# Generalizing the Curry-Howard Isomorphism to Classical Logic

A Thesis
Presented to
The Division of Mathematics and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Luis Edmund Maldonado

May 2013

# Acknowledgments

This thesis wouldn't have been possible without the various forms of help and encouragement that I've received from the people around me. I'd like to thank Jim, of course, for being such an enthusiastic and hard-working advisor. I wish to thank my fellow students for their level of enthusiasm and interest in my project. I wish to thank my family for their support of and interest in my education in general, and this thesis in particular. Last, but certainly not least, I wish to thank Angelica for being with me throughout this whole process.

# Table of Contents

**Introduction** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **1**

**Chapter 1: Logic** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3**

1.1 Language . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.2 Natural Deduction . . . . . . . . . . . . . . . . . . . . . . . 4

1.3 Some Examples and Results . . . . . . . . . . . . . . . . . . 7

1.4 Semantics . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

**Chapter 2: The Lambda Calculus** . . . . . . . . . . . . . . . . . . . . **11**

2.1 Syntax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.2 Reduction . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

2.2.1 Free and Bound Variables . . . . . . . . . . . . . . . . 13

2.2.2 Substitution . . . . . . . . . . . . . . . . . . . . . . . 13

2.2.3 The $\rightarrow_\beta$ Relation . . . . . . . . . . . . . . . . . . . 15

2.2.4 Results . . . . . . . . . . . . . . . . . . . . . . . . . . 15

2.3 Types . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16

2.3.1 Notation . . . . . . . . . . . . . . . . . . . . . . . . . 16

2.3.2 Type Inference Rules . . . . . . . . . . . . . . . . . . 17

2.3.3 Consistency of Types Under Reduction . . . . . . . . . 18

2.4 Some Lambda Terms and their Types . . . . . . . . . . . . . 18

2.5 Extended Type System . . . . . . . . . . . . . . . . . . . . . 19

**Chapter 3: The Curry-Howard Isomorphism** . . . . . . . . . . . . . . **21**

3.1 The Isomorphism . . . . . . . . . . . . . . . . . . . . . . . 21

3.2 Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

3.3 The Full Curry-Howard Isomorphism . . . . . . . . . . . . . 25

3.4 Some Remarks . . . . . . . . . . . . . . . . . . . . . . . . . 25

**Chapter 4: Classical Logic and Translation** . . . . . . . . . . . . . . **27**

4.1 Classical Logic . . . . . . . . . . . . . . . . . . . . . . . . . 27

4.2 The Kolmogorov Double Negation Translation . . . . . . . . 28

4.3 Inter-Definability of Connectives . . . . . . . . . . . . . . . 34

**Chapter 5: Continuations and Continuation Passing Style** . . . . . . . . **37**

5.1 Motivation: Control Flow . . . . . . . . . . . . . . . . . . . 37

5.1.1 Example: Scheme . . . . . . . . . . . . . . . . . . . . . . . . . 38
5.2 Control Operators . . . . . . . . . . . . . . . . . . . . . . . . . . 39
5.3 The Curry-Howard Isomorphism for Classical Logic . . . . . . . . . . 41
5.4 Continuation Passing Style . . . . . . . . . . . . . . . . . . . . . . 41
5.4.1 Type of Continuation Passing Style Translations . . . . . . . . 42
5.5 Definability of Pair and Variant Types . . . . . . . . . . . . . . . . 42

**Conclusion** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **45**

**References** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **47**

# Abstract

The Curry-Howard Isomorphism is an idea dating back to the early 20th century for relating logic and the lambda calculus, a certain abstract model of a programming language. Originally, such a relationship was only established with intuitionistic logic. However, continued work eventually produced an isomorphism between classical logic and an extension of the lambda calculus. We lay out an introduction to formal logic and the lambda calculus to motivate the basic isomorphism, present the original result, and explore the issue of extending the correspondence to classical logic in detail. The process of producing this extension and studying it illuminates some of the basic properties of classical and intuitionistic logic in a novel manner.

# Introduction

The Curry-Howard Correspondence, or Isomorphism, is a remarkable relation between programs and proofs. Specifically, it relates terms of the simply-typed lambda calculus to proofs in a natural deduction system of intuitionistic logic.

Let's bring out the basic idea of what's going on with a simple example. Suppose we are performing a proof, and we have derivations of both $p$ and $p \to q$. Then, by the familiar operation of modus ponens, we may derive $q$. On the other hand, suppose that we are writing a program of some sort, and we have an expression that computes a value of type $\phi$ along with a function that takes a parameter of type $\phi$ and returns a value of type $\psi$. In fact, this function could be thought of as having a type, something like $\phi \to \psi$. Then, it makes sense to apply this function to our expression of type $\phi$, and we would regard this whole application expression as having type $\psi$. This illustrates a structural correspondence between use of modus ponens in proofs and function application in programs. This observation is the basic idea underlying the Curry-Howard Isomorphism.

It turns out that the idea of a program that takes things of one type and produces (or, perhaps more suggestively, *constructs*) something of another type corresponds to an interpretation of intuitionistic proofs as themselves being constructions.[1] That is, we can think of a proof of $p \to q$ as a procedure for turning proofs of $p$ into proofs of $q$. Similarly, we can think of modus ponens as being a "recipe" for a new constructive proof: if we have a proof of $p$ and a proof of $p \to q$, we can combine them to get a proof of $q$.

The general idea of relating programs to proofs has been expanded to other computational calculi and formalizations of different systems of logic, and any such correspondence is usually classified under the general umbrella term "Curry-Howard Correspondence." The details of the original correspondence exploit particular features of the typed lambda calculus and of the natural deduction system for intuitionistic logic. If, however, we wish to find a similar correspondence for a different logical system, such as a Hilbert-style axiomatic system of intuitionistic logic, we must turn instead to something other than the simply-typed lambda calculus — a system called combinator logic, in this case [SU06, ch. 5]. All of this talk of "constructions" may suggest that there is something fundamentally intuitionistic about the Curry-Howard correspondence. However, this is not the case:

[1]This is known as the *Brouwer-Heyting-Kolmogorov interpretation*. See [SU06, p. 28–32] for further details.

in 1990, the idea was finally generalized to classical logic by Griffin in [Gri90].

The fact that these further isomorphisms are possible to begin with leads us to wonder how far this project can be generalized. In particular, we may ask:

1. For which systems of logic is there a reasonable computational calculus that it can be placed in correspondence with?

2. How do the particular features of these systems of logic and computational calculi relate to and illuminate each other?

So far, so good. Even if we were to stop here, we would have a fruitful mechanism for studying proofs in terms of programs, and vice versa. However, we can go even further. These various systems of logic do not stand completely alone and independent of each other. Consider, for example, the Kolmogorov double negation embedding of classical logic in intuitionistic logic. As it turns out, this can be made sense of on the computational side of the correspondence, as well. So, let's additionally ask ourselves one further question:

3. How do the connections between different systems of logic correspond to connections between different computational calculi?

This gives us a pretty good road map with which to proceed. To begin answering these questions, we need to start by laying out our formalizations of computational and logical systems in detail.

# Chapter 1

# Logic

OUR investigation must begin with a development and formalization of a system of logic. For our purposes, we will be concerned solely with propositional logic. Full-blown first-order logic with quantification can be used to form very complicated expressions, such as:

$$(\exists x)(\forall y)(\neg Fy \to Gxy \vee Gyx)$$

which says, roughly, "there is some $x$ such that any $y$ that isn't $F$ is borne $G$ by $x$ or bears $G$ to $x$." With propositional logic, we are limited to examples of the following form:

$$p \to q \wedge r$$

which we may interpret as saying "if $p$, then $q$ and $r$." That is, we have letters to stand in for propositions and sentential connectives to glue them together, but nothing else.

From a certain standpoint, propositional logic may not appear to be very interesting. It is primarily seen as the foundation for first-order and higher logics, and the grand logical results of the early 20th century — compactness, incompleteness, and so on — all deal with first-order logic. However we are headed in a direction where the richness of the structure of propositional logic will become apparent.

## 1.1 Language

We wish to rigorously define a system of propositional logic — a language together with rules of derivation. There are, in fact, many ways in which this can be done, so we will take a moment to fix a system for our use here.

First, we have our alphabet. This consists of:

1. One proposition symbol for each natural number, $p_1, p_2, p_3, \ldots$

2. The special proposition symbol $\perp$, representing the false proposition.

3. The connectives $\to$ ("if-then"), $\vee$ ("or"), and $\wedge$ ("and").

4. Left and right parentheses, ( and ).

Now we may define a language on top of this alphabet recursively as follows:

**Definition 1.1.** Let $A$ be a set of strings over our alphabet. Then we say that $A \in \mathfrak{M}$ iff the following conditions hold:

1. $p_i \in A$ for all $i \in \mathbf{N}$.
2. If $\alpha, \beta \in A$, then $(\alpha \to \beta) \in A$, $(\alpha \vee \beta) \in A$, and $(\alpha \wedge \beta) \in A$ as well.

