*, and all other occurrences are *free*. We say that a variable x occurs free in a formula A if some occurrence of x is free in A. A variable x will be bound by the innermost enclosing quantifier if one exists.

Example 1. In the formula

$$\forall y.((x > y) \wedge (\forall x.(P(x) \implies P(y))) \wedge Q(y, x))$$

the second occurrence of x is bound, while the first and third occurrence are free.

Now that we've acknowledged the differences in types of variables, we want to be able to switch terms arbitrarily. To do this we will recursively define *term substitution*.

Definition 2.19 (Term substitution)**.** The *substitution* $s[t/x]$ where s, t are terms and x is a variable is defined as

- $x[t/x] \equiv t$ and for a variable $y \neq x$, $y[t/x] \equiv y$.
- For composite terms

$$(f_{n,m}(t_1, \ldots, t_n))[t/x] \equiv f_{n,m}(t_1[t/x], \ldots, t_n[t/x])$$

The idea for term substitution is that we're replacing all occurrences of x with t. This allows us to change arbitrary variables for more complex expressions or just change our variables around.

Definition 2.20 (Formula substitution)**.** We define substitution of a term t for a term x in an a arbitrary formula A, written $A[t/x]$ as follows:

- If A is an atomic formula of either the form,

$$(P_{n,m}(t_1, \ldots, t_n))[t/x] \equiv P_{n,m}(t_1[t/x], \ldots, t_n[t/x])$$

  or of the form

$$(t_1 = t_2)[t/x] \equiv (t_1[t/x] = t_2[t/x])$$

- If we have a propositional combination, substitution will commute with it so that

$$(A \wedge B)[t/x] \equiv (A[t/x] \wedge B[t/x])$$

  and similarly for the other combinations.

With these two substitutions, it's important to note that we are replacing variables with terms. This may seem a strange thing to define, but it's something we take for granted and must be given some foundation. To illustrate this, we can look at the formula

$$\exists z.(z > x).$$

In this formula, we intuitively understand that as x is free we (granted the statement is true) can replace x with anything. Term substitution is the formal notion which allows us to do this. It's important to note that when we substitute, we can not substitute with a variable that is bound. In the above example, we would not substitute x for z, as it is already bound.

With substitution out of the way, all we have to do is define our quantifier rules to match how we would like to use them. Once we have that, we can prove statements using quantifier logic and have a fully fleshed out system of doing mathematics.

Before we write out the rules, we will observe that when we write out mathematical formulas with free variables, we typically assume that they hold for all instantiations of the variable. A mathematician knows that $\sin^2 x + \cos^2 x = 1$ holds for all values of

x in the reals, and can be rewritten as $\forall x. \sin^2 x + \cos^2 x = 1$, as long as we understand we are working in the real numbers. This idea generalizes to all uses of formulas, and often times $\forall$ is left off of propositions as we have a mutual understanding it can be introduced over any variable. This inspires our $\forall$ introduction rule to be as follows.

Definition 2.21 ($\forall$ introduction)**.**

$$\frac{A}{\forall x.A}$$

On the elimination side, when we say something holds for every single case, we can replace the variable with a specific term to get rid of the universal quantifier. Matching this intuition is our $\forall$ elimination rule.

Definition 2.22 ($\forall$ elimination)**.**

$$\frac{\forall x.A(x)}{A(t)}$$

On the other side, $\exists$ is a little trickier to deal with. Introduction is straightforward, if we have $A(t)$ for some term t, we can say that $\exists x.A(x)$ for a variable x. This allows us to step away from the specific case. This is used mostly in constructive proof where a specific result is created, but we don't actually need to know the exact result. The common example is $\forall t, v \in \mathbb{R}/\mathbb{Q}.t < v.\exists q \in \mathbb{Q}.t < q < v$, or for every two irrationals there is a rational in between them. When using this theorem, we just care about the fact that the term in between the irrational numbers is rational, not it's specific construction. However, to prove this statement, we typically construct such a rational then use that to say $\exists$. This inspires our $\exists$ rule.

Definition 2.23 ($\exists$ introduction)**.**

$$\frac{A(t)}{\exists x.A(x)}$$

$\exists$ elimination is far more complicated. In order to use it, we need to show that for a variable x if $\exists x.A(x)$ and $A \implies B$ where x is not free anywhere in the proof of B or B itself, then we can use our $\exists$ elimination to prove that B exists.

Definition 2.24 ($\exists$ elimination)**.**

$$\frac{\exists x.A(x) \quad A \implies B}{B}$$

where x is not free in B or in any of the assumptions of the proof of B, except for A itself in which it may be free.

These two quantifiers have a dual nature to them and how they handle both free and bound variables. While $\forall$ gives a variable an arbitrary context, meaning we can replace it with anything, $\exists$ gives us a specific object with a property. If we so wish, we could view $\forall x.A(x)$ and $\exists x.A(x)$ as extensions of $\wedge$ and $\vee$ to the infinite cases. We can only say $\forall x.A(x)$ if we could also say (for every term in our meta-theory) $A(a) \wedge A(b) \wedge A(c) \wedge \ldots$. Similarly, we can only say $\exists x.A(x)$ if we could also say (for every term in our meta-theory) $A(a) \vee A(b) \vee A(c) \vee \ldots$.

What we have constructed here is a system of formal logic which does not use axioms, but instead uses rewrite rules which tell us operations we can perform on logical statements. This gives us a slightly different view of logic, but each of the introduction and elimination rules given above correspond to axioms of first order logic.

In the next section, we will construct the λ-calculus. It will be beneficial to try and note some of the similarities between how we constructed propositional and predicate logic. These two things will be more clearly illustrated as being connected later. This connection is why we chose to use rewrite rules rather than axioms for our first order logic system. As we continue, try to think of rewrite rules as functions over propositions, taking one proposition to another.

Chapter 3

λ-Calculus

The λ-calculus was first developed by Alonso Church as a way of studying functions, independent of their notation in various fields of mathematics [5]. The λ-calculus frees the idea of functions from the set theory definition by reducing computation to evaluating a λ-expression. The λ-calculus is a formal language that allows us to write arbitrary functions with only a few symbols. In our λ-expressions, the input shall be known as the *argument* and the output shall be known as the *value* of the function at the argument.

When we apply our λ-expressions to some meta-theory, we gain a lot more power than just restricting ourselves to working within the λ-calculus alone. For now, we will ignore any possible meta-theories and instead try to focus on the pure λ-calculus.

As the λ-calculus is a form of computation, many computer languages have been created using λ-expressions as a backbone. Such languages are known as functional programming languages, a popular example being Haskell. We wish to view a λ-expression as a primitive programming language which we have to evaluate ourselves.

3.1 Untyped λ-calculus

The main benefit of the λ-calculus is that it provides a skeleton upon which to construct functions with a simple syntax. The λ-calculus can be viewed as a programming language, as we can view λ-expressions as simple functions we can evaluate by hand

or with a computer. We must first define how to construct a λ-expression.

Definition 3.1 (λ-expression)**.** There are three types of λ-expressions, given by

1. **Variables** notated by v_0, v_1, $v_2 \ldots$. These are the atoms of λ-expressions and can represent arbitrary λ-expressions or arbitrary objects in some meta-theory we are applying the λ-calculus to.

2. **Abstraction** Notated by $\lambda x.e$ where e is an arbitrary λ-expression. This gives us a function which returns the value e when given an argument x. The value of e may depend on the input x.

3. **Application** Notated by $e_1 e_2$ where e_1 and e_2 are arbitrary λ-expressions. This gives us the application of e_1 to the argument e_2.

This is a verbose definition for λ-expressions. For a definition easier to prove statements over, we have an equivalent, terse, inductive definition.

Definition 3.2. λ-expressions are words over the alphabet of

$v_0, v_1, v_2, \ldots$ variables,

λ abstractor,

. dot,

( ) parentheses.

we define the set of all λ-expressions Λ as follows

(i) $x \in \Lambda$, where x is a variable,

(ii) $M \in \Lambda \implies (M) \in \Lambda$,

(ii) $M \in \Lambda \implies \lambda x.M \in \Lambda$,

(iii) $M \in \Lambda$ and $N \in \Lambda \implies MN \in \Lambda$.

With the λ-calculus we also include parentheses, which will tell us what to evaluate first in our λ-expressions. Parentheses will give us a clear order of evaluation for every expression and we will always evaluate the lowest level of our parentheses first. To make our λ-expressions easier to write, we want to hide as many parentheses as possible. We will establish some standard rules for writing λ-expressions.

Definition 3.3. There are three conventions for writing λ-expressions which we will adopt for writing λ-expressions.

1. λ-expression application is applied before abstraction, so $\lambda x.xe_1 = (\lambda x.(xe_1))$.
2. λ-expression application associates to the left, so $e_1e_2e_3 = (e_1e_2)e_3$.
3. λ-expression abstraction will parenthesize everything after it, so

$$\lambda x_1.\lambda x_2.\ldots.\lambda x_n.e = \lambda x_1.(\lambda x_2.\ldots.(\lambda x_n.e)\ldots)$$

These conventions establish a syntax for our lambda calculus. With our syntax established and λ-expressions defined we can start to discuss variables place and behavior within functions.

Definition 3.4. All occurrences of a variable x in e are *bound* in the expression $\lambda x.e$. The λx is called the *binding instance* of x. Any variables in e which are not bound are known as *free* variables.

A bound variable is intuitively just a variable which can be changed by an input on our λ-expression. Any variables which are not able to be changed via input are free, as they can be freely changed to arbitrary expressions without breaking our expression's ability to be evaluated.

Similar to variable substitution in propositions and predicates, we would like to be able to substitute variables arbitrarily in our λ-expressions. This will allow us to show that two λ-expressions which are identical up to variables will actually be identical λ-expressions in terms of their evaluation.

Definition 3.5. The *subsitution* of a free variable x in a λ-expression e with some expression f is written $e[f/x]$ and is defined as follows,

- $x[f/x] = f$ and for a variable $y \neq x$, $y[f/x] = y$.
- Given a λ-expression which is at the highest layer an application, we split the substitution over the application.

$$(e_1 e_2)[f/x] = (e_1[f/x] e_2[f/x])$$

- If e is an abstraction binding x, $e = \lambda x.g$, then $e[f/x] = e$. If $y \neq x$, and $e = \lambda y.g$, then we must consider two cases.
  - If y is not free in f, $e[f/x] = \lambda y.g[f/x]$
  - If y is free in f, we must pick a variable z which does not occur in either f or g, then we can write

$$e[f/x] = \lambda z.(g[z/y][f/x])$$

- In general, if x is not free in e, then $e[f/x] = e$

Using substitution we can finally define functional application. We will call this application β-reduction. When we define reductions they are computations which take our λ-expressions and make them "simpler" in some sense. We call them computations as we will have to by hand or by computer rewrite our expressions in a simpler form.

Definition 3.6 (β-reduction)**.** The *rule of β-reduction* states that any expression of the form $(\lambda x.e)f$, we can reduce (or β-reduce) the application as follows.

$$(\lambda x.e)f \rightarrow_\beta e[f/x]$$

To evaluate a functional application we just replace the argument with the value of the input.

Assuming an understanding of set theory, we can view β-recuction as a relation over the set of λ-expressions. We could also view β-reduction as a predicate symbol which deals with the theory of λ expressions. From a foundations perspective the second way to view β-reduction is more satisfying. However for the simplicity of proof we will treat β-reduction as a relation over the set of λ-expressions.