**Definition 1.2.** We define $M$, our language of logic, as follows:

$$M = \bigcap_{A \in \mathfrak{M}} A$$

To give a more concrete sense of what our language looks like, here are some strings in $M$:

1. $(p_1 \to \bot)$
2. $((p_1 \to p_2) \to p_2)$
3. $(p_1 \to (p_1 \vee p_2))$
4. $((p_1 \to p_2) \wedge (p_2 \to p_1))$

Note that our notation is slightly minimalist. Since this bare language is not always intuitive to deal with or particularly convenient when not working at the meta level, we'll introduce some abbreviations that will make our formulae more readable. In particular, let's adopt the following conventions:

1. Distinct letters of the alphabet such as $p, q, r, \ldots$ may stand in for our subscripted proposition symbols for the sake of legibility.
2. We use $\neg p$ to abbreviate $(p \to \bot)$.

We will freely sprinkle these abbreviations throughout our discussion, although it should always be understood that they should be ultimately unwrapped into the underlying language as defined above.

## 1.2 Natural Deduction

Now that we have defined our language, we wish to define our system of deduction. For our initial purposes, we will define a system of intuitionistic logic. There are actually several ways to accomplish this, but we'll adopt a natural deduction system, which will work well for our purposes. There are other systems that are interesting in their own right, including from a Curry-Howard perspective, but this is the best place to start, and was historically the system first used in formulating the isomorphism.

Our judgments about deductions will be expressed in symbols in the following manner:

$$\frac{}{\Gamma, \alpha \vdash \alpha}\,(\mathrm{AX}) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \alpha}\,(\mathrm{PC})$$

$$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \to \beta}\,(\mathrm{DP}) \qquad \frac{\Gamma \vdash \alpha, \alpha \to \beta}{\Gamma \vdash \beta}\,(\mathrm{MP})$$

$$\frac{\Gamma \vdash \alpha}{\Gamma \vdash \alpha \vee \beta}\,(\vee \mathrm{I}_1) \qquad \frac{\Gamma \vdash \beta}{\Gamma \vdash \alpha \vee \beta}\,(\vee \mathrm{I}_2)$$

$$\frac{\Gamma \vdash \alpha \vee \beta \quad \Gamma, \alpha \vdash \delta \quad \Gamma, \beta \vdash \delta}{\Gamma \vdash \delta}\,(\vee \mathrm{E})$$

$$\frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \alpha}\,(\wedge \mathrm{E}_1) \qquad \frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \beta}\,(\wedge \mathrm{E}_2) \qquad \frac{\Gamma \vdash \alpha, \beta}{\Gamma \vdash \alpha \wedge \beta}\,(\wedge \mathrm{I})$$

Figure 1.1: Natural Deduction Rules for Intuitionistic Logic

**Definition 1.3** (Derivation Expression)**.** An expression of the form $\Gamma \vdash \alpha$ where $\Gamma$ is a finite set of sentences and $\alpha$ is a single sentence is called a *derivation expression*. This particular example can be read as saying "$\Gamma$ (syntactically) implies $\alpha$." When listing the premises on the left-hand side of the turnstile symbol, we will typically abbreviate expressions of the form "$\Gamma \cup \{\alpha\}$" as "$\Gamma, \alpha$". Similarly, we will abbreviate the conjunction of "$\Gamma \vdash \alpha$" and "$\Gamma \vdash \beta$" as "$\Gamma \vdash \alpha, \beta$".

In addition, we need notation for how we move from one such judgment to the next. When we write something of the form (where we might have $k = 0$):

$$\frac{\Gamma_1 \vdash \alpha_1 \quad \Gamma_2 \vdash \alpha_2 \quad \cdots \quad \Gamma_k \vdash \alpha_k}{\Delta \vdash \beta}\,(\mathrm{X})$$

we mean that we can deduce $\Delta \vdash \beta$ from $\Gamma_1 \vdash \alpha_1$ through $\Gamma_k \vdash \alpha_k$ using rule X. $\Gamma_1 \vdash \alpha_1$ through $\Gamma_k \vdash \alpha_k$ are said to be the *premises* of this deduction step, while $\Delta \vdash \beta$ is said to be the *conclusion*. We may write multi-step deductions recursively in a tree format, where the premise of one rule application is the conclusion of another rule application; this will be demonstrated in practice in the next section.

**Definition 1.4** (Leaf)**.** A deduction step in a tree such that its premises are not the conclusion of any other deduction step is called a *leaf*.

**Definition 1.5** (Proof)**.** A tree whose leaves have no premises is called a *proof*. If $\Gamma \vdash \alpha$ is the derivation expression at the bottom of the tree, then we say that it is a *proof of* $\Gamma \vdash \alpha$.

$$\frac{}{\Gamma, \alpha \vdash \alpha}\,(\text{AX}) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \alpha}\,(\text{PC})$$

$$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \to \beta}\,(\text{DP}) \qquad \frac{\Gamma \vdash \alpha, \alpha \to \beta}{\Gamma \vdash \beta}\,(\text{MP})$$

Figure 1.2: Natural Deduction Rules for The Implicational Fragment

**Definition 1.6** (Theorem)**.** If we can prove $\vdash \alpha$, then we say that $\alpha$ is a *theorem*.

Now, we introduce the specific rules by which we can move from one derivation expression to another. For example, if we have some background set of premises $\Gamma$ from which we can derive $\alpha$ and $\alpha \to \beta$, we should be able to derive $\beta$ from $\Gamma$. This is the familiar rule of inference known as modus ponens. Our system has ten such rules, summarized in figure 1.1 on page 5.

In addition to the rule MP for modus ponens, already discussed, we have rules corresponding to axiom introduction (AX), proof by contradiction (PC), and the deduction principle (DP). AX just states that, if $\alpha$ is among our assumptions, that implies $\alpha$. PC allows us to deduce anything from $\bot$; since $\bot$ is the false proposition, deriving it shows a contradiction. The deduction principle says that, if $\alpha$ is among our suppositions in proving $\beta$, we can prove from our other suppositions that $\alpha \to \beta$. Finally, we have introduction and elimination rules for $\vee$ and $\wedge$. Since $\vee$ corresponds to the "or" connective, showing $\alpha \vee \beta$ just requires showing one of $\alpha$ or $\beta$ and then performing $\vee I_1$ or $\vee I_2$. Deriving anything from $\alpha \vee \beta$, however, is more complicated, requiring deriving some $\delta$ from both $\alpha$ and $\beta$. Our rules for $\wedge$ are similar, except that introducing $\wedge$ is more complicated than eliminating. If we can derive $\alpha$ and $\beta$ on their own, then $\wedge I$ allows us to derive $\alpha \wedge \beta$. On the other hand, once we have $\alpha \wedge \beta$, rules $\wedge E_1$ and $\wedge E_2$ allow us to derive $\alpha$ and $\beta$, respectively.

Before moving on, we need to note one particular subset of our system that is very simple to study and produces the most natural computational interpretation possible when we go on.

**Definition 1.7** (Implicational Fragment)**.** The *implicational fragment* of intuitionistic logic is the system obtained by taking only those sentences consisting of proposition symbols other than $\bot$, implication, and parentheses, and applying only the deduction rules found in figure 1.2, namely AX, PC, DP, and MP.[1] We will also consider the *implicational fragment with falsehood*, which is the same system as before, but with $\bot$ included in the alphabet.

[1]PC, however, can only be used if we allow $\bot$ to occur in our sentences to begin with.

## 1.3 Some Examples and Results

In order to get the sense of how our system works and what sort of theorems it can prove, it will be instructive to look at some examples. In particular, it will be instructive to take a closer look at the implicational fragment, as it is both theoretically interesting and also perhaps somewhat unintuitive. Suppose that we wish to show that $p \to ((p \to q) \to q)$ is a theorem. We may do so as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p, p \to q \vdash p}\,(\mathrm{AX}) \qquad \dfrac{}{p, p \to q \vdash p \to q}\,(\mathrm{AX})}{p, p \to q \vdash q}\,(\mathrm{MP})}{p \vdash (p \to q) \to q}\,(\mathrm{DP})}{\vdash p \to ((p \to q) \to q)}\,(\mathrm{DP})$$

This illustrates the aforementioned use of our deduction rules to build "trees" representing proofs.