Definition 3.7. In set theory, a *relation* R over a set A is a subset $R \subset A \times A$. We will say for $a, b \in A$ that $a \sim_R b \iff (a, b) \in R$.

β reduction is actually a special type of relation called a *reduction*.

Definition 3.8. A relation $\to$ is *compatible* (with the λ-calculus operations) if

$$M \to M' \implies ZM \to ZM', MZ \to M'Z, \text{ and } \lambda x.M \to \lambda x.M'$$

Definition 3.9 (reduction)**.** A reduction is a relation which is compatible, reflexive, and transitive.

We cannot apply β-reduction to arbitrary λ-expressions. To apply β-reduction, we need to have λ-abstraction and an expression to apply our abstraction to. Expressions which have both of these will be referred to as β-redexes and we will be able to immediately reduce them into smaller λ-expressions.

Definition 3.10 ($(\beta$-)redex)**.** A *redex* or *reducible expression* is any λ-expression to which the above β-reduction rule can be immediately applied. For example, $\lambda x.((\lambda y.y)z)$ is not a redex as it has nothing apply to, but the nested expression $(\lambda y.y)z$ is a redex, as we can β-reduce: $(\lambda y.y)z \to_\beta z$.

The expressions we reach by β-reduction will be known as reducts. It is important to note that some λ-expressions will have many reducts and many different paths for reducts to take.

Definition 3.11 (reduct)**.** For λ-expressions f and e, we say f is a *reduct* of e if there exists a sequence of zero or more β-reductions so that

$$e = e_0 \to_\beta \cdots \to_\beta e_n = f,$$

giving us a sequence of reduction steps to get to f from e. We write $e \twoheadrightarrow_\beta f$ for "f is a reduct of e".

With the defining of β-reduction, redexes, and reducts we have constructed a simple form of computation on λ-expressions. We will later attempt to observe how these different reduct paths converge, but first we must establish normal form.

Definition 3.12 (Normal form)**.** A λ-expression e is in *normal form* if it contains no redexes.

Normal form is simply a way of showing we can do no further computations within or on an expression. Normal form gives us a minimal way to write our function with respect to β-reduction. There are two other normal form type expressions, the first of which is *Head normal form*. Head normal form are functions which we can find reducts within and have a λ-abstraction at the top leftmost level.

Definition 3.13 (Head normal form)**.** A λ-expression e is in *head normal form* if it is of the form

$$\lambda x_1 \lambda x_2 \ldots \lambda x_n . y e_1 \ldots e_m$$

where y is a variable, $e_1, \ldots, e_m$ are arbitrary expressions and n and m are greater than or equal to zero.

Similar to head normal form is weak head normal form without the λ-abstractions

Definition 3.14 (Weak head normal form)**.** A λ-expression e is in *weak head normal form* if it is of the form

$$y e_1 \ldots e_m$$

where y is a variable, $e_1, \ldots, e_m$ are arbitrary expressions and m is greater than or equal to zero.

In weak head normal form, our top left most expression must be a variable. We can convert from weak head normal form to head normal form by performing λ-abstraction.

Definition 3.15. We say that e is a *normal form* (similarly for a *head normal form* or *weak head normal form*) of f if $f \twoheadrightarrow_\beta e$ and e is in normal form.

Definition 3.16 (Structural Induction)**.** *Structural induction* states that to prove the result $P(e)$ for all λ-expressions e it is sufficient

- to prove $P(x)$ for all variables x,
- to prove $P(ef)$ assuming that $P(e)$ and $P(f)$ hold,
- to prove $P(\lambda x.e)$ assuming $P(e)$ holds.

We need the tool of structural induction to prove things about λ-expressions. We will take structural induction as an axiom. We will use it to prove the Church-Rosser theorem (Theorem 3.6).

Definition 3.17. A relation $\rightarrow$ satisfies the *diamond property* if $A \rightarrow B$ and $A \rightarrow C$ then $\exists D$ such that $B \rightarrow D$ and $C \rightarrow D$.

When dealing with terms like β-reduction, having the diamond property will give us a sense of regularity in that the order we do reductions does not matter, as we can always find some further term to which to reduce even if our reductions diverge.

Definition 3.18. The transitive closure T of a relation R is the smallest relation such that $R \subset T$ and T is a transitive relation.

We can use the transitive closure to talk about doing multiple β-reductions in a row. The transitive closure of $\rightarrow_\beta$, given by $\twoheadrightarrow_\beta$ will be all possible λ-expressions we can β-reduce to. In the next proposition, we will show that taking this transitive closure will preserve the diamond property.

Proposition 3.1. *Let $\rightarrow$ be a binary relation on a set and let $\twoheadrightarrow$ be its transitive closure. Then if $\rightarrow$ satisfies the diamond property, so does $\twoheadrightarrow$.*

Proof. Let $x \twoheadrightarrow y$ and $x \twoheadrightarrow z$. Then as $\twoheadrightarrow$ is the transitive closure of $\rightarrow$, we can find ordered sets $a_{i\,i=1}^{n}$ and $b_{j\,j=1}^{m}$ such that $x \rightarrow a_1, x \rightarrow b_1, a_i \rightarrow a_{i+1}, b_j \rightarrow b_{j+1}, a_n = y$

and $b_m = z$. These are the sequences of relations under $\to$ which are used to get $x \twoheadrightarrow y$ and $x \twoheadrightarrow z$. We will prove that $\twoheadrightarrow$ satisfies the diamond property by induction.

Let $n = 1$.

Furthermore, let $m = 1$. As $n = 1$, $m = 1$ we know $x \to a_1 = y$ and $x \to b_1 = z$. But $\to$ satisfies the diamond property, so we can find a c such that $y \to c$ and $z \to c$. Therefore for $m = 1$, the diamond property is satisfied.

Suppose this holds for $m = k$. Looking at the $k + 1$ case, we can see $x \twoheadrightarrow b_k$ and $x \twoheadrightarrow a_1$ with a sequence of length k and 1 respectively. By our inductive hypothesis, we can find a c such that $b_k \twoheadrightarrow c'$ and $a_1 \twoheadrightarrow c'$. Then $b_k \to b_{k+1}$ and $b_k \to c'$, so there exists a c such that $z = b_{k+1} \to c$ and $c' \to c$, so $y \twoheadrightarrow c$ and $x \twoheadrightarrow c$. Therefore if this holds for the k case, then it holds for the $k + 1$ case.

Suppose this holds for sequences of length $n = l$.

Furthermore, let $m = 1$. By the same proof above for the $m = k$ and $n = 1$ case, we can show that the $l + 1$ case satisfies the diamond property.

Let the $m = k$ case satisfy the diamond property as well, and we attempt to show that the $l + 1$ case satisfies the diamond property. As $x \to a_1$ and $x \to b_1$, there exists a c' such that $a_1 \to c'$ and $b_1 \to c$. As $a_1 \twoheadrightarrow a_{l+1}$ with a sequence length of l, and $b_1 \twoheadrightarrow a_{k+1}$ with a sequence length of k, by our hypothesis we can find a g such that $c' \twoheadrightarrow g$ with sequence length l and a h such that $c' \twoheadrightarrow h$ with sequence length k. By the inductive hypothesis, we can find a c such that $h \twoheadrightarrow c$ and $g \twoheadrightarrow c$, but $a_{l+1} \twoheadrightarrow h \twoheadrightarrow c$ and $b_{k+1} \twoheadrightarrow g \twoheadrightarrow c$. Therefore, $\exists c$ such that $y \twoheadrightarrow c$ and $z \twoheadrightarrow c$.

Therefore by the principle of induction, $x \twoheadrightarrow y$ and $x \twoheadrightarrow z \implies \exists c : y \twoheadrightarrow c$ and $z \twoheadrightarrow c$. □

Definition 3.19. Define a binary relation $\to_1$ on the set of all λ-expressions inductively as follows.

$M \to_1 M$;

$M \to_1 M' \implies \lambda x.M \to_1 \lambda x.M'$;

$M \to_1 M', N \to_1 N' \implies MN \to_1 M'N'$;

$$M \to_1 M', N \to_1 N' \implies (\lambda x.M)N \to_1 M'[x/N'].$$

We are defining this relation so it is slightly stronger than $\to_\beta$ but weaker than $\twoheadrightarrow_\beta$. Meaning that if $M \to_\beta M'$ then $M \to_1 M'$ but if $N \twoheadrightarrow_\beta N'$ it might not be the case that $N \to_1 N'$. It will give us something easier to prove statements about, as proving things using $\to_\beta$ we care about the structure of the actual expressions.

Lemma 3.2. *[1] If $M \to_1 M'$ and $N \to_1 N'$, then $M[x/N] \to_1 M'[x/N']$.*

Proof. By induction on the definition of $M \to_1 M'$.

Case 1. $M \to_1 M'$ is $M \to M$. Then one has to show $M[x/N] \to_1 M[x/N']$. This is done by induction on the structure of M.

Subcase 1. $M = x$. Then $x[x/N] = N \to_1 N' = x[x/N']$, as required.

Subcase 2. $M = y$. Then $y[x/N] = y \to_1 y = y[x/N']$, as required.

Subcase 3. $M = PQ$, supposing $P[x/N] \to_1 P[x/N']$ and $Q[x/N] \to_1 Q[x/N']$. Then $PQ[x/N] = P[x/N]Q[x/N] \to_1 P[x/N']Q[x/N'] = PQ[x/N']$.

Subcase 4. $M = \lambda y.P$. Then $(\lambda y.P)[x/N] = \lambda x.(\lambda y.P)N$ which as $N \to_1 N'$ and $\lambda y.P \to_1 \lambda y.P$, $\lambda x(\lambda y.P)N \to_1 \lambda x(\lambda y.P)N' = (\lambda y.P)[x/N']$ as required.

Case 2. $M \to_1 M'$ is $\lambda y.P \to_1 \lambda y.P'$ and is a direct consequence of $P \to_1 P$. As $N \to_1 N'$, we have that $P[x/N] \to_1 P'[x/N']$. But then $\lambda y.P[x/N] \to_1 \lambda y.P'[x/N']$.

Case 3. $M \to_1 M'$ is $PQ \to_1 P'Q'$ and is a direct consequence of $P \to_1 P'$ and $Q \to_1 Q'$. Then $M[x/N] = P[x/N]Q[x/N] \to_1 P'[x/N']Q'[x/N'] = M'[x/N']$.

Case 4. $M \to_1 M'$ is $(\lambda y.P)Q \to_1 P'[y/Q']$ and is a direct consequence of $P \to_1 P', Q \to_1 Q'$. Then $M[x/N] = (\lambda y.P[x/N])(Q[x/N]) \to_1 P'[x/N'][y/Q'[x/N']] = P'[y/Q'][x/N'] = M'[x/N']$. □

The above just tells us that $\to_1$ will respect variable substitution by operating on both the substitution N and the statement we are substituting into, M.

Lemma 3.3. *(i) $\lambda x.M \to_1 N$ implies $N = \lambda x.M'$ with $M \to_1 M'$.*

(ii) $MN \to_1 L$ implies either

(a) $L = M'N'$ with $M \to_1 M', N \to_1 N'$.

(b) $M = \lambda x.P, L = P'$

The proof of this statement is omitted and can be found in Barendegt's *Lambda Calculus* [1]. This lemma tells us that λ-abstraction remains unaffected by our relation $\to_1$.

Lemma 3.4. $\to_1$ *satisfies the diamond property.*

Proof. [1] We will show by induction on the definition of $M \to_1 M_1$ it will be shown for all $M \to_1 M_2$ there is an M_3 such that $M_1 \to_1 M_3, M_2 \to_1 M_3$.

Case 1. $M \to_1 M_1$ because $M = M_1$, then we can take $M_3 = M_2$.

Case 2. $M \to_1 M_1$ is $(\lambda x.P)Q \to_1 P'[x/Q']$ and is a consequence of $P \to_1 P', Q \to_1 Q'$. By Lemma 3.3, we can make two subcases.

Subcase 2.1. $M_2 = (\lambda x.P'')Q''$ with $P \to_1 P''$, $Q \to_1 Q''$. By the inductive hypotesis, there are terms P''' such that $P' \to_1 P''', P'' \to_1 P'''$ and similarly for the Q's. By Lemma 3.1, one can take $M_3 = P'''[x/Q''']$.

Subcase 2.2. $M_2 = P''[x/Q'']$ with $P \to_1 P''$ and $Q \to_1 Q''$. Then by the inductive hypothesis, we can again take $M_3 = P'''[x/Q''']$.

Case 3. $M \to_1 M_1$ is $PQ \to_1 P'Q'$ and is a direct consequence of $P \to_1 P'$ and $Q \to_1 Q'$. Again there are two subcases.

Subcase 3.1. $M_2 = P''Q''$ with $P \to_1 P'', Q \to_1 Q''$. Then, using the induction hypothesis we can take $M_3 = P'''Q'''$.

Subcase 3.2. $P = (\lambda x.P_1), M_2 = P_1''[x/Q'']$ and $P_1 \to_1 P_1'', Q \to_1 Q''$. By Lemma 3.3 one has $P' = \lambda x.P_1'$ with $P_1 \to_1 P_1'$. Using the inductive hypothesis, take $M_3 = {P''}_1'[x/Q''']$.

Case 4. $M \to_1 M_1$ is $\lambda x.P \to_1 \lambda x.P'$ and is a direct consequence of $P \to P'$. Then $M_2 = \lambda x.P''$. By the inductive hypothesis, we can take $M_3 = \lambda x.P'''$.

□

We get that $\to_1$ satisfies the diamond property far easier than if we were to try to prove this for $\to_\beta$. Using some clever trickery, we can then show this property holds for $\twoheadrightarrow_\beta$ with the help of our next lemma.

Lemma 3.5. *$\twoheadrightarrow_\beta$ is the transitive closure of $\rightarrow_1$.*

Proof. $\twoheadrightarrow_\beta$ is certainly the transitive closure of $\rightarrow_\beta$.

Note that if $M \rightarrow_\beta N$, then $M \rightarrow_1 N$ and if $M \rightarrow_1 N$ then $M \twoheadrightarrow_\beta N$. Therefore as $\twoheadrightarrow_\beta$ is the transitive closure of $\rightarrow_\beta$, it must also be the transitive closure of $\rightarrow_1$.

□

Now we can tackle one of the most significant theorems of basic λ-calculus. The Church-Rosser theorem is important as it will give us the property that any function with a normal form will always end after a finite number of computations, as it can only have a finite number of β reductions. We will then restrict ourselves to look at functions with normal forms.

Theorem 3.6 (Church-Rosser)**.** *β-reduction satisfies the diamond property.*

Proof. This follows directly from Proposition 3.1, Lemma 3.5, and Lemma 3.4.

□

Theorem 3.7 (Normalizaton)**.** *If a term has a normal form, then it is unique.*

Proof. Let M be a term with a normal form N.

Suppose N' is a normal form of M.

Then by the Church-Rosser theorem, there exists a P such that $N \twoheadrightarrow_\beta P$ and $N' \twoheadrightarrow_\beta P$.

As N, N' are both normal, this means $N = P$ and $N' = P$ so $N' = N$.

Therefore, all normal forms are unique. □

This statement does not hold for head normal or weak head normal form. Unfortunately our system of λ-calculus is too broad, and we cannot prove that terms will always have a normal form. In particular, the term

$$(\lambda x.(xx))(\lambda x.(xx))$$

has no normal form, as after you perform the only beta reduction possible, we end up with

$$(\lambda x.(xx))(\lambda x.(xx)) \rightarrow_\beta (\lambda x.(xx))(\lambda x.(xx))$$

Thankfully, there is a solution to this problem. The theory of types will allow us to restrict our λ-expressions to only expressions which have normal forms. This restriction will allow us to draw a connection between λ-expressions and propositions with the *Curry-Howard Isomorphism*.

Chapter 4

Type Theory

4.1 Type Theory

The basics of type theory begin with how we construct Types. The idea behind a Type is tied closely with construction and computation. If two objects are the same type, they should be constructed from the same atomic elements. Their construction may vary, but they are at the base the same. If two objects are the same type, we also expect that a function which operates on one object should also operate on the other. This will allow us to define our functions based on the constructions of the type of input they expect. From this basis we will build our theory of types.

Definition 4.1. A *Type* is either

- A *type variable*, or just variable, typically notated by a capital letter.
- A *compound type* made using a *type constructor* on a type variable.

For a given compound type, we will create construction and elimination rules which will be similar to our propositions, and we will create a type system for λ-expressions. To do this, we will need to add evaluations. There are three kinds of evaluations we care about.

1. x is of type T, written $x : T$.
2. T_1 and T_2 are the same type.

3. x and y are of the same type.

Definition 4.2 (Assumption rule)**.** The assumption rule states that if we are given that A is a type, we can derive that A is a type.

$$\frac{A \text{ is a type}}{A \text{ is a type}}$$

Our type system is meant to describe a way of restricting functions, so the first new type we define will be the type of functions.

$$\frac{A, B \text{ are types.}}{A \to B \text{ is a type.}}$$

We then need to discuss how we construct a *value* of type $A \to B$ from values of type A and B. A value will just be an object that we can work with which has the type $A \to B$. This is similar to introduction from propositional logic.

$$\frac{A \text{ is a type, for any } a : A,\ e[x/a] : B}{(\lambda x.e) : A \to B.}$$

We will also have some type of deconstructor for values of our types. For propositional logic, we had the type deconstructor of modus ponens. The corresponding idea is functional application, and we can define a λ-expression that corresponds to it. We will call this expression *ponens*.

$$ponens : (A \to B) \to A \to B$$

$$ponens\ f\ x = f(x)$$

Our possible inputs for the function mirror our assumptions for $\implies$ elimination, Definition 2.10, where we had to have $A \implies B$ and A, which correspond to $A \to B$ and A. With this definition, (.) ends up just being functional application. This gives us a way of viewing propositional logic and proofs as just applying functions to statements to produce new statements. We will explore this here and culminate

everything with the Curry-Howard Isomorphism. We will try and define types which correspond to each of our propositional connectives in Definition 2.1.

The first connective we approach will be $\wedge$. First we need to define a type and a way to construct values of the type.

$$\frac{A, B \text{ are types.}}{(A, B) \text{ is a type}}$$

$$\frac{a : A, \quad b : B}{(a, b) : (A, B)}$$

this will allow us to create pairs. This mirrors $\wedge$ introduction, allowing us to make one proposition or type from two.

When we want to construct objects of our type (A, B), we can accomplish this using the λ calculus. Let's define a function in the λ calculus called $mkPair$. $mkPair$ will be a function from A to B to (A, B), seeing as it takes an object from A and B to make a pair in (A, B).

$$mkPair : A \to B \to (A, B)$$

$$mkPair = (\lambda a(\lambda b(a, b)))$$

We can now define a way to deconstruct our type (A, B) using the λ-calculus. We will write two functions, fst and snd.

$$fst : (A, B) \to A$$

$$fst = (\lambda(a, b).a)$$

$$snd : (A, B) \to B$$

$$snd = (\lambda(a, b).b)$$

Sneakily, we have done something called *pattern matching* when we set our variable

to (a, b). When we know how our types are constructed, we can choose to write out functions based on the topmost type constructor. This is called case splitting over the input. This will allow us to create more concise function definitions and give us more information about what our input looks like to decide how to construct our output. Following this process to define functions will allow our types to determine the structure of our function and uses types for our computations. In fst and snd above we're using the structure (A, B) to define how our function operates. In our λ-abstraction, we unwrap the top $Pair$ constructor to get at both a and b. This is how types can affect our actual computation, by telling us what our inputs will look like.

Notice that these functions can also operate on (A, B) as a type to return our types A and B. If A, B correspond to propositions under $mkPair, fst$, and snd, then the natural question is what do values of type A and B correspond to. This correspondence will be explored more in depth, but the natural correspondence is to view these values as proofs of the propositions the types correspond to.

We can define new types corresponding to each of our rules for propositions, although some are more complex than others. In particular $\vee$ corresponds to a type called $Either\ A\ B$.

$$\frac{A, B \text{ are types.}}{Either\ A\ B \text{ is a type.}}$$

This type definition looks no different from our definition for (A, B) as it is just a way of combining two different types, but it differs in how we actually create values of type $Either\ A\ B$.

$$\frac{a : A, \quad B \text{ is a type.}}{Left\ a : Either\ A\ B.}$$

$$\frac{b : B, \quad A \text{ is a type.}}{Right\ a : Either\ A\ B.}$$

We will define a type deconstructor for $Either$, called $either$.

$$either : Either\ A\ B \to (A \to C) \to (B \to C) \to C$$

$$either\ (Left\ a)\ f\ g = f\ a$$

$$either\ (Right\ b)\ f\ g = g\ a$$

We're using a more advanced form of pattern matching here. We write two functions, one to operate on the case of $Left$ and one to operate on the case of $Right$. Much like we can look at our value and discern between the two, a computer can look at the value and discern which function to use. From this idea, we get a simple function which mirrors how we operate on $\vee$ propositions. f and g correspond to proofs of C from A and B respectively; In $\vee$ elimination (Definition 2.8, we required that we had proofs of C in order to prove C from $A \vee B$.

We now have the three most important propositional logic tools (and one predicate logic tool) built in our type theory. From our initial system, we don't have a way to discuss $\neg$ directly. This is not a problem. In our propositional logic system, we could easily have used $\neg A \equiv A \implies \bot$. We will use the equivalent in our typed system, giving us a new type, $Not; A$.

First we will define a type which corresponds to $\bot$.

$$\frac{}{Void \text{ is a type.}}$$

Much like $\bot$ has no proof, we want $Void$ to have no values of its type so we will not give a way to construct $Void$. Instead, we will have a deconstructor for $Void$ which can produce an object of any type. We will call this deconstructor $absurd$.

$$\frac{A \text{ is a type.} \quad x : Void}{absurd : Void \to A \quad absurd\ x : A}$$

Now we can define our $Not\ A$ from $Void$.

$$\frac{A \to Void}{Not\ A}$$

This is just a renaming of $A \to Void$, so constructing values of $Not\ A$ will be the same as constructing values of $A \to Void$.

This is an interesting quality to note, and it has to do with *constructive mathematics*. This *absurd* function corresponds to the principle of contradiction. While this is widely accepted as something we can use, it does not give us any idea of what the output looks like. This creates trouble for our functional system, as any output we could produce by a λ-expression has a function or formula for how it is constructed. Therefore if we want to use the principle of contradiction to prove things, we overstep the capabilities of our λ-expressions to represent proofs. This is okay if we introduce *absurd*, but is a fair warning that we should avoid proof by contradiction and instead work on constructing our proofs. We can still use *absurd* to verify proof, but it will not give us a construction of the final proof. This is because absurd can not be written as a function in the λ-calculus so we cannot compute an output from it, we just use type verification instead.