As it turns out, we can replace $q$ with $\bot$ and derive $p \to \neg\neg p$ as a theorem as well:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{p, p \to \bot \vdash p}\,(\mathrm{AX}) \qquad \dfrac{}{p, p \to \bot \vdash p \to \bot}\,(\mathrm{AX})}{p, p \to \bot \vdash \bot}\,(\mathrm{MP})}{p \vdash (p \to \bot) \to \bot}\,(\mathrm{DP})}{\vdash p \to ((p \to \bot) \to \bot)}\,(\mathrm{DP})$$

This means that we can introduce double negations in intuitionistic logic, although we can't do the opposite, i.e. produce $\neg\neg\alpha \to \alpha$ as a theorem for arbitrary $\alpha$.

These examples raise one important issue — namely, that form matters more than the particular proposition symbols used in our logical system. Just as we can derive $p \to p$ as a theorem, we can derive $q \to q$ as a theorem. In fact, we can always derive $\alpha \to \alpha$ as a theorem for any sentence $\alpha$. Of course, simply saying that $\alpha \to \alpha$ is a theorem without an explicit $\alpha$ doesn't make sense in our system — unless we mean it to abbreviate some specific string containing $p_i$s, it's not even a sentence. However, we will adopt this terminology to mean that $\alpha \to \alpha$ is a theorem, *whenever you choose some arbitrary but specific* $\alpha$.

Returning to the nitty-gritty details of logic, we should note that there are some specific cases where something of the form $\neg\neg\alpha \to \alpha$ is a theorem. In particular, we can prove $\neg\neg\neg\alpha \to \neg\alpha$ as a theorem:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\neg\neg\neg\alpha, \alpha, \neg\alpha \vdash \alpha}\text{(AX)} \quad \dfrac{}{\neg\neg\neg\alpha, \alpha, \neg\alpha \vdash \neg\alpha}\text{(AX)}}{\neg\neg\neg\alpha, \alpha, \neg\alpha \vdash \bot}\text{(MP)}}{\neg\neg\neg\alpha, \alpha \vdash \neg\neg\alpha}\text{(DP)} \quad \dfrac{}{\neg\neg\neg\alpha, \alpha \vdash \neg\neg\neg\alpha}\text{(AX)}}{\dfrac{\neg\neg\neg\alpha, \alpha \vdash \bot}{\neg\neg\neg\alpha \vdash \neg\alpha}\text{(DP)}}\text{(MP)}}{\vdash \neg\neg\neg\alpha \to \neg\alpha}\text{(DP)}$$

So far, the only rule allowable in the implicational fragment that we haven't seen is PC. We can demonstrate its use by proving $\alpha \to ((\alpha \to \bot) \to \beta)$ as a theorem ("if $\alpha$ is true, then $\neg\alpha$ implies $\beta$"):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha, \alpha \to \bot \vdash \alpha}\text{(AX)} \quad \dfrac{}{\alpha, \alpha \to \bot \vdash \alpha \to \bot}\text{(AX)}}{\alpha, \alpha \to \bot \vdash \bot}\text{(MP)}}{\alpha, \alpha \to \bot \vdash \beta}\text{(PC)}}{\alpha \vdash (\alpha \to \bot) \to \beta}\text{(DP)}}{\vdash \alpha \to ((\alpha \to \bot) \to \beta)}\text{(DP)}$$

More broadly, though, we can generalize to all of our system and prove theorems involving $\wedge$ and $\vee$. In particular, for example, we can derive $(\alpha \wedge \beta) \to (\alpha \vee \beta)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha \wedge \beta \vdash \alpha \wedge \beta}\text{(AX)}}{\alpha \wedge \beta \vdash \alpha}(\wedge E_1)}{\alpha \wedge \beta \vdash \alpha \vee \beta}(\vee I_1)}{\vdash (\alpha \wedge \beta) \to (\alpha \vee \beta)}\text{(DP)}$$

Note that conjunctions and disjunctions are a bit less interesting in intuitionistic logic than one may expect. In particular, $\vee E$ is often used in classical logic in conjunction with the theorem $p \vee \neg p$. However, $p \vee \neg p$ isn't an intuitionistic theorem — if it were, a few deduction steps would show that we would be able to eliminate double negations in general, which, as we just remarked, isn't possible. The only way to get something of the form $\alpha \vee \beta$ intuitionistically is to actually prove one of $\alpha$ or $\beta$ on its own, so in a sense there are no intuitionistic disjunctions that tell us anything we didn't know already, so to speak.

Anyhow, these examples have hopefully illustrated the use of our logical system to the point that the reader is comfortable and familiar with it. The main hurdle, apart from working out the details of the deduction rules and their use, is coming to grips with how intuitionism works. Most readers are likely familiar with classical logic, but intuitionism differs in some surprisingly subtle ways, some of them illustrated above. We'll come back to classical logic in chapter 4, at which point the relationship to intuitionism will be made clearer.

## 1.4 Semantics

The preceding remarks establish our language and system of deduction, but make only the most cursory and informal comments on semantics. Such a discussion would fall outside the scope of the current work, but see [SU06] or numerous other references on the subject for further details.

# Chapter 2

# The Lambda Calculus

The lambda calculus gives us an abstract model for computation and function evaluation. Historically, it arose from work of Alonzo Church first published in 1932 [Chu32] and developed further into something more closely resembling the current theory in a 1936 paper [Chu36]. In fact, the lambda calculus slightly predates the more well-known Turing machine model. There are really two components to the theory as we shall study it. One is purely a model of computation. We take terms of the lambda calculus, and talk about how they *reduce* or *evaluate* to other lambda terms. This is a rich and fruitful area of study, and understanding it to some degree is necessary to understanding the material on the Curry-Howard Isomorphism that follows. However, from our point of view, the essential component of the theory is the *type system*, where we talk about how we can assign types to terms in a way that satisfies certain reasonable constraints. One can, in fact, consider a "bare," typeless version of the theory, but the typed lambda calculus is where the central idea of our subject arises. Just as in logic, we will define a set of inference rules — if term $M$ has type $\alpha$ in a certain environment, then term $N$ has type $\beta$ in another environment, and so on.

We will now begin laying the system out in detail. While this material is separate from that in the previous chapter, the two have been written in such a way that the reader will hopefully notice some of the parallels that our theory of the Curry-Howard Isomorphism will begin to make more formal in the coming chapters.

## 2.1 Syntax

To begin, let us lay out the language and syntax within which we will develop the lambda calculus. The discussion of the basics found here is drawn largely from [Bar84, ch. 2]. For our alphabet, we have:

1. One variable symbol for each natural number $i \in \mathbf{N}$, call them $x_1, x_2, x_3, \ldots$
2. The $\lambda$ operator.
3. Parentheses, ( and ), for grouping.

As with logic, our language will be defined recursively. To do so, we construct the set $\mathfrak{L}$ as follows:

**Definition 2.1.** If $\Delta$ is a set of strings over our alphabet, then we say that $\Delta \in \mathfrak{L}$ iff:

1. $x_i \in \Delta$ for all $i \in \mathbf{N}$.
2. For any $i \in \mathbf{N}$ and $M \in \Delta$, we have $(\lambda x_i M) \in \Delta$ as well (function abstraction).
3. For any $M, N \in \Delta$, we have $MN \in \Delta$ as well (juxtaposition, i.e. function application).

**Definition 2.2** (Language of the Lambda Calculus)**.** The language $\Lambda$ of the lambda calculus is simply:

$$\Lambda = \bigcap_{\Delta \in \mathfrak{L}} \Delta$$

Terms of the form $(\lambda x_i M)$ are meant to be thought of as representing functions; in this case, a function of $x_i$ given by the rule in term $M$. Juxtaposing two terms $M$ and $N$ as $MN$ represents the application of the function represented by $M$ to the argument represented by $N$.

Some basic examples are in order at this point, just to give a preview of what's possible in our language. The following are all strings in $\Lambda$:

1. $(\lambda x_1 x_1)$
2. $x_1 x_2$
3. $(\lambda x_1 x_1)(\lambda x_1(\lambda x_2 x_1))$
4. $(\lambda x_1 x_2)$

These examples are all rather opaque at the moment, but as our discussion continues we will find out how to interpret them.

As before, it is best to adopt some notational conventions to aid us in working easily with the language at hand. In particular, when being informal, we will often write $\lambda x_i.M$ in place of $(\lambda x_i M)$, as the dot improves legibility by clearly separating the abstracted variable from the body of the function. In addition, we will frequently use different letters $x, y, \ldots$ as variable names, rather than placing subscripts on $x$.

## 2.2 Reduction

Since the lambda calculus is a model for computation, we need some way to actually move from one term to another. The primary way for getting a new lambda term from another one is through $\beta$-reduction, which will be defined in this section. There are several preliminaries and technicalities that need to be taken care of first. In particular, we need to make formal the way in which the $x$ in a term of the form $\lambda x.M$ acts as the "argument" of the function specified, and how to reason about the subtleties surrounding which occurrences of $x$ are "bound" by the function abstraction.