We will now quickly construct the function that corresponds to our $\neg$ introduction.

$$lnotIntro : (A \to B) \to (A \to (B \to Void)) \to (A \to Void)$$

$$lnotIntro\ f\ g\ = (\lambda a\,.\,(g\ a)(f\ a))$$

In $\neg$ introduction (Definition 2.12) we required that $A \implies B$ and $A \implies \neg B$ to get $\neg A$. We have left the type unwrapped a bit so it was clearer what was happening, but we could easily have written it as

$$lnotIntro : (A \to B) \to (A \to Not\ B) \to Not\ A$$

which is easier to draw a connection to our original definition of $\neg$ introduction.

We also have an eliminator for $\neg$ which comes from *absurd*. Remember that our

$\neg$ elimination involved us assuming A and $\neg A$. In this eliminator we will not be expanding out $Not\ A$, but note that $Not\ A = A \to Void$.

$$lnotElim : A \to Not\ A \to B$$

$$lnotElim\ a\ notA\ = absurd\ (notA\ a)$$

This eliminator relies on *absurd*, and corresponds to the principle of contradiction. *absurd* will take the $Void$ produced by $notA\ a$ and give us any B that we would like. Notice our assumptions do not have B, so we can eliminate $\neg$ for any B. This is a functionally unsatisfying eliminator because we have to use *absurd*, so we have no idea what the B produced will look like, but it can be useful if we want to prove things using our λ calculus.

Now we have produced introductions and eliminators for all of our basic propositional connectives, so we will move on to our universal quantifiers. This is where our types will get more complex. When we talked about predicate calculus before, we discussed how it comes into play when we want to discuss meta-theories. In this situation, our meta-theory will be objects of arbitrary types. Each type can be thought of as the meta-theory, and values of that type are the objects within the meta-theory. This will be important to our construction of both $\forall$ and $\exists$ as types.

Our type system is not robust enough to handle this yet. The type system we have been working with is what's known as a parametric type system. It allows values of arbitrary types, rather than us having to specify types as in a static type system. The natural extension we will be using is called a *dependent type system*.

4.2 Dependent Types

Dependent types allow us to capture the idea that the types of outputs of functions may depend on the values of the inputs, rather than just the value of the outputs depending on the value of the inputs. In order to do this, we need a way for types to be constructed from values. We will analyze this through a common example, $Vect$,

and generally construct our quantifiers. A $Vect$ will be a collection of types which have a length specified by their type.

We will need to construct two types, one which corresponds to $\forall$ and another which corresponds to $\exists$. The first will be the type equivalent to $\forall$. To do this, we need a function P which takes an input and produces a new type. Our introduction rule will be

$$\frac{x : A \quad P : A \to Type}{P(x) \text{ is a Type.}}$$

We can now include in the notation for our type definition the $(x : A)$ then we can produce

$$Q : (x : A) \to P(x)$$

As we can view the type P as an arbitrary proposition, and the type A as some meta-theory we're iterating over, Q corresponds to the proposition $\forall(x : A), P(x)$.

Our deconstruction rule for $\forall$ will just be functional application, but the type of the output will be dependent.

$$depApp : ((x : A) \to P(x)) \to (t : A) \to P(t)$$

$$depApp\ f\ x = f(x)$$

In order to make a concrete illustration of this, we're going to use the example of $Vect$. A $Vect$ will be a $List$ with a specified length, so first we're going to define both $List$ and Nat for lists and natural numbers.

$$\frac{}{Nat \text{ is a type}}$$

For Nat we will need two type constructors, Z and S which will correspond to Zero and the Successor function.

$$\frac{}{Z : Nat} \qquad \frac{x : Nat}{S(x) : Nat}$$

The natural number 0 will therefore correspond to Z, and a number like 4 will be $S(S(S(S(Z))))$, 4 successor cases applied to 0.

$$\frac{A \text{ is a type}}{List\ A \text{ is a type}}$$

List here is a parametric type, where it takes some arbitrary type and creates a list of that type. This restricts us in the sense that we can only make lists of things of the same type, but it also gives us the power to actually know what type of objects are inside our list.

To construct our lists, we need to have a base empty list to which we can add items. We will also have to introduce a new type constructor :: which will be the symbol used to append objects to the front of our list.

$$\frac{A \text{ is a type.}}{[\,] : List\ A} \qquad \frac{x : A \quad xs : List\ A}{x :: xs : List\ A}$$

We can combine these two types to make a $Vect$, which will be a list that has its length in the type.

$$\frac{x : Nat \quad A \text{ is a type.}}{Vect\ x\ A \text{ is a type.}}$$

A $Vect$ will be constructed very similarly to a $List$, we will have to just increase the Nat in our type definition as we add items on.

$$\frac{A \text{ is a type.}}{emptyVec : Vect\ Z\ A} \qquad \frac{x : A \quad xs : Vect\ n\ A}{vecCons\ x\ xs : Vect\ S(n)\ A}$$

Now we can establish fundamental theorems concerning $Vect$. The simplest of which would be that there exists a $Vect$ for every length. We will have to be specific about what the contents of our $Vect$s are, so we will prove there exists a $Vect$ of natural numbers for every length. We have to do this because an arbitrary type is not guaranteed to be inhabited, and we cannot put items in a $Vect$ if our type is

uninhabited. A type is inhabited if it has values of its type. This proposition would be $\forall n : Nat.\exists VectnNat$, and the type of a proof would be $(x; Nat) \to VectxNat$. The function which corresponds to the proof will be

$$vectProof : (x : Nat) \to Vect\ x\ Nat$$

$$vectProof\ Z = emptyVect$$

$$vectProof\ (S(k)) = vectCons\ k\ (vectProof\ k)$$

which highlights something very interesting. This function corresponds to an inductive proof, where we rely on a base case with Z and $emptyVect$, then use an inductive step involving $S(k)$. Our use of pattern matching here corresponds to how in a proof we will occasionally have to split our proof into different cases.

Next we approach the corresponding type for $\exists$. Whenever we have a function which produces an output of type B, it is tempting for us to say that constitutes a B existing. However, the type of our function is $A \to B$. If we have no values of type A, we will not necessarily have values of type B. This function from $A \to B$ is really saying $\forall A.\exists B$. Therefore we need something more. For our $\exists$ introduction we had to assume $A(t)$ for some t. This is the same as saying our type is inhabited.

What we're really trying to create is something that mirrors the statement $\exists x.P(x)$ for some proposition/type P. The idea we are going to use is called a *dependent pair*.

$$\frac{A \text{ is a type.} \quad P : A \to Type}{(A **P) \text{ is a type}}$$

Our type constructor will be given by $(A **P)$ to differentiate it from a normal pair.

We will construct dependent pairs by making the type of the second value depend on the first.

$$\frac{x : A \quad P : A \to Type}{(x **P(x)) : (A **P)}$$

then we can deconstruct our dependent pairs in the same way we can deconstruct

our regular pairs. We can now properly formulate the statement $\exists n : Nat.Vect\ n\ Nat$ in our type system.

$$vectExistsProof : (x : Nat) \to (x : VectxNat) vectExistsProof x = (x ** vectProof x)$$

We can also eliminate our dependent types like how we eliminated our $\exists$.

$$depElim : (x : A) \to (x ** P(x)) \to (P \to Q) \to Q$$

$$depElim\ (a ** prf)\ implication = implication\ proof$$

With this set out, we have shown that our type system allows us to represent quantifier logic using the λ-calculus. We construct propositions and corresponding objects using the type system, then use λ-calculus to manipulate the statements and make proofs. This connection will be explicitly spelled out with the Curry-Howard Isomorphism, and then we will talk about how you can begin to prove statements using a language called *Idris*. *Idris* is unique in that it wasn't made just to prove statements, but is a powerful general purpose programming language as well.

Chapter 5

The Curry-Howard Isomorphism

The Curry-Howard Isomorphism is the key result that gives us the powerful tool of proof assistants. The isomorphism is something we've been slowly building up in the last few sections. The Curry-Howard Isomorphism states that for any predicate, we can write a corresponding type. Proofs of any predicate will correspond to λ-expressions of the corresponding type under this isomorphism. Using this our computer can verify a proof by verifying a function is of the proper type. The Curry-Howard isomorphism is expressed by stating that the following statements are equivalent

P is a type	P is a formula
P, Q are the same type	P, Q are the same formula
p is of type P	p is a proof of the formula P

Type constructors themselves are just functions. In the last chapter, we defined functions to create types. However, these are different from our λ-expresions. This is because our λ-calculus does not inherently have a way to define types. It is possible to define types as certain groups of λ-expressions. In practice, it is easier to just take type constructors as functions given to us. Separating the two allows us to make our programs more readable, which is key for proof assistants as we want to be able to read our proofs easily so we can understand the logic behind them.

There are some key factors we need in order to establish the Curry-Howard isomorphism. We will not have the Curry-Howard isomorphism with just any programming

language. The most important thing by far is our functions used in proving statements must be *total*.

Definition 5.1. A function is *total* if we are guaranteed an output for every possible input.

If we have a function that is not total, then on some input we might end up with no possible return. The simplest example of a function which is still useful, but not total, is the function *head*. *head* is a function which will give us the first item in a list. We will be using the definition of $List\ a$ we made in our type theory section.

$$head : List\ a \to a$$

$$head\ (x :: xs) = x$$

Now if we take *head* ([ ]), we haven't matched the pattern of [ ] in our function, so we will not get an output. In fact, we can't actually write an output for *head* ([ ]) as we're writing this function over arbitrary types. We can't generate an output of type a without knowing how to construct a or having an input of type a to pass back. *head* is therefore not a total function, and we can't use it to write proofs as we can't guarantee outputs. Trying to prove statements using functions that are not total is equivalent to having a gap in your proof. The proof is incomplete, just as the function will not be defined for all possible inputs.

The heart of the Curry-Howard isomorphism is one of structure. We can build propositions in the way we can build types. If we fix our meta-theory to some type, each proposition can correspond to a type which is constructed using the corresponding rules given in Chapter 2 and Chapter 4. A proof that satisfies the proposition can be thought of as a way of rewriting statements using our elimination and introduction rules to produce an output. This is the abstract idea captured by our λ-calculus, which we also manifest through corresponding elimination and introduction rules on types. If our functions are total, they will produce outputs for all inputs and therefore be considered valid proof of the propositions which correspond to our types by

virtue of the fact that the program compiles. There is no theorem in mathematics which only works for specific objects which satisfy the first part of implication, and on others it gives us some kind of error. Our functions must also have this property.

Variables also have some logical meaning under the Curry-Howard Isomorphism. If we take a variable defined as

$$Four : Nat$$

$$Four = S(S(S(S(Z))))$$

then we have essentially proven that there exists a Nat by writing down an explicit example. If we could not write an output for our function, then it will not compile, and there will be no existential proof. For example, we cannot write a function which satisfies the type

$$impossible : Void$$

as there are no values of type $Void$.

To illustrate the way we transform proofs into functions and back, we need some meta-theory to look at. Our meta-theory will be boolean logic, which is defined as a theory with two constants, $True$ and $False$, variables, equality, and functions with boolean values. We will introduce some basic types which correspond to the constants and equality in this theory. With λ-calculus we already have functions and variables, so this will allow us to make a type theory basis for boolean logic. Our types will be $Bool$ and $(=)$, which will be enough to prove basic statements about functions from $Bool \to Bool$.

$$\frac{}{Bool \text{ is a type.}} \qquad \frac{x : A \quad y : A}{x = y \text{ is a type}}$$

Now that we have our types we need introduction and elimination rules for our types. First, the introduction rules are

$$\frac{}{False : Bool \quad True : Bool} \qquad \frac{x : A}{Refl : x = x}$$

Here we set up our two Boolean operators, $True$ and $False$. We also define a way to construct equality based on the simple idea that everything is equal to itself. For any $x : A$, the type constructor $Refl$ will have type $x = x$. We will depend on context to tell us what value x takes.

Next we will describe our eliminators. We won't need an eliminator for $=$ or $Bool$ so we will leave them undefined and instead rely on pattern matching on our functions to make things simpler. The three functions we are going to prove things about will be $not, \&\&$, and $||$ which will be analogs for $\neg$, $\wedge$, and $\vee$. First, not:

$$not : Bool \to Bool$$

$$not\ False = True$$

$$not\ True = False$$

Second, $\&\&$:

$$(\&\&) : Bool \to Bool \to Bool$$

$$True\ \&\&\ True = True$$

$$False\ \&\&\ y = False$$

$$x\ \&\&\ False = False$$

and finally $||$.

$$(||) : Bool \to Bool \to Bool$$

$$True\ ||\ x = True$$

$$x\ ||\ True = True$$

$$False\ ||\ False = False$$

First, we note that these are defined exactly how $\neg, \wedge$, and $\vee$ are. This allows us to represent $\neg, \wedge$, and $\vee$ in our typed λ-calculus. We will use this to prove some basic facts about Boolean propositions.

First, we prove the proposition $\forall x, y : Bool, x \;\&\&\; y = y \;\&\&\; x$ using propositional logic.

Proof. As $x, y : Bool$ there are four possibilities.

(i) $x = True$, $y = True$

(ii) $x = True$, $y = False$

(iii) $x = False$, $y = True$

(iv) $x = False$, $y = False$

We check each case by computation.

(i) $x \;\&\&\; y = True \;\&\&\; True = y \;\&\&\; x.$

(ii) $x \;\&\&\; y = True \;\&\&\; False = False = False \;\&\&\; True = y \;\&\&\; x.$

(iii) $x \;\&\&\; y = False \;\&\&\; True = False = True \;\&\&\; False = y \;\&\&\; x.$

(iv) $x \;\&\&\; y = False \;\&\&\; False = y \;\&\&\; x.$ □

We then have an equivalent proof by code, where we write a function which produces proof that && commutes for any boolean inputs.

$$and\_commutes : (x : Bool) \to (y : Bool) \to x \;\&\&\; y = y \;\&\&\; x$$

$$and\_commutes\ False\ False = Refl$$

$$and\_commutes\ False\ True = Refl$$

$$and\_commutes\ True\ False = Refl$$

$$and\_commutes\ True\ True = Refl$$

The structure of the function mirrors the structure of the proof. Just as we split into cases in the proof, we split into cases in the function. The type system actually handles checking each case by putting each term into its normal form. We can then use $Refl$ to construct our equality. This is the power of the Curry-Howard isomorphism. Our proofs have the same structure because they are in two different systems built

the same way. The benefit of the typed λ-calculus approach is that the computer can work for us by verifying our proofs and doing some simple computations through normalizing types. This idea of normalizing will be explored soon, but first we look at || commuting. We will omit the proof that || commutes, as it is almost identical to the proof that && commutes. The function that corresponds to the proof is shown below.

$$or\_commutes\ : (x : Bool) \to (y : Bool) \to x\ ||\ y = y\ ||\ x$$

$$or\_commutes\ False\ False = Refl$$

$$or\_commutes\ False\ True = Refl$$

$$or\_commutes\ True\ False = Refl$$

$$or\_commutes\ True\ True = Refl$$

In both of the proofs above, something is happening behind the scenes when we pattern match. Once we match our patterns and change x, y to actual Boolean values such as $True$, $False$ our types are being rewritten. If we did a partial pattern match, we can see the types transform. For example, if we look at *or_commutes* and only pattern match the first variable we have two functions

$$or\_commutes\ False\ y : (y : Bool) \to False\ ||\ y = y\ ||\ False$$

$$or\_commutes\ True\ y : (y : Bool) \to True\ ||\ y = y\ ||\ True$$

Once we do the pattern match over our variable y, we end up with four functions. We will only look at the $x = False$, $y = False$ case. What we might expect would be

$$or\_commutes\ False\ False : False\ ||\ False = False\ ||\ False$$

but now we can convert $False \mid\mid False$ to it's normal form, giving us

$$or\_commutes\ False\ False : False = False$$

and we can satisfy $False = False$ using $Refl$ to construct an equality proof. This is the power of strong normalization, it allows us to talk about equality and guarantee that we can reduce statements like $False \mid\mid False$ to a normal form. This way, it doesn't matter if we had $False \mid\mid (False \mid\mid False)$ or another similar statement which evaluates to $False$, it will have the same normal form so we apply that reduction in our types.

The final function we will look at will be a proof of both versions of DeMorgan's Law,

$$\neg(x \wedge y) = (\neg x) \vee (\neg y)$$

and

$$\neg(x \vee y) = (\neg x) \wedge (\neg y)$$

These proofs will have types

$$dm\_bool\_and : (x : Bool) \to (y : Bool) \to (not\ (x\ \&\&\ y) = not\ x \mid\mid not\ y)$$

$$dm\_bool\_or : (x : Bool) \to (y : Bool) \to (not\ (x \mid\mid y) = not\ x\ \&\&\ not\ y)$$

When proving DeMorgan's laws in mathematics, we again will do a case split. We will only show the function for dm_bool_and, as the one for dm_bool_or is identical.

$$dm\_bool\_and : (e : Bool) \to (y : Bool) \to (not(x\ \&\&\ y) = not\ x \mid\mid not\ y)$$

$$dm\_bool\_and\ False\ False = Refl$$

$$dm\_bool\_and\ False\ True = Refl$$

$$dm\_bool\_and\ True\ False = Refl$$

$$dm\_bool\_and\ True\ True = Refl$$

Our proofs end up being case splits primarily because there are only two values of type *Bool*. There are more complicated theories that we can look at, in particular we will look at some theorems about the natural numbers. The theory of natural numbers has one constant, 0 and a function *Succ* which takes 0 to 1, 1 to 2, etc. We will represent these through the type Nat.

$$\frac{}{Nat \text{ is a type.}} \quad \frac{}{Z : Nat.} \quad \frac{}{S : Nat \to Nat} \quad \frac{n : Nat}{S(n) : Nat}$$

This gives us two cases for proofs over the Naturals. The base case, and the successor case. This allows us to do induction. First, we will define some necessary functions.

$$(+) : Nat \to Nat \to Nat$$

$$Z + x = x$$

$$S(k) + x = S(k + x)$$

This gives us basic addition. We can now prove commutivity, Z as right identity, and associativity. First, Z as right identity will be a function of type

$$zRightID : (n : Nat) \to n + Z = n$$

and our function body will be

$$zRightID\ Z = Refl$$

$$zRightID\ S(k) = ?hole$$

What should the type of $?hole$ be? If we take a look at the type, we will see $S(k)+Z = S(k)$ which can be reduced to $S(k + Z) = S(k)$. We need to use the property of congruence here to take a proof that $k + Z = k$ to a proof that $S(k + Z) = S(k)$. We

will define a function *cong* which will give us this property.

$$cong : (x : A) \to (y : A) \to x = y \to (f : A \to B) \to f\ x = f\ y$$

$$cong\ x\ x\ Refl\ f = Refl$$

Once we pattern match our $x = y$ to $Refl$, we change the variable name and replace all y in our type with x, meaning we have to produce a value of type $f\ x = f\ x$, which can be constructed with $Refl$. Now that we have *cong*, we can finish our proof.

$$zRightID : (n : Nat) \to n + Z = n$$

$$zRightID\ Z = Refl$$

$$zRightID\ S(k) = cong\ (k + Z)\ k\ (zRightID\ k)\ S$$

where S is our successor type constructor. This will step down to our lower k case, and lift the proof using successor to our $(S\ k)$ case. We can observe these proofs becoming more and more complex, where we had to pass *cong* four bits of information to get anywhere. This can be alleviated when we no longer have to check computations ourselves. Instead, writing a programming language to handle a large portion of the computations means we can also have the computer make certain assumptions about inputs based on context. A programming language designed to prove theorems using computer assistance is sometimes called a *proof assistant*. In the next two sections, we will be looking at Idris, a programming language developed by Edwin Brady which supports computer assisted proof. Other proof assistants are devoted to proving theorems rather than programming. Idris is first and foremost a general purpose programming language and so it is an interesting question to see how capable of proof Idris is.

Chapter 6

Idris

There are many proof assistants available for use. The one we will cover will be Idris, developed by Edwin Brady [2]. Idris is built on top of the functional programming language Haskell[1]. Those interested in Idris as a general purpose programming language are encouraged to read Edwin Brady's book, Type Driven Development [3] which focuses on Idris' use as a traditional programming language. Here we will take a look at the other side of Idris: As a dependently typed language, Idris can also be used to prove statements alongside code. For a language that can produce reasonable code, being able to blend proof into our code is very powerful.

To install Idris, go to https://www.idris-lang.org/pages/download.html and download the binary for Idris 1. From there, it's very simple to use Idris. In the text editor of your choice, create a file with the filename "test.idr". Save that file in any location, then open your command line. Navigate to the same location as the file and type "idris". This will launch the Idris read, evaluate, print loop or REPL in your command line. From here, type ":l test.idr" to load your file. If we had a function in the file "test", we could then type the name of the function in the command line to run the function. The basic commands needed for Idris can be found in Table 6.1. With the basics of running Idris explained, we can dive into the language itself.

To take a look at Idris, we will need to talk a bit about the syntax of Idris. Most of

[1]Idris follows the syntax of Haskell closely. A familiarity with Haskell may be beneficial to learning Idris. The free resource "Learn You a Haskell for a Great Good" [9] is a great place to begin learning Haskell.

:l <filename>	Loads the file into the REPL.
:t <function name>	Gives the type of a function.
<function name> <argument(s)>	Evaluates a function on the arguments and prints the result.