### 2.2.1 Free and Bound Variables

We will work by defining the free variables of a lambda term recursively. We do so by specifying a rule for the free variables of a term consisting of just a variable, then giving rules corresponding to the various ways in which we can build lambda terms. Essentially, a variable occurs free in a term unless it is captured as the parameter to a $\lambda$ expression at some point.

**Definition 2.3** (Free Variables)**.** Let $M$ be a lambda term. We shall define $FV(M)$, the *free variables of* $M$, recursively as follows:

1. If $M = T_1T_2$, then $FV(M) = FV(T_1) \cup FV(T_2)$.
2. If $M = x_i$, then $FV(M) = \{x_i\}$.
3. If $M = \lambda x_i T$, then $FV(M) = FV(T)\backslash\{x_i\}$.

**Definition 2.4** (Closed Terms and Combinators)**.** If $M$ is a term such that

$$FV(M) = \varnothing$$

then we say that $M$ is a *closed term*. A closed term of the form $\lambda x.L$ is called a *combinator*.

### 2.2.2 Substitution

When we discuss function application, we will need a way to talk about the result of substituting a lambda term in for an occurrence of a particular variable in another lambda term. Suppose we wish to reduce some term $MN$, which has the following form:

$$(\lambda x.L)N$$

Essentially, we want to take $L$ (which can be thought of as the *body* or *rule* of a function) and substitute instances of $x$ within it with $N$. For example, if we let $M = (\lambda x.x)$ and let $N = (\lambda y.z)$, then $MN$ should just reduce to $(\lambda y.z)$.

However, this is actually a somewhat misleading gloss, in that it fails to account for the various ways in which a particular variable can occur in a lambda term. For example, consider the following:

$$(\lambda x.(\lambda y.x))y$$

We don't want to simply go through replacing instances of $x$ with $y$. If we followed the naive approach, we would be left with:

$$(\lambda y.y)$$

which represents a sort of identity function — it takes its argument and gives it right back. However, the intended reading of $(\lambda x.(\lambda y.x))$ is that it takes an argument and gives back a constant function always returning that original argument. In this case, we should get something of the form:

$$(\lambda z.y)$$

The actual variable used in the $\lambda$ abstraction is, in some sense, unimportant. What matters is which variables inside the term being abstracted refer back to it, or are *bound* to it.

Now, some notation and definitions. We will use:

$$M[x := N]$$

to denote the result of substituting all (appropriate) occurrences of $x$ with the term $N$ in the term $M$. As with FV, we will define this concept recursively based on the form that $M$ can take.

**Definition 2.5** (Substitution)**.** We define $M[x_i := N]$ recursively as follows:

1. If $M = x_j$, with $i \neq j$, then $M[x_i := N] = M$.
2. If $M = x_i$, then $M[x_i := N] = N$.
3. If $M = T_1T_2$, then $M[x_i := N] = T_1[x_i := N]T_2[x_i := N]$.
4. If $M = \lambda x_i T$, then $M[x_i := N] = M$.
5. If $M = \lambda x_j T$, with $i \neq j$ and $x_j \notin FV(N)$, then $M[x_i := N] = \lambda x_j T[x_i := N]$.
6. Finally, if $M = \lambda x_j T$, with $i \neq j$ and $x_j \in FV(N)$, then we define:

$$M[x_i := N] = (\lambda x_k T[x_j := x_k])[x_i := N]$$

where we choose $x_k \notin FV(N) \cup FV(T)$.

Let's run through the example of using this definition to determine:

$$(\lambda y.x)[x := y]$$

The variable of the abstraction occurs free in the term being substituted, since, by definition 2.3, we have $y \in FV(y)$. Applying rule 6, we reduce this to finding:

$$(\lambda z.x)[x := y]$$

We then apply rule 5 to get our intended result of $(\lambda z.y)$.

### 2.2.3 The $\to_\beta$ Relation

Terms may be $\beta$-*reduced* by taking instances of juxtaposition — terms of the form $MN$ where $M$ and $N$ are themselves terms — and removing them by performing substitution, subject to the following definitions.

**Definition 2.6** ($\beta$-redex)**.** A term of the form $MN$, where $M$ is of the form $(\lambda x_i L)$ for some term $L$, is called a $\beta$-*redex*.

**Definition 2.7** ($\beta$ Reduction)**.** We define the $\to_\beta$ relation on lambda terms as the minimal relation satisfying the following conditions:

1. $(\lambda x_i M)N \to_\beta M[x_i := N]$
2. When $M \to_\beta M'$, we have $MN \to_\beta M'N$.
3. When $N \to_\beta N'$, we have $MN \to_\beta MN'$ .
4. When $M \to_\beta M'$, we have $(\lambda x_i M) \to_\beta (\lambda x_i M')$.

**Definition 2.8** (The $\twoheadrightarrow_\beta$ Relation)**.** If $M$ and $N$ are terms such that:

$$M \to_\beta M_1 \to_\beta M_2 \to_\beta \cdots \to_\beta M_k \to_\beta N$$

for some terms $M_1, M_2, \ldots, M_k$ (including the degenerate cases where $k = 0$ or $M = N$), then we say:

$$M \twoheadrightarrow_\beta N$$

### 2.2.4 Results

Before moving on, we need to briefly cite some important results about $\beta$-reduction. These will not figure heavily in our development of the subject, but they are worth noting. First, we need to informally define what it means for two lambda terms to be "essentially the same."

**Definition 2.9** (Alpha Equivalence)**.** For terms $N_1$ and $N_2$, we say that $N_1 \sim_\alpha N_2$ if $N_2$ can be obtained by substituting in new bound variables in $N_1$ in such a way as to avoid capture.

**Theorem 2.10** (Church-Rosser)**.** *Suppose that $M \twoheadrightarrow_\beta N_1$ and $M \twoheadrightarrow_\beta N_2$. Then, there exist some $N_1'$ and $N_2'$ with $N_1' \sim_\alpha N_2'$ such that $N_1 \twoheadrightarrow_\beta N_1'$ and $N_2 \twoheadrightarrow_\beta N_2'$. This is sometimes called the* diamond property*, after the shape of the diagram frequently used to illustrate it.*

Detailed proofs of this result, which involve many technical details that would lead us too far astray from our main topic, may be found in [Bar84, p. 277–283] and [SU06, p. 12–14].

**Definition 2.11** (Normal Form)**.** We say that a term $M$ is in *normal form* if there is no $N$ such that $M \to_\beta N$. If $M \twoheadrightarrow_\beta M'$ and $M'$ is in normal form, then we say that $M'$ is the *normal form of* $M$.

**Theorem 2.12** (Uniqueness of Normal Form). *If $N_1$ and $N_2$ are normal forms of $M$, then $N_1 \sim_\alpha N_2$. Therefore, it makes sense to talk about* the *normal form of a lambda term.*

*Proof.* Corollary of 2.10. □

Note that not all lambda terms have a normal form. For example, consider:

$$(\lambda f.ff)(\lambda f.ff)$$

This term has only one beta redex and beta reduces to itself, so it never beta reduces to a term in normal form.

## 2.3 Types

What we have described so far is the untyped lambda calculus. However, we wish to tag our terms with extra information about what "sort" of thing it is that they represent. As a result, certain terms will be "well-typed," representing a sensible combination of data and operations. Others will fail to type correctly due to nonsensical combinations or uses of different subterms. This idea is easily motivated by practical considerations in writing and debugging software, but of course it also has great theoretical repercussions. This aspect of the theory is where the analogy with logic enters the picture.

### 2.3.1 Notation

To begin, we need to establish how we are going to represent types. In a slightly abbreviated version of our now-routine process, we lay down some language with which to work.

**Definition 2.13** (Type Expressions). A *type expression* is a string generated recursively through the following rules:

1. We have *atomic type* symbols corresponding to the natural numbers: $\iota_1, \iota_2, \iota_3, \ldots$
2. We have a special atomic type symbol, $\perp$, representing the *empty type*, or *void type*.
3. For type expressions $\tau$ and $\sigma$, we have a further *arrow type* (or *function type*) $(\tau \to \sigma)$. The $\to$ symbol is conventionally right associative, so we will write $\tau \to \sigma \to \upsilon$ for $\tau \to (\sigma \to \upsilon)$.

We will speak almost exclusively in terms of general type expressions $\tau$, $\sigma$, and so on, rarely ever making explicit reference to atomic types. In addition, we will abbreviate $\tau \to \perp$ as $\neg\tau$ when convenient.

Furthermore, we wish to represent some sort of background environment in which types are assigned. We need to start out with the types of individual variable symbols as a given and move from there. We will usually use lowercase Greek letters to represent types expressions.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\ (\text{Var}) \qquad \frac{\Gamma \vdash M : \bot}{\Gamma \vdash *(M) : \tau}\ (\text{Mir})$$

$$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash (\lambda x M) : (\tau \to \sigma)}\ (\text{Abs}) \qquad \frac{\Gamma \vdash N : \tau, M : (\tau \to \sigma)}{\Gamma \vdash MN : \sigma}\ (\text{App})$$

Figure 2.1: Type Inference Rules

**Definition 2.14** (Type Context)**.** By a *type context* $\Gamma$, we mean something of the form:

$$\Gamma = \{M_1 : \tau_1, M_2 : \tau_2, \ldots, M_n : \tau_n\}$$

where each $M_i \in \Lambda$ and each $\tau_i$ is a type expression. By $M_i : \gamma$, we mean that $M_i$ has type $\gamma$.

What a context gives us, then, is a set of associations of variables with types.[1] In fact, $\Gamma$ is not unlike a function taking variable symbols to types, so we can define an analog of domain and range for an environment:

**Definition 2.15** (Domain and Range of Type Contexts)**.** We use $\text{dom}(\Gamma)$ and $\text{rg}(\Gamma)$ to denote the following sets:

$$\begin{aligned} \text{dom}(\Gamma) &= \{M \mid M : \tau \in \Gamma \text{ for some } \tau\} \\ \text{rg}(\Gamma) &= \{\tau \mid M : \tau \in \Gamma \text{ for some } M\} \end{aligned}$$

## 2.3.2 Type Inference Rules

We now need ways to deduce the overall type of a term from a given environment. To that end, we introduce a set of inference rules — based on the types of terms in a known environment, they allow us to deduce the types of further terms built up from terms that are already types. These rules are given in figure 2.1, following the same notation and conventions as in our presentation of logic in section 1.2. If we can deduce $\Gamma \vdash M : \tau$, then we say that $M$ has type $\tau$ in the environment $\Gamma$.

In detail, the interpretation of the rules is as follows: Var states that, if we assume $x$ has type $\tau$, then we can conclude that $x$ has type $\tau$. Abs tells us that, if $M$ has type $\sigma$ on the assumption that $x$ has type $\tau$, then the resulting lambda abstraction $\lambda x.M$ has type $\tau \to \sigma$. Rule App says that, if $N$ has type $\tau$ in our environment and $M$ has type $\tau \to \sigma$, then the application $MN$ has type $\sigma$ in the same environment.

Finally, we have our rule Mir, short for "miracle," which deserves comment. What this says is that, if we can assign type $\bot$ to some term $M$, then there is a

[1]If one wishes to keep the set theory involved as formal and correct as possible, environments can be defined as sets of appropriate ordered pairs.

term denoted by $*(M)$ of type $\tau$, for any choice of $\tau$. These terms do not serve a role in our computational system, because to obtain such a thing we first need to find a term occupying the empty type, which is impossible. This sounds rather mysterious, but the coming presentation of the Curry-Howard Isomorphism and its generalizations will hopefully make this matter more concrete. The key point that is worth keeping in mind here when trying to work through the confusion is that we aren't saying that there is actually any such thing as a "miracle" lambda term; we are saying that, were it the case that the empty type was inhabited, then we could infer whatever we wished about types of terms. If we had liked, the rule could have stipulated that any term could have been given any type, but having a distinct way of referring to these hypothetical terms is useful. Again, this is purely a device in our rules of inference, much the same way that the special proposition $\perp$ plays a special role in our system of logic.

### 2.3.3 Consistency of Types Under Reduction

It is important now that we pause and note a basic result about our type system, namely that it is (in some sense) congruent with our already-established notion of reduction.

**Theorem 2.16.** *Suppose that* $M$ *and* $N$ *are lambda terms such that* $M \twoheadrightarrow_\beta N$. *Then if* $\Gamma \vdash M : \tau$, *it follows that* $\Gamma \vdash N : \tau$.

A proof of this result may be found in [SU06, p. 59]. This says that our type system respects $\beta$ reduction — that is, continuing on with the reduction of a term doesn't alter its type.

## 2.4 Some Lambda Terms and their Types

It will be instructive to the reader unfamiliar with the subject matter to consider some basic examples, along with some way of interpreting them. These are all commonly discussed lambda terms; they are described, among other places, in [SU06, p. 11].

Take, for example, the term $\mathbf{I} \equiv \lambda x.x$, which behaves sort of like an "identity function." If we take some arbitrary term $M$, then we have:

$$\mathbf{I}M = (\lambda x.x)M \to_\beta M$$

What do our type inference rules say about this term? Intuitively, it would seem that $\mathbf{I}$ has any type of the form $\tau \to \tau$, as it takes whatever term is given to it and hands it right back. This is, indeed, correct, as shown by the following application of type inference rules:

$$\dfrac{\dfrac{}{x : \tau \vdash x : \tau}\,(\mathrm{Var})}{\vdash (\lambda x.x) : (\tau \to \tau)}\,(\mathrm{Abs})$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \mathrm{in}_1^{\tau\vee\sigma} M : \tau \vee \sigma}\,(\vee I_1) \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \mathrm{in}_2^{\tau\vee\sigma} M : \tau \vee \sigma}\,(\vee I_2)$$

$$\frac{\Gamma \vdash M : \tau \vee \sigma \quad \Gamma, x_i : \tau \vdash B : \delta \quad \Gamma, x_j : \sigma \vdash C : \delta}{\Gamma \vdash \mathbf{case}\ M\ \mathbf{of}\ x \Rightarrow B\ \mathbf{or}\ y \Rightarrow C : \delta}\,(\vee E)$$

$$\frac{\Gamma \vdash M : \tau \wedge \sigma}{\Gamma \vdash \pi_1 M : \tau}\,(\wedge E_1) \qquad \frac{\Gamma \vdash M : \tau \wedge \sigma}{\Gamma \vdash \pi_2 M : \sigma}\,(\wedge E_2)$$

$$\frac{\Gamma \vdash M : \tau, N : \sigma}{\Gamma \vdash \langle M, N\rangle : \tau \wedge \sigma}\,(\wedge I)$$

Figure 2.2: Extended Type Inference Rules

Now consider $\mathbf{K} \equiv (\lambda x.(\lambda y.x))$. Again, thinking in intuitive terms of conventional mathematical functions, **K** takes an argument and gives back the corresponding constant function. Take some arbitrary term $M$, and apply **K** to it to get:

$$\mathbf{K}M = (\lambda x.(\lambda y.x))M \to_\beta \lambda y.M$$

As far as types go, **K** takes an argument of some type, and then gives back a function that takes any given input and hands back something of the type that **K** was initially given. We formalize this by the application of our type inference rules as follows:

$$\dfrac{\dfrac{\dfrac{}{x : \tau, y : \sigma \vdash x : \tau}\,(\mathrm{Var})}{x : \tau \vdash (\lambda y.x) : (\sigma \to \tau)}\,(\mathrm{Abs})}{\vdash (\lambda x.(\lambda y.x)) : (\tau \to (\sigma \to \tau))}\,(\mathrm{Abs})$$

As it turns out, not all terms in $\Lambda$ can be consistently assigned a type. Consider $\lambda x.xx$. The term $xx$ would need to be assigned a type through rule App. This means that we must have $\Gamma \vdash x : \tau \to \sigma$ for some type expressions $\tau$ and $\sigma$. However, we would also need to have $\Gamma \vdash x : \tau$ for the very same value of $\tau$. This is certainly an odd example, although it should be noted that self-application does have its place in the untyped lambda calculus (i.e. it would make perfect sense to have $\lambda x.xx$ as a subterm in some larger untyped term that $\beta$-reduced to another term).

## 2.5 Extended Type System

We will now introduce some more sophisticated types into our system, along with corresponding type inference rules. In particular, we will define *pair* and *variant* types. We will denote the pair and variant types formed from $\tau$ and $\sigma$ as $\tau \wedge \sigma$ and $\tau \vee \sigma$, respectively.[2] The pair formed from terms $M$ and $N$ is itself a new lambda term, which is written as $\langle M, N \rangle$. We introduce two new combinators, $\pi_1$ and $\pi_2$, called *projections*. These have associated reduction rules, as follows:

$$\pi_1\langle M, N\rangle \to_\chi M$$
$$\pi_2\langle M, N\rangle \to_\chi N$$

We write the associated type for a pair of terms of type $\tau$ and $\sigma$ as $\tau \times \sigma$.

Variants, on the other hand, are things that can be of one type or another. The variant type of types $\tau$ and $\sigma$ is designated $\tau \vee \sigma$. For all pairs of types $\tau$ and $\sigma$, we get a new pair of combinators, $\mathrm{in}_1^{\tau\vee\sigma}$ and $\mathrm{in}_2^{\tau\vee\sigma}$. The intuitive gloss on these is that they take a term and "lift" it to the variant containing it — that is, they take something of type $\tau$ to the corresponding term that's either an $\tau$ or a $\sigma$. Then, we can "pull apart" a variant with a **case** term, which has a new reduction rule of its own:

$$\textbf{case } \mathrm{in}_1^{\tau\vee\sigma} M \textbf{ of } x \Rightarrow B \textbf{ or } y \Rightarrow C \to_\chi B[x := M]$$
$$\textbf{case } \mathrm{in}_2^{\tau\vee\sigma} M \textbf{ of } x \Rightarrow B \textbf{ or } y \Rightarrow C \to_\chi C[y := M]$$

**Definition 2.17.** We will refer to our original system as *the simply-typed lambda calculus*. This new system, with added terms and types, will be referred to as the *extended typed lambda calculus*.