Table 6.1: Basic Idris REPL Commands

the syntax matches what we've covered already. For example, to say that a variable is of a certain type in Idris we use a single colon. However, we will introduce types in a completely different way. Idris allows us to define types using something called *data types*. We will define our data type for Bool here. For the rest of this thesis, all the text in between two lines will be actual code for Idris which can be compiled and checked.

```
data Bool : Type where
    True : Bool
    False : Bool
```

The line `data Bool : Type where` is our type declaration and is akin to us saying

$$\frac{}{Bool \text{ is a type.}}$$

The lines after `data Bool : Type where` are where we define our type constructors. This gives us two ways to construct Bool, `True` and `False`. These lines are the same as in the previous chapter when we said

$$\frac{}{True \text{ is of type Bool.}} \quad \text{and} \quad \frac{}{False \text{ is of type Bool.}}$$

and in order to prove everything from the last section in Idris code, we will define equality the standard way.

```
data (=) : a -> b -> Type where
    Refl : x = x
```

We can now write our functions from the last chapter. In fact, in the last chapter all

of the proofs as programs were actually Idris code. Meaning we can just write out our proofs as

```
and_commutes : (x : Bool) -> (y : Bool) -> x && y = y && x
and_commutes False False = Refl
and_commutes False True  = Refl
and_commutes True  False = Refl
and_commutes True  True  = Refl

or_commutes : (x : Bool) -> (y : Bool) -> x || y = y || x
or_commutes False False = Refl
or_commutes False True  = Refl
or_commutes True False  = Refl
or_commutes True True   = Refl

dm_bool_and : (x : Bool) -> (y : Bool) ->
              (not (x && y) = not x || not y)
dm_bool_and False False = Refl
dm_bool_and False True  = Refl
dm_bool_and True False  = Refl
dm_bool_and True True   = Refl

dm_bool_or : (x : Bool) -> (y : Bool) ->
             (not (x || y) = not x && not y)
dm_bool_or False False = Refl
dm_bool_or False True  = Refl
dm_bool_or True False  = Refl
dm_bool_or True True   = Refl
```

and verify the proofs by compiling.

We can also define *Nat* from last section as a data type through

```
data Nat : Type where
    Z : Nat
```

```
S : Nat -> Nat
```

In our language, we previously had only one kind of argument. All arguments to a function would have to be made explicitly. In Idris, we have a system which allows us to make arguments *implicit*. An implicit argument is one that Idris can figure out based on the context. When we want to make an argument implicit, we will use curly braces ({}) instead of parentheses (()) or nothing ( ) when defining the type of our argument. The best example is in Idris' definition of `cong`. Then we can define `cong` using the standard definition

```
cong : {f : a -> b} -> (x = y) -> f x = f y
cong Refl = Refl
```

We make `f` implicit in our definition of `cong` as it is not used anywhere in the function itself. This allows us to deal with less cases in our function definition, as we don't need to look at the details of `f`. If an argument is implicit and we need it, we can bring the argument *into scope*, meaning make it available for use in the function, using curly braces by the function's definition. For `cong` we can bring `f` into scope by writing

```
cong : {f : a -> b} -> (x = y) -> f x = f y
cong {f} Refl = Refl
```

We could then use `f` in our function. As we do not need `f`, we will leave it out of scope. When we call the function `cong`, we will not have to specify what function `f` represents. A call of `cong` where `f` is implicit will look like this

```
zRightID : (n : Nat) -> n + Z = n
zRightID Z = Refl
zRightID (S k) = cong (zRightID k)
```

However, in some situations this may give us a type error if our program can't automatically interpret what `f` should be. If you load this code into Idris, it will give us exactly that type error. Thankfully, we can make implicit calls explicit using a

simple syntax. In our function call, we will add a set of curly braces and the variable name {`f = Func`} where `Func` is the function we'd like to call for `f`.

```
zRightID : (n : Nat) -> n + Z = n
zRightID Z = Refl
zRightID (S k) = cong {f=S} (zRightID k)
```

This code will compile, confirming for us that $\forall n \in \mathbb{N} : n + 0 = n$

There is another tool that Idris has which is useful for theorem proving. Idris allows us to create type *interfaces*. An interface is a way for us to deal with types that have functions of a certain type defined over them. This is useful because we can use the types to give us properties of a type without worrying about the specific implementation. Idris has many interfaces implemented by default. One interface is called `Eq` and deals with boolean equality. The implementation for this interface will be of this form.

```
interface Eq ty where
    (==) : ty -> ty -> Bool
    (/=) : ty -> ty -> Bool
```

The line `interface Eq ty where` is called the *interface declaration* and tells Idris that we are defining a new interface. The next lines are called the *method declarations* and will give us functions of a certain type over our type `ty`. An interface acts as a shell for our type `ty`. We can then constrain the inputs of a function to have a certain interface defined for a type. This will allow us to use all of the interface's method in the function body and type definition. To do this, we will look at a function which decides if lists are equal based on all their elements being equal. First, the definition of a list in Idris looks like

```
data List : Type -> Type where
    [] : List a
    :: : a -> List a
```

where the empty brackets are called the *empty list* and the two colons (`::`) is the *cons operator* which will allow us to add new elements to the front of our list. We

can now constrain our type in our function declaration.

```
listEq : Eq a => List a -> List a -> Bool
listEq [] [] = True
listEq [] (x :: xs) = False
listEq (x :: xs) [] = False
listEq (x :: xs) (y :: ys) = x == y && listEq xs ys
```

Our use of `Eq a` says that everything of type a that comes afterward will have an implementation of Equality. We then use this power in our fourth case of listEq to test the first elements of each list against each other. Interfaces also have a lot of power for mathematicians which will be shown in the next section. Since types correspond to propositions, interfaces can be used to write up axiom systems.

Idris has a useful tool for checking types in your program called a *hole system*. When defining a function, normally leaving a function incomplete will cause a type error. We can make our compiler ignore certain parts of the function by calling a `?hole`.

```
incomplete : t -> t
incomplete x = ?hole
```

We can then ask our compiler what type it is expecting to show up in `?hole` through Idris by calling the command `:t hole`. The output in this situation will tell us it is expecting `hole : t` and tell us we have one argument `x : t`. Using this information, we can tell that we must complete the function by returning `x`, as we have no way to construct an abstract type `t`.

```
incomplete : t -> t
incomplete x = x
```

Holes can be used to help guide proof as well, as types will correspond to propositions and guide us towards what we must construct as proof. In the above example, we can interpret the type as $\forall t : \exists t$ and this proof is satisfied by the identity function.

There is one final tool we must discuss about in Idris, and that is the function `rewrite x in y`. Rewriting in Idris is a useful way to deal with proofs about equality.

Rewriting is based on a function called replace. Replace allows us to take a type dependent on `x` and using proof that `x = y`, we can produce the same type but dependent on `y`. This allows us to just rewrite our types to control the outputs.

```
replace : {P : a -> Type} -> {x : a} -> (x = y) -> P x -> P y
replace Refl x = x
```

Mathematically this just tells us if $P(x)$ has a proof and $x = y$ then $P(y)$ has a proof. We can use rewrite to prove `zRightId` without using cong.

```
zRightID : (n : Nat) -> n + Z = n
zRightID Z = Refl
zRightID (S k) = rewrite zRightID k in Refl
```

Which will take `Refl : S k = S k` and using `zRightID k : k + Z = k`, rewrite the type from `S k = S k` to `S (k + Z) = S k`. This will satisfy the type we need for the `(S k)` case. Often times there will be a `Refl` at the end of our chain of rewrites, standing in for some abstract proposition. The key to reading these proofs is figuring out what `Refl` stands in for.

In the next section, we will be looking at proofs in Idris which will use the systems described in this chapter. These proofs will serve as evidence of Idris' capabilities in formal proof verification without looking at code.

Chapter 7

Implementation

To further illustrate applying the Curry-Howard Isomorphism, we will look at some annotated code from Idris. The two examples we will look at will be the basic definitions of prime numbers and division. The full code can be found online in a github repository[1]. Through this section, we will create a commentary on the code found in the repository in order to illustrate the Curry-Howard isomorphism.

7.1 Division

```
module NumberTheory

import Data.Fin

%default total
%access public export
```

Here we just set up the beginnings of number theory, $Data.Fin$ is a type which is guaranteed to have a finite number of items. This is useful for restricting the numbers we are working with to numbers less than some $n \in \mathbb{N}$.

[1] https://github.com/bencald/newcollegethesis

```
data Divides : Nat -> Nat -> Type where
  divPrf : {a : Nat} -> {b : Nat} ->
           (q : Nat ** (a * q) = b) -> a `Divides` b

data Prime : Nat -> Type where
  primePrf : (x : Fin n) -> (finToNat x) `Divides` n ->
              x = (FS (FZ)) -> Prime n
```

We set up our two basic data types that we want to prove things over. To write these we need to look at the propositions we are trying to represent through types. Here, our meta-theory is elementary number theory on the Natural numbers, so we use the data type Nat which includes a 0 and a Successor function. Then we look at division and primes over these numbers.

We say a "divides" b for $a, b \in \mathbb{N} \iff \exists\, q$ s.t. $a * q = b$.

```
dividesRefl  : (n : Nat) -> n `Divides` n
dividesRefl n = divPrf (1 ** multOneRightNeutral n)
```

To prove that every number divides itself, we show that right multiplication on n by 1 gives us n, and put that proof in a dependent pair with 1.

```
oneDivides : (n : Nat) -> 1 `Divides` n
oneDivides n = divPrf (n ** plusZeroRightNeutral n)
```

Similarly, we prove that 1 divides every number n by showing that right multiplication of 1 by n gives us n, and put that proof in a dependent pair with n.

```
dividesZero : (n : Nat) -> n `Divides` 0
dividesZero n = divPrf (0 ** multZeroRightZero n)
```

Here we show that every number divides 0 using a similar method.

```
zeroDividesOnlyZero : 0 `Divides` n -> 0 = n
zeroDividesOnlyZero (divPrf (x ** pf)) = pf
```

To show $0|n \implies 0 = n$, we must unwrap our dependent pair and get at the proof underneath it which will be a value of type $0 * x = n$, which simplifies automatically to a value of type $0 = n$, exactly what we need.

```
dividesTrans : d `Divides` n -> n `Divides` m -> d `Divides` m
dividesTrans (divPrf (x ** pfx)) (divPrf (y ** pfy)) =
  divPrf (x * y **
    rewrite multAssociative d x y in
    rewrite pfx in rewrite pfy in Refl)
```

Here we prove the transitivity of division, $d|n \wedge n|m \implies d|m$. We do this by leveraging the rewrite power of Idris to rewrite our equality proofs.

```
dividesLinearity : d `Divides` n -> d `Divides` m ->
                   d `Divides` ((a*n) + (b*m))
dividesLinearity {a} {b} {n} {m}
(divPrf (x ** pfx)) (divPrf (y ** pfy)) =
  divPrf (plus (mult a x) (mult b y) **
    rewrite multDistributesOverPlusRight d
            (mult a x) (mult b y) in
    rewrite multCommutative a x   in
    rewrite multCommutative b y   in
    rewrite multAssociative d x a in
    rewrite multAssociative d y b in
    rewrite pfx in rewrite pfy    in
    rewrite multCommutative n a   in
    rewrite multCommutative m b   in Refl)
```

Here we prove the linearity of division, so $d|n \wedge d|m \implies d|(a*n)+(b*m)$. We accomplish this by first constructing the proper q for our *Divides* type, then we construct the equality through multiple rewrites.

```
dividesMultiplication : (a : Nat) -> d `Divides` n ->
                        (a*d) `Divides` (a*n)
dividesMultiplication {d} a (divPrf (x ** pf)) =
  divPrf (x **
          rewrite sym $ multAssociative a d x in
          rewrite pf in Refl)
```

The final thing we prove is the multiplicativity of division, or $\forall a : Nat$, $d|n \implies (a*d)|(a*n)$. We accomplish this through a rewrite which shows that if $d*q=n$, then $(a*d)*q=(a*n)$.

To deal with more complex theories, we need to talk about how to describe an axiom system in Idris. A great example of how to do this comes up when we want to define groups. Here, we leverage a powerful tool of Idris, the interface system. This will allow us to make a type with functions defined on it already. These functions do not need actual implementations, as their types will give us the basic axioms of a group as propositions. We can then use these unwritten functions to prove things. If we want to prove a certain type g is a group or use any theorems we write, we will then have to implement an interface over the type g.