These terms and the new special combinators introduced to handle them make it possible to represent many computations in the lambda calculus in a way that looks more natural when compared to typical programming languages. That said, our ultimate motivation in introducing these types is to provide a lambda calculus analog of conjunction and disjunction, as will be seen in the next chapter, 3. We will extend the lambda calculus yet again in chapter 5, albeit in a way that makes it possible do away with pair and variant types as primitives.

[2] A word of warning: this notation is rather nonstandard, and employed solely for the purpose of making the Curry-Howard Isomorphism easier to present. Typically, the pair is written as $\tau \times \sigma$ and the variant is written as $\tau + \sigma$, as in [Pie02, p. 126].

# Chapter 3

# The Curry-Howard Isomorphism

Now that we have established the necessary prerequisites from logic and the lambda calculus, we may begin to explore the Curry-Howard Isomorphism itself. The preceding chapters were presented in such a way that the reader hopefully has some intuitive awareness of the connection already. Figures 1.1 and 2.2 have clear similarities — see figure 3.1 — and our goal is now to formalize this connection.

## 3.1 The Isomorphism

We shall now state and prove the central theorem of our subject.

**Theorem 3.1** (The Curry-Howard Isomorphism)**.** *In a type context* $\Gamma$*, there is a lambda term* $M$ *in the simply typed lambda calculus such that* $\Gamma \vdash M : \phi$ *iff* $\mathrm{rg}(\Gamma) \vdash \phi$ *in the implicational fragment of intuitionistic logic with falsehood.*

*Proof.* This result sounds like it should, in some sense, be simple, and it is tempting to invite the reader to just inspect figure 3.1 and think about it. However, we can prove this theorem in an interesting way that illuminates what's going on and hints at an algorithm for producing terms corresponding to proofs. We will show the direction going from logic to the lambda calculus, although the other direction can be filled in with an isomorphic set of definitions and arguments.

We will proceed by induction on proof trees. First, we need to expand a little bit on our vocabulary with regards to proof tree. We will use $\mathfrak{T}_1, \mathfrak{T}_2, \ldots$ for trees and $\mathcal{A}, \mathcal{B}, \mathcal{C}, \ldots$ for derivation expressions (possibly including the empty derivation expression).

**Definition 3.2** (Depth of a Tree)**.** We define the depth of a tree recursively in the natural way. A tree of the form:

$$\frac{\mathcal{A}}{\mathcal{B}}$$

has depth $1$, while a tree of the form:

$$\frac{\mathfrak{T}_1 \quad \mathfrak{T}_2 \quad \cdots \quad \mathfrak{T}_n}{\mathcal{A}}$$

$$\frac{}{\Gamma, \alpha \vdash \alpha}\,(\mathrm{AX}) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \alpha}\,(\mathrm{PC})$$

$$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \to \beta}\,(\mathrm{DP}) \qquad \frac{\Gamma \vdash \alpha, \alpha \to \beta}{\Gamma \vdash \beta}\,(\mathrm{MP})$$

(a) Implicational Fragment of Intuitionistic Logic

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\,(\mathrm{Var}) \qquad \frac{\Gamma \vdash M : \bot}{\Gamma \vdash \ast(M) : \tau}\,(\mathrm{Mir})$$

$$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash (\lambda x M) : (\tau \to \sigma)}\,(\mathrm{Abs}) \qquad \frac{\Gamma \vdash N : \tau, M : (\tau \to \sigma)}{\Gamma \vdash MN : \sigma}\,(\mathrm{App})$$

(b) Simply-Typed Lambda Calculus Type Inference Rules

Figure 3.1: Natural Deduction and Type Inference Rules Together

has depth $1 + \max_{1 \leqslant i \leqslant n} \{\text{depth of } \mathfrak{T}_i\}$.

**Definition 3.3** (Minimal Depth Proof)**.** Let $\mathfrak{T}$ be a tree representing a proof of $\Gamma \vdash \alpha$. Then we say that $\mathfrak{T}$ is a *minimal depth proof* of $\Gamma \vdash \alpha$ if there does not exist any tree $\mathfrak{T}'$ proving $\Gamma \vdash \alpha$ with depth $\mathfrak{T}' <$ depth $\mathfrak{T}$. We say that the depth of a minimal depth proof tree of $\Gamma \vdash \alpha$ is the *minimal proof depth* of that derivation expression.[1]

Now we can proceed by performing induction on minimal proof depth of derivation expressions. As our base case, consider those derivation expressions with minimal proof depth $1$. These will have proof trees of the form:

$$\frac{}{\alpha_1, \alpha_2, \ldots, \alpha_n \vdash \alpha_i}\,(\mathrm{AX})$$

Then, we just note that the lambda term $x_i$ can be typed as follows:

$$\frac{}{x_1 : \alpha_1, x_2 : \alpha_2, \ldots, x_n : \alpha_n \vdash x_i : \alpha_i}\,(\mathrm{Var})$$

Now, suppose that the result holds true for all provable derivation expressions with a minimal proof depth of $n$ or less. Let $\mathrm{rg}(\Gamma) \vdash \alpha$ be a derivation expression

[1]It's worth pointing out that, while this definition may sound ominous to those of the constructivist persuasion, nothing fishy is going on here. The set of depths of proof trees of a given derivation expression is a subset of the natural numbers, so it certainly has a least element by conventional mathematical reasoning. However, one could also argue this by showing that, if we know that we can prove $\Gamma \vdash \alpha$, then we can actually compute a minimum depth proof easily by performing a breadth-first search through proofs.

with minimal proof depth $n + 1$, and let $\mathfrak{T}$ be a minimal proof tree of $\mathrm{rg}(\Gamma) \vdash \alpha$. Now, we work by cases. If $\mathfrak{T}$ is of the form:

$$\frac{\mathfrak{T}'}{\mathrm{rg}(\Gamma) \vdash \beta \to \gamma}\ (\mathrm{DP})$$

where $\mathfrak{T}'$ is a proof tree for $\mathrm{rg}(\Gamma), \beta \vdash \gamma$, then we take a lambda term of the form $(\lambda x_i.M)$, where $M$ has type $\gamma$ in a type context $\Gamma, x_i : \beta$. Such a term is guaranteed to exist by induction, as $\mathrm{rg}(\Gamma), \beta \vdash \gamma$ has minimal proof depth at most $n$. We then type this term as follows:

$$\frac{\Gamma, x_i : \alpha \vdash M : \gamma}{\Gamma \vdash (\lambda x_i.M) : (\beta \to \gamma)}\ (\mathrm{Abs})$$

Now suppose that $\mathfrak{T}$ is of the form:

$$\frac{\mathfrak{T}' \quad \mathfrak{T}''}{\mathrm{rg}(\Gamma) \vdash \gamma}\ (\mathrm{MP})$$

where $\mathfrak{T}'$ is a proof tree of $\mathrm{rg}(\Gamma) \vdash \beta$ and $\mathfrak{T}''$ is a proof tree of $\mathrm{rg}(\Gamma) \vdash \beta \to \gamma$. Then, by induction, we introduce lambda terms $M$ and $N$ such that $\Gamma \vdash M : \beta$ and $\Gamma \vdash N : \beta \to \gamma$ and then type the term $NM$ as follows:

$$\frac{\Gamma \vdash M : \beta \quad \Gamma \vdash N : (\beta \to \gamma)}{\Gamma \vdash NM : \gamma}\ (\mathrm{App})$$

Finally, it may be the case that $\mathfrak{T}$ is of the form:

$$\frac{\mathfrak{T}'}{\mathrm{rg}(\Gamma) \vdash \alpha}\ (\mathrm{PC})$$

where $\mathfrak{T}'$ is a proof tree of $\mathrm{rg}(\Gamma) \vdash \bot$. Then, by induction, we have some lambda term $M$ such that $\Gamma \vdash M : \bot$, and then produce the term $\ast(M)$ and type it as follows:

$$\frac{\Gamma \vdash M : \bot}{\Gamma \vdash \ast(M) : \alpha}\ (\mathrm{Mir})$$

This exhausts the possible cases, and shows that whenever we can prove $\mathrm{rg}(\Gamma) \vdash \alpha$ in the implicational fragment of intuitionistic logic with falsehood, we can find a term $M$ such that $\Gamma \vdash M : \alpha$. As previously noted, the reverse direction holds by a similar argument by cases. □

Note that our proof involved recursively building a lambda term whose type corresponds to the proposition we were starting from, and whose type can be determined through a series of type inference rules corresponding to the steps involved in proving the proposition. We can think of lambda terms as encoding proofs, type-checking a term as corresponding to verifying the correctness of a proof.