For quick review, and so we can easily compare between the types and the axioms they are based on, the group axioms are placed below.

Definition 7.1. A *group* is a set G with a binary operation $\circ : G \times G \to G$ satisfying the following axioms:

1. For every $a, b, c \in G : a \circ (b \circ c) = (a \circ b) \circ c$.
2. There exists an $e \in G$ satisfying $\forall\, a \in G : e \circ a = a$.
3. For every $a \in G$ there exists a $b \in G$ satisfying $b \circ a = e$.

The code to implement an interface for a group will look like:

```
interface Group g where
  (<>)     : g -> g -> g
  assoc    : (a : g) -> (b : g) -> (c : g) ->
             a <> (b <> c) = (a <> b) <> c
  identity : g
  leftID   : (a : g) -> identity <> a = a
  leftInv  : (a : g) -> (b : g ** b <> a = identity)
```

where `(<>)` or "diamond" corresponds to the group multiplication, and the other functions correspond to the group axioms. It is satisfying to note that implementing all of the functions of these types will correspond to the process a mathematician goes through to prove a set and binary operation form a group.

We can now use functions to write short definitions for properties of group elements. For ease of writing and reading types, we can define *Identity* and *Idempotent* as such:

```
Idempotent : Group g => g -> Type
Idempotent a = (a <> a = a)

Identity : Group g => g -> Type
Identity a = (a = identity)
```

It is important to note that the equality contained inside the parentheses is different from the one outside of the parentheses. The equality inside of the parentheses is a type constructor, which constructs a type which describes equality. The equality outside of the parenthesis is a part of Idris's syntax, and defines a function.

In general, using interfaces in Idris will allow us to write functions, corresponding to proofs or definitions like *Idempotent* or *Identity* over arbitrary groups. We do not care necessarily about the implementation of the group interface or what type the interface is constructed over. All of the information we could need is encoded in the types of the functions in the *Group g* interface. This allows for very quick (and

somewhat easier to read) theorem definitions. Idris accomplishes this through the use of constraints, which can be seen above with the *Group g => ...*, which binds g in the following type to be a type with a group implementation. Once we have this basic structure, we wish to test if it is usable. In order to do this, we prove some basic properties of groups up to the cancellation laws.

7.2 Theorems using the Group Interface

To test our interface, we took a look at the first section of [6]. Our systems should ideally be able to prove any statement about groups from the early sections of this book at this point. The first proposition we prove is

Proposition 7.1. *For a group G:*

1. *The identity of G is unique.*
2. *For each $a \in G$, the inverse of a is unique.*
3. *For $a \in G$: $a^{-1^{-1}} = a$*
4. *For $a, b \in G$: $(a \circ b)^{-1} = b^{-1} \circ a^{-1}$*

The first section tackled is the uniqueness of the identity operator. The functions we wrote for this proof made heavy use of Idris's rewrite capabilities. Rewrite in Idris allows us to take some value of type $P\ x$, and convert it into a value of type $P\ y$ given that we have a value of type $x = y$. This is a very powerful tool to assist with proofs, and is something we must be conscious of when building new structures. When building structures, relating back to Idris's equality will allow us to utilize the power of rewrite fully in our code. Therefore, we should not try to redefine equality.

Rewrite works very well when combined with the hole system, as we can rewrite the type of our hole into an already solved problem. To illustrate this, we will step back from the proposition and look at an example of proving the transitive property of equality using rewrite. There are other ways to prove the transitive property of

equality, some simpler than rewrite, but this example allows us to avoid the detail of doing multiple rewrites.

```
transitivity : x = y -> y = z -> x = z
transitivity xeqy yeqz = ?hole
```

In the above, the type of ?*hole* will be $x = z$. If we could just replace the x in that equality with a y, as $xeqy$ tells us should be possible, we could solve the problem using $yeqz$. This is precisely what rewrite will allow us to do. In one rewrite, we can get to

```
transitivity : x = y -> y = z -> x = z
transitivity xeqy yeqz = rewrite xeqy in ?hole
```

where the type of ?*hole* is now reduced to $y = z$ by merits of replacing every x in $x = z$ with a y. We can now replace ?*hole* with $yeqz$ to complete the proof. When doing multiple rewrites, it can be easy to lose the thread of the proof when reading later. However, while actually proving the statements, rewrites can be used to slowly guide us in the right direction. To prove Proposition 7.1, we will have to use multiple rewrites. If you have Idris downloaded, you can go through each rewrite step by step using the hole system. Alternatively, each proof can be taken at face value using the types to understand what is being proven.

Tackling the first part of the proposition we will outline the mathematical proof which we are trying to represent in Idris. We will outline each step as we will have to tell the computer every single step. Idris cannot make inferences about why something might be true, it must have justification for every step. If we compare this to sharing proof with another mathematician, they can make inferences about why implications are true even if the explicit reason is not given.

Lemma 7.2. $a <> a = a \implies a = identity$

Proof.

$$a <> a = a$$

$$\text{as } a = identity <> a \implies a <> a = identity <> a$$

$$\text{as } identity = a^{-1} <> a \implies a <> a = (a^{-1} <> a) <> a$$

$$\text{as } (a^{-1} <> a) <> a = a^{-1} <> (a <> a) \implies a <> a = a^{-1} <> (a <> a)$$

$$\text{as } a <> a = a \implies a = a^{-1} <> a$$

$$\text{as } a^{-1} <> a = identity \implies a = identity$$

□

When translating the above proof to a rewrite function, we actually flip our reasoning around. This is because our function is expecting a type of value $a = identity$, and we progressively rewrite this type to be $a <> a = a$. When reading this function, look back at the proof and work from $a = identity$ backwards through to see what rewrite is accomplishing.

```
idempotentIsIdentity : Group g => {a : g} -> Idempotent a ->
                       Identity a
idempotentIsIdentity {a} axaisa = let
  (aInv ** invPrf) = leftInv a
  in

    rewrite sym invPrf in
    rewrite sym axaisa in
    rewrite assoc aInv a a in
    rewrite invPrf in
    rewrite leftID a in axaisa

identityIsUnique : Group g => {a : g} ->
                   ((b : g) -> a <> b = b) ->
                   Identity a
identityIsUnique {a} prf = idempotentIsIdentity $ prf a
```

To prove part 2, we again turn to the use of rewrite. When dealing with equational reasoning, rewrite will come up frequently. This is because rewrite is easy to use

alongside the REPL, slowly rewriting the type of a hole into whatever we need.

```
backwardsIsIdempotent : Group g => {a : g} ->
                        Identity (a<>b) -> Idempotent (b<>a)
backwardsIsIdempotent {a} {b} idPrf =
    rewrite sym $ assoc b a (b<>a) in rewrite assoc a b a in
    rewrite idPrf in rewrite leftID a in Refl

leftInvisRightInv : Group g => {a : g} -> {b : g} ->
                    Identity(a<>b) -> Identity(b<>a)
leftInvisRightInv =
    idempotentIsIdentity . backwardsIsIdempotent
```

Part 3 can be proved with one function as follows:

```
aInvInvisa : Group g => {a : g} -> Identity (b<>a) ->
                        Identity (c<>b) -> a = c
aInvInvisa {a} {b} {c} idba idcb =
  rewrite sym $ leftID a in rewrite sym idcb in
  rewrite sym $ assoc c b a in rewrite idba in
  rewrite rightID c in Refl
```

Similarly, we can finish the proposition with part 4 in one single function:

```
abInvisbInvaInv : Group g => {a : g} -> Identity(d<>b) ->
                  Identity (c<>a) -> Identity((d<>c)<>(a<>b))
abInvisbInvaInv {a} {b} {c} {d} dbid caid =
  rewrite sym $ assoc d c (a<>b) in rewrite assoc c a b in
  rewrite caid in rewrite leftID b in dbid
```

The use of rewrite made these proofs short and easy to write, but much harder to read. Without seeing an example of how rewrite actually operates on the types, or using rewrite yourself, these proofs can be very difficult to read. There are paths around this, but rewrites are incredibly easy to use in the moment thanks to Idris's

hole system. The idea is that you can progressively rewrite the type of a hole in order to reduce the hole to an input for which you have a value which matches its type. Then, the rewrite will operate on that value to get the final solution.

Our next goal is to build the cancellation laws. This will allow us to easily manipulate equality over groups. The cancellation laws are

$$a <> u = b <> u \implies a = b \text{ and } u <> a = u <> b \implies a = b$$

We build to the cancellation laws first by giving us a simple way to handle multiplying by inverses on both the left and right sides.

```
leftMultInv : Group g => {a : g} -> a <> x = b ->
                         Identity(c<>a) -> x = c <> b
leftMultInv {a} {b} {c} {x} eqPrf idPrf =
  rewrite sym eqPrf in rewrite assoc c a x in
  rewrite idPrf in rewrite leftID x  in Refl

rightMultInv : Group g => {a : g} -> x <> a = b ->
                          Identity(a<>c) -> x = b <> c
rightMultInv {a} {b} {c} {x} eqPrf idPrf =
  rewrite sym eqPrf in rewrite sym $ assoc x a c in
  rewrite idPrf in rewrite rightID x in Refl
```

Once we have our inverses, we can make the statement that $a <> x = b$ is satisfied by a unique x. Again, we split over the left and right multiplication cases.

```
rUniqueSolution : Group g => {a : g} -> a <> x = b ->
                             a <> y = b -> x = y
rUniqueSolution {a} {b} {x} {y} axisb ayisb = let
  (aInv ** aInvPrf) = leftInv a
  xIsAInvB =
    leftMultInv {a} {b} {c=aInv} {x} axisb aInvPrf
  yIsAInvB =
    leftMultInv {a} {b} {c=aInv} {x=y} ayisb aInvPrf
```

```
  in
    xIsAInvB `trans` (sym yIsAInvB)

lUniqueSolution : Group g => {a : g} -> x <> a = b ->
                             y <> a = b -> x = y
lUniqueSolution {a} {b} {x} {y} xaisb yaisb = let
  (aInv ** aInvPrf) = rightInv a
  xIsBAInv =
    rightMultInv {a=a} {b=b} {c=aInv} {x=x} xaisb aInvPrf
  yIsBAInv =
    rightMultInv {a=a} {b=b} {c=aInv} {x=y} yaisb aInvPrf
  in
    xIsBAInv `trans` (sym yIsBAInv)
```

Once we have the unique solution property, we can easily prove our cancellation laws. By building up these tools, we hope to lay the groundwork for powerful equational reasoning tools. Manipulating equality will always be a part of proving statements in Idris

```
lCancel : Group g => {a : g} -> a <> u = b <> u -> a = b
lCancel prf = lUniqueSolution prf (Refl)

rCancel : Group g => {a : g} -> v <> a = v <> b -> a = b
rCancel prf = rUniqueSolution prf (Refl)
```

The cancellation laws are a useful tool which should hopefully allow us to write more concise proofs rather than doing every step, every time when using equational reasoning.