## 3.2 Examples

At this point, one may wish to inspect actual theorems of intuitionistic logic and see the lambda terms corresponding to their proofs. For example, take $p \to (q \to p)$. We may prove this theorem as follows:

$$\dfrac{\dfrac{\dfrac{}{p, q \vdash p}\,(\text{AX})}{p \vdash q \to p}\,(\text{DP})}{\vdash p \to (q \to p)}\,(\text{DP})$$

Now consider the $\mathbf{K}$ combinator, introduced in section 2.4:

$$\mathbf{K} \equiv (\lambda x.(\lambda y.x))$$

From our intuitive grasp of the lambda calculus and type systems, we can see that this can be consistently assigned any type of the form $\alpha \to (\beta \to \alpha)$. We may verify this by performing the following deduction using our type inference rules:

$$\dfrac{\dfrac{\dfrac{}{x : \alpha, y : \beta \vdash x : \alpha}\,(\text{Var})}{x : \alpha \vdash (\lambda y.x) : (\beta \to \alpha)}\,(\text{Abs})}{\vdash (\lambda x.(\lambda y.x)) : (\alpha \to (\beta \to \alpha))}\,(\text{Abs})$$

Note that the structure of our program mimics the structure of our proof, as each type inference rule corresponds to one of the ways in which we may build lambda terms.

Now for a slightly more complex (and important) case. Any sentence of the form

$$p \to ((p \to q) \to q)$$

is a theorem of intuitionistic logic. In particular,

$$p \to ((p \to \bot) \to \bot)$$

is a theorem; note that this can be abbreviated as $p \to \neg\neg p$. What, then, is the Curry-Howard interpretation of this result? Consider the lambda term $(\lambda x.(\lambda y.yx))$. We may infer its type as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \alpha, y : (\alpha \to \beta) \vdash x : \alpha}\,(\text{Var}) \qquad \dfrac{}{x : \alpha, y : (\alpha \to \beta) \vdash y : (\alpha \to \beta)}\,(\text{Var})}{x : \alpha, y : (\alpha \to \beta) \vdash yx : \beta}\,(\text{App})}{x : \alpha \vdash (\lambda y.yx) : ((\alpha \to \beta) \to \beta)}\,(\text{Abs})}{\vdash (\lambda x.(\lambda y.yx)) : (\alpha \to ((\alpha \to \beta) \to \beta))}\,(\text{Abs})$$

Finally, let's look at a simple example that will illustrate an important point. Consider the term $\lambda y.\mathbf{I}y$. We can assign this term the type $\alpha \to \alpha$, as follows:

$$\dfrac{\dfrac{\dfrac{}{y:\alpha\vdash y:\alpha}\,(\mathrm{Var}) \qquad \dfrac{\dfrac{}{x:\alpha,y:\alpha\vdash x:\alpha}\,(\mathrm{Var})}{y:\alpha\vdash(\lambda x.x):\alpha\to\alpha}\,(\mathrm{Abs})}{y:\alpha\vdash(\lambda x.x)y:\alpha}\,(\mathrm{App})}{\vdash(\lambda y.(\lambda x.x)y):\alpha\to\alpha}\,(\mathrm{Abs})$$

However, the term $\mathbf{I}$ itself can also be assigned the type $\alpha \to \alpha$ through the following steps:

$$\dfrac{\dfrac{}{x:\alpha\vdash x:\alpha}\,(\mathrm{Var})}{\vdash(\lambda x.x):\alpha\to\alpha}\,(\mathrm{Abs})$$

These two terms encode two different proofs of $\alpha \to \alpha$, although the second one is certainly, in some sense, more direct. We can formalize this by noting that $\lambda y.\mathbf{I}y \to_\beta \mathbf{I}$, so $\mathbf{I}$ is actually the normal form of the first term that we looked at. The correspondence between proofs and lambda terms allows us to talk about normalizing *proofs*. Exploring this idea is one possible direction to go with this subject matter, and the idea of normalization is important in proof theory.

## 3.3 The Full Curry-Howard Isomorphism

We may generalize our isomorphism beyond just the implicational fragment using the extended type system that we developed at the end of the previous chapter.

**Theorem 3.4** (Full Curry-Howard Isomorphism)**.** *In a type context $\Gamma$, there is a lambda term $M$ in the extended typed lambda calculus such that $\Gamma \vdash M : \phi$ iff $\mathrm{rg}(\Gamma) \vdash \phi$ in intuitionistic logic.*

The proof of this result just involves generalizing the proof of theorem 3.1, adding cases for our rules involving conjunction and disjunction.

## 3.4 Some Remarks

If it feels like very little has happened in this chapter, it's because, in some sense, that's the case; the beauty of the Curry-Howard Isomorphism is its obviousness and simplicity. However, noting a simple similarity between diagrams does not make a full understanding of the subject. The interpretation of computation in terms of logic and vice versa is very subtle and takes time to understand fully. It has been my goal to help illuminate the matter by providing a proof of theorem 3.1 that relies on an actual algorithm for determining the lambda term corresponding to a given proof. The examples presented are intended to be helpful, but going back to puzzle over them further (or to investigate examples of your own) would almost certainly be helpful.

# Chapter 4

# Classical Logic and Translation

We are now ready to move beyond the case of intuitionistic logic alone and consider ways of generalizing the Curry-Howard Isomorphism to other systems of logic. The most obvious candidate would be our usual standard baseline system of logic, namely classical logic.

As it turns out, classical and intuitionistic logic are related in a number of surprising ways. Historically, the development and adoption of intuitionistic logic was motivated by the perceived failings of classical logic, either on purely philosophical grounds or in terms of avoiding contradiction in the useful mathematical theories. However, intuitionism is not as isolated from the woes of classical logic as its early proponents may have liked. In particular, given a proof of a contradiction in Peano (classical) arithmetic, one may apply a translation to produce a proof of a contradiction in Heyting (intuitionistic) arithmetic [FO10, p. 21]. If classical arithmetic is in trouble in this regard, then intuitionistic arithmetic is no better off.

While this is arguably a blow to the philosophical advocate of intuitionism, the translation by which this is accomplished is a great upshot to those of us who wish to study various logical systems and their relations. There are actually several such translations — see [FO10] for an overview. However, we will primarily be concerned with the first such translation to be discovered, that of Kolmogorov.

## 4.1 Classical Logic

There are several equivalent ways in which our system of natural deduction given in figure 1.1 can be extended to give classical logic. Our approach will be to add a single new rule, DN, which allows for double negation elimination. The resulting set of natural deduction rules is shown in figure 4.1.

This additional rule is simple, but it allows us to prove a lot of theorems classically that can't be shown intuitionistically. This rule will be most familiar to the working mathematician for the following application: suppose that we can derive $\Gamma, \neg\alpha \vdash \bot$ (i.e. $\neg\alpha$ implies a contradiction). Then, we want to be able to somehow derive $\Gamma \vdash \alpha$. To that end, we can perform DP in order to get $\Gamma \vdash (\alpha \to \bot) \to \bot$ or, in abbreviated form, $\Gamma \vdash \neg\neg\alpha$. Then, we just perform DN to get $\Gamma \vdash \alpha$, allowing

$$\frac{}{\Gamma, \alpha \vdash \alpha}\,(\mathrm{AX}) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \alpha}\,(\mathrm{PC}) \qquad \frac{\Gamma \vdash \neg\neg\alpha}{\Gamma \vdash \alpha}\,(\mathrm{DN})$$

$$\frac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \to \beta}\,(\mathrm{DP}) \qquad \frac{\Gamma \vdash \alpha, \alpha \to \beta}{\Gamma \vdash \beta}\,(\mathrm{MP})$$

$$\frac{\Gamma \vdash \alpha}{\Gamma \vdash \alpha \vee \beta}\,(\vee \mathrm{I}_1) \qquad \frac{\Gamma \vdash \beta}{\Gamma \vdash \alpha \vee \beta}\,(\vee \mathrm{I}_2)$$

$$\frac{\Gamma \vdash \alpha \vee \beta \quad \Gamma, \alpha \vdash \delta \quad \Gamma, \beta \vdash \delta}{\Gamma \vdash \delta}\,(\vee \mathrm{E})$$

$$\frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \alpha}\,(\wedge \mathrm{E}_1) \qquad \frac{\Gamma \vdash \alpha \wedge \beta}{\Gamma \vdash \beta}\,(\wedge \mathrm{E}_2) \qquad \frac{\Gamma \vdash \alpha, \beta}{\Gamma \vdash \alpha \wedge \beta}\,(\wedge \mathrm{I})$$

Figure 4.1: Natural Deduction Schemes for Classical Logic

us to perform something like a conventional proof by contradiction.