7.3 Further Group Structures in Idris

There are other structures from group theory which will be useful to define. We will spend this section going over their definitions, some of which will be shorter than

others. In brief, we will define

- Abelian groups
- Group actions
- Subgroups
- Quotient groups

```
interface Group g => Abelian g where
  op_commutes : (a : g) -> (b : g) -> a <> b = b <> a
```

Group actions are the quickest to define, as they will just be a function between our group and some other, arbitrary type. We will have to have two axioms, one for associativity and another which guarantees that the identity acts as it should. As such, an interface is a good choice to define a group action as an extention of some existing group g.

```
interface Group g => GroupAction g a where
  (<.>)          : g -> a -> a
  assocAction    : (g' : g) -> (g'' : g) -> (a' : a) ->
                    g' <.> (g'' <.> a') = (g' <> g'') <.> a'
  identityAction : (a' : a) ->
                   Interfaces.identity <.> a' = a'
```

Once we have groups, we need to find a way to talk about subgroups. First, we must have a way to discuss subtypes. The natural idea is refinement typing [7], which allows us to work only with values which evaluate to true under some predicate p. If this predicate happens to have operative closure, the identity element of the group, and inverses for every element of the group, then we happen to have a refinement that will act as a subgroup. We can then talk about the group over that refinement type. The following three code blocks follow this pattern precisely.

```
data Refinement : (g : Type) ->
                  (p : g -> Bool) -> Type where
  In : (e : g) -> {auto prf : p e = True} -> Refinement g p
```

As you have to provide proof that $p\ e\ =\ True$, a refinement will only give us values for which $p\ e$ actually evaluates to $True$. This can be seen as comparable to checking inclusion in a subgroup, and gives us a concise notion of subtype.

```
interface Group g => Subgroup g (p : g -> Bool) where
  op_closure  : Group g => (a : g) -> (b : g) ->
                           {auto prfa : p a = True} ->
                           {auto prfb : p b = True} ->
                           f (a <> b) = True
  id_closure  : Group g => p (Interfaces.identity) = True
  inv_closure : Group g => (a : g) ->
                           {auto prf : p a = True} ->
                           (p (fst $ leftInv a) = True)
```

A subgroup is just a special type of refinement function, which has operative, identity, and inverse closure. This parallels the properties a mathematician would have to prove in order to show that a subset is actually a subgroup. Once we have all these facts proved, we can actually talk about the group over the refinement type with our subgroup function.

```
Subgroup g f => Group (Refinement g p) where
  (<>) (In x) (In y) =
    (In (x<>y) {prf=op_closure x y})
  assoc (In x) (In y) (In z) = inRespectsEq $ assoc x y z
  identity       = (In identity {prf=id_closure {p}})
  leftID (In x)  = inRespectsEq $ leftID x
  leftInv (In x) =
    ((In (fst $ leftInv x) {prf=inv_closure {p} x}) **
        inRespectsEq $ snd $ leftInv x)
```

Therefore, once we prove that a function can create a subgroup we already have a subgroup defined over this function thanks to our *Refinement* data type and the *Subgroup* interface of the predicate p. Some functions used to prove these properties have been left out, and can be found in the GitHub linked in the introduction.

The final structure we tackled was the quotient group [4]. One idea for proceeding is to define quotient types, which we did not pursue but could show some promise. However, quotient types were beyond the scope of this project. Information on quotient types can be found in Nuo Li's 2015 thesis [8]. Instead, we opt to have a choice function select a distinguished element from the equivalence classes which make up our quotient. We then deal only with the distinguished elements and see how they act as a group. In order to do this we must first define equivalence relations and choice functions. To define equivalence relations, we look to the mathematical axioms for an equivalence relation R, namely

1. **Reflexivity:** For all a we conclude that $a\ R\ a$.
2. **Symmetry:** For all a, b such that $a\ R\ b$ we conclude that $b\ R\ a$.
3. **Transitivity:** For all a, b, c such that $a\ R\ b$ and $b\ R\ c$ we conclude that $a\ R\ c$.

```
interface EquivalenceRelation (r : t -> t -> Type) where
  reflexive : {a : t} -> a `r` a
  symmetric : {a : t} -> {b : t} -> a `r` b -> b `r` a
  transitive : {a : t} -> {b : t} -> {c : t} ->
               a `r` b -> b `r` c -> a `r` c
```

Note that for our relation r, we care that it outputs a *Type* and not a specific value. Viewing this as a logician, it says we can form some statement which is true about objects when they relate. The programmer will say that when objects relate we can use them to describe an output.

Once we have a relation, we need to talk about how to pick things out of equivalence classes. To accomplish this, we have to put a few restrictions on our functions to ensure they act as proper choice functions. These restrictions will guarantee that our

choice function is well behaved with respect to our operation <>, meaning it does not matter which specific element our choice function selects. This will allow us to shrink down the size of the group to focus on these distinguished elements. If we tried to work with whole equivalence classes, say as lists, we would run into trouble quickly. Lists can have multiple representations, and couldn't handle an infinite equivalence class. These are both things that come up very quickly. Working with distinguished elements has other benefits, too. The canonizer approach matches our intuitive notions for quotient groups such as $\mathbb{Z}_4$, where we care only about the canonical elements $0, 1, 2$, and 3. When working with these objects, we do not think of them as sets of related objects unless we are required to. The proper implementation of a choice function with it's underlying relation is given as follows, where the axioms guarantee that the function maps within equivalence classes, maps related things to the same object, and these choices respect our group operation <>.

```
interface (Group g, EquivalenceRelation r) =>
          Choice g (f : g -> g) (r : g -> g -> Type) where
  choice      : Group g => {a : g} -> a `r` f a
  welldefined : Group g => {a : g} -> {b : g} ->
                           a `r` b -> f (a) = f (b)
  coherence   : Group g => {a : g} -> {a' : g} ->
                           {b : g} -> {b' : g} ->
                           a `r` a' -> b `r` b' ->
                           f (a <> b) = f (a' <> b')
```

Given both definitions for choice functions and equivalence relations, we can start to talk about what elements are distinguished. A distinguished element will just be elements fixed by the choice function. These fixed elements will be the value under choice for all their related functions, and will therefore be the only element mapped to in their equivalence class. Our definition for the *Canon* data type will look like our definition for *Refinement*, with one key difference.

```
data Canon : (g : Type) -> (f : g -> g) ->
             (r : g -> g -> Type) -> Type where
```

```
  InCanon : (a : g) -> {auto prf: a = f a} -> Canon g f r
```

The key difference being that instead of checking if our function f on a evaluates to true, we check that a is a fixed point of f. If $f\ a\ =\ a$, then we will have that a is the canoncial element for its equivalence class. Our *Canon* data type restricts us to only working with distinguished elements under the *Choice* $f\ r$.

The final step is to show that we can act as a group on these distinguished elements. We accomplish this in the same manner as the *Subgroup*. However, the *Canon* case is much more complex, and has many of the functions used to prove this structure left out for brevity's sake. To fully understand what is happening here, the code found on the GitHub listed in the introduction is a valuable resource.

```
(Group g, Choice g f r) => Group (Canon g f r) where
  (<>) (InCanon x) (InCanon a)
    = InCanon (f (x <> a)) {prf= (canonizeOnce (x<>a) {r})}
  assoc (InCanon x {prf=prfx})
                (InCanon y {prf=prfy})
                        (InCanon z {prf=prfz})
    = let
        canLeft   = canonizeLastLeft {g} {f} {r} {x=x<>y} {y=z}
        canRight  = sym $
                    canonizeLastRight {g} {f} {r} {x} {y=y<>z}
        assocCong = cong {f} $ assoc x y z
        equality  = canRight `trans` assocCong `trans` canLeft
        in canonRespectsEq equality
  identity
    = InCanon (f(identity)) {prf = canonizeOnce (identity) {r}}
  leftID (InCanon y {prf})
    = let
        canLeft   = sym $
                    canonizeLastLeft {g} {f} {r}
                        {x=identity} {y}
        congIDlaw = cong {f} $ leftID y
        equality  = canLeft `trans` congIDlaw `trans` (sym prf)
        in canonRespectsEq equality
```

```
leftInv (InCanon x)
  = let
    (xInv ** xInvPrf) = leftInv x
    canLeft  = sym $ canonizeLastLeft {g} {f} {r} {x=xInv} {y=x}
    congInv  = cong {f} xInvPrf
    equality = canLeft `trans` congInv
    in ( (InCanon (f xInv) {prf= canonizeOnce {g} {f} {r} xInv})
          ** canonRespectsEq equality)
```

The code above exists as a potential future way to prove facts about quotient groups using the natural idea of picking a distinguished element.

Chapter 8

Conclusions

Hopefully by now you have a better understanding of the basics of proof assistants, type theory, and functional programming. There are many other resources that you can use to continue to learn about these topics. If you are a programmer, you may be interested in learning more about Idris. What we have done here has barely scratched the surface of Idris' capabilities. A great resource to learn more is *Type Driven Development* by Edwin Brady [3]. For a treatment of λ-calculus and type theory, Simon Thompson's *Type Theory and Functional Programming* [10] is a good choice due to its focus on programming over mathematical formalism and proof. We relied heavily on the above text, but the texts themselves contain a wealth of additional information.

If you are interested more in the theory behind the Curry-Howard Isomorphism from a mathematical standpoint, Barendregt's *Lambda Calculus* [1] is a comprehensive treatment of everything from the theory section of this thesis and more. A mathematician might find that Idris is a difficult language to use as it does not have libraries available for proving theorems. With worries in that direction, the proof assistants Coq or Agda might be far preferable. They are older and more dedicated to theorem proving, and as such they have vast libraries of tools for mathematicians that Idris just does not have.

There are some future directions to take the code found in this thesis. We have barely scratched the surface of group theory, and have been trying to build the parts

necessary to prove some more significant theorems. One theorem which could be approached would be Lagrange's Theorem, that for any finite group G, the order of any subgroup H divides the order of G. To do this, we would have to establish a way to talk about orders and finite groups. This could be accomplished by looking at the data type `Fin`, which is a finite type. The order of the group would therefore be the size of `Fin`. Potentially this could be defined through an isomorphism. The next steps certainly involve finding some satisfying way to represent finite groups, as the current system does not represent finite groups in any way. Another theorem that could be looked at after solving the problem of representing finite groups is Cauchy's Theorem, that if G is a finite group and p a prime number which divides G, then G contains an element of order p. To do this, we would need a way to talk about specific orders of elements of G. This would potentially have to be a new datatype which encodes the definition for order.

Theorems from other fields of math can be proved using Idris. Proof assistants are only as powerful as the libraries they have built up over time. For Idris to truly blend programming and proof, a large library of proof tools will be invaluable. Hopefully this thesis has been enough to grasp the basics, and you are prepared to learn more about Idris or proof assistants.

Bibliography

[1] Hendrik P Barendregt et al. *The lambda calculus*, volume 3. North-Holland Amsterdam, 1984.

[2] Edwin Brady. Idris, a general-purpose dependently typed programming language: design and implementation. *J. Funct. Programming*, 23(5):552–593, 2013.

[3] Edwin Brady. *Type-driven development with Idris*. Manning Publications Co, Shelter Island, NY, 2017. OCLC: ocn950958936.

[4] William Burnside. *Theory of groups of finite order*. University, 1911.

[5] Alonzo Church. *The Calculi of Lambda-conversion*. Princeton University Press, 1941.

[6] David Steven Dummit and Richard M. Foote. *Abstract algebra*. Wiley, Hoboken, NJ, 3rd ed edition, 2004.

[7] Tim Freeman. Refinement types ml. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1994.

[8] Nuo Li. Quotient types in type theory, July 2015.

[9] Miran Lipovaca. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, USA, 1st edition, 2011.

[10] Simon Thompson. *Type theory and functional programming*. Addison Wesley, 1991.