A word on notation: often, we will want to disambiguate between classical and intuitionistic logic when discussing them side-by-side. If it isn't clear from the context, we will use $\vdash_I$ for judgments deduced in intuitionistic logic, and $\vdash_C$ for judgments in classical logic.

## 4.2 The Kolmogorov Double Negation Translation

The Kolmogorov Double Negation translation gives us a way of translating sentences of logic into new sentences, in such a way that the new sentence is related to the original in an interesting way. Following [SU06, p. 154], we may recursively define an operator $K$ which performs this translation.

**Definition 4.1** (Kolmogorov Translation)**.** Let $\alpha$ be a sentence consisting solely of an atomic proposition, and let $\beta$ and $\gamma$ be any sentences. Then, define:

$$\begin{aligned} K(\alpha) &= \neg\neg\alpha \\ K(\beta \to \gamma) &= \neg\neg(K(\beta) \to K(\gamma)) \\ K(\beta \wedge \gamma) &= \neg\neg(K(\beta) \wedge K(\gamma)) \\ K(\beta \vee \gamma) &= \neg\neg(K(\beta) \vee K(\gamma)) \end{aligned}$$

In addition to looking at the whole translation, we could restrict ourselves to just the implicational fragment. However, the interesting results in question hold generally, so it is best to move on and simply note that we could consider a simpler version of $K$ if we so desired.

We may now explore how $K$ relates classical and intuitionistic logic, first looking at the classical relation between a sentence and its image under $K$.

**Theorem 4.2.** *Let $\phi$ be a sentence. Then $\phi \vdash_C K(\phi)$ and $K(\phi) \vdash_C \phi$.*

*Proof.* By induction on depth of nested connectives in a sentence. The result clearly holds if $\phi$ is a sentence consisting solely of an atomic proposition, by rules DN and PC.

Now, suppose we have some sentence $\phi$ that does not take the form $p_i$ for any $i \in \mathbf{N}$. Then, we either have $\phi = \alpha \to \beta$, $\phi = \alpha \vee \beta$, or $\phi = \alpha \wedge \beta$, where $\alpha$ and $\beta$ are sentences with a strictly lower depth of connective nesting. By induction, we have $\alpha \vdash_C K(\alpha)$, $K(\alpha) \vdash_C \alpha$, $\beta \vdash_C K(\beta)$, and $K(\beta) \vdash_C \beta$.

Step back for a second and note that, in general, if $\alpha$ and $\alpha'$ are classically equivalent and $\beta$ and $\beta'$ are classically equivalent, we have:

$$\alpha \to \beta \vdash_C \alpha' \to \beta'$$
$$\alpha \vee \beta \vdash_C \alpha' \vee \beta'$$
$$\alpha \wedge \beta \vdash_C \alpha' \wedge \beta'$$

Consider the case of the $\to$ connective. Since $\alpha$ and $\alpha'$ are classically equivalent, we can derive $\vdash_C \alpha \to \alpha'$ and $\vdash \alpha' \to \alpha$, along with the analogous results for $\beta$ and $\beta'$. The proof tree showing the end result can be seen in figure 4.2 on page 30.

The corresponding proofs for $\vee$ and $\wedge$ are slightly different, taking advantage of the introduction and elimination rules for those connectives, but follow in a similar manner.

We may now complete the proof by using the aforementioned facts, double negation introduction and elimination, and our inductive hypothesis to show that $\phi \vdash_C K(\phi)$ and $K(\phi) \vdash_C \phi$. Suppose that $\phi$ is of the form $\alpha \to \beta$. By our inductive hypothesis, we know that $\alpha$ and $K(\alpha)$ are classically equivalent, as are $\beta$ and $K(\beta)$. By the above remarks, this means that we can derive $\alpha \to \beta \vdash K(\alpha) \to K(\beta)$. By our remark in section 1.3, we can go through some steps to perform double negation introduction and get $\alpha \to \beta \vdash \neg\neg(K(\alpha) \to K(\beta))$. However, by definition, $\neg\neg(K(\alpha) \to K(\beta)) = K(\alpha \to \beta)$, so we have $\alpha \to \beta \vdash K(\alpha \to \beta)$.

Similarly, we can consider the other direction. If we assume $K(\alpha \to \beta)$, we can use DN to get $K(\alpha) \to K(\beta)$, then apply our inductive hypothesis and the above remarks to get $\alpha \to \beta$.

The cases where $\phi$ takes the form of a conjunction or disjunction follow similarly. □

In order to show how the Kolmogorov Translation gives us a relationship between classical and intuitionistic logic, we first need to note some properties of the translation itself within the intuitionistic context.

**Lemma 4.3.** *The expressions that follow are all derivable in intuitionistic logic; items 1–5, and other pertinent intuitionistic tautologies, can be found in [FO10, p. 23].*

$$
\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha \to \beta, \alpha' \vdash \alpha'}\,(\mathrm{AX}) \qquad \dfrac{\vdots}{\alpha \to \beta, \alpha' \vdash \alpha' \to \alpha}}{\alpha \to \beta, \alpha' \vdash \alpha}\,(\mathrm{MP}) \qquad \dfrac{}{\alpha \to \beta, \alpha' \vdash \alpha \to \beta}\,(\mathrm{AX})}{\alpha \to \beta, \alpha' \vdash \beta}\,(\mathrm{MP}) \qquad \dfrac{\vdots}{\alpha \to \beta, \alpha' \vdash \beta \to \beta'}}{\dfrac{\alpha \to \beta, \alpha' \vdash \beta'}{\alpha \to \beta \vdash \alpha' \to \beta'}\,(\mathrm{DP})}\,(\mathrm{MP})
$$

By hypothesis, we can derive $\vdash \alpha' \to \alpha$ and $\vdash \beta \to \beta'$, so the dots elide those corresponding proof trees. Furthermore, note that, if we can derive $\Gamma \vdash \gamma$ and $\Gamma \subseteq \Delta$, we can also derive $\Delta \vdash \gamma$, so there is no worry about deriving $\alpha' \to \alpha$ and $\beta \to \beta'$ in a larger environment.

Figure 4.2: Proof Tree for Theorem 4.2

1. $\mathsf{K}(\alpha) \to \mathsf{K}(\beta) \vdash_I \mathsf{K}(\alpha \to \beta)$
2. $\mathsf{K}(\alpha \to \beta) \vdash_I \mathsf{K}(\alpha) \to \mathsf{K}(\beta)$
3. $\mathsf{K}(\alpha) \vee \mathsf{K}(\beta) \vdash_I \mathsf{K}(\alpha \vee \beta)$
4. $\mathsf{K}(\alpha) \wedge \mathsf{K}(\beta) \vdash_I \mathsf{K}(\alpha \wedge \beta)$
5. $\mathsf{K}(\alpha \wedge \beta) \vdash_I \mathsf{K}(\alpha) \wedge \mathsf{K}(\beta)$
6. $(\alpha \to \neg\beta) \to (\neg\neg\alpha \to \neg\beta)$

*Proof.* Results 1, 3, and 4 all follow fairly easily from the fact that we can perform double negation introduction in intuitionistic logic. We prove 2 explicitly by giving a tree in figure 4.3 on page 32; this shows that a derivation expression of the relevant form is derivable intuitionistically.

The one result that stands out as being unlike the others is 6. This is a sort of fussy technical detail, but we will need to deploy it while proving our next major result. We give the proof tree here, where we define $\Gamma = \{\alpha \to \neg\beta, \neg\neg\alpha, \beta, \alpha\}$ in order to save space:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\alpha \to \neg\beta, \neg\neg\alpha, \beta \vdash \neg\neg\alpha}\,(\text{AX}) \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash \beta}\,(\text{AX}) \quad \dfrac{\dfrac{}{\Gamma \vdash \alpha}\,(\text{AX}) \quad \dfrac{}{\Gamma \vdash \alpha \to \neg\beta}\,(\text{AX})}{\Gamma \vdash \neg\beta}\,(\text{MP})}{\Gamma \vdash \bot}}{\alpha \to \neg\beta, \neg\neg\alpha, \beta \vdash \neg\alpha}\,(\text{DP})}{\alpha \to \neg\beta, \neg\neg\alpha, \beta \vdash \bot}\,(\text{MP})}{\alpha \to \neg\beta, \neg\neg\alpha \vdash \neg\beta}\,(\text{DP})}{\alpha \to \neg\beta \vdash \neg\neg\alpha \to \neg\beta}\,(\text{DP})}{\vdash (\alpha \to \neg\beta) \to (\neg\neg\alpha \to \neg\beta)}\,(\text{DP})$$

We leave the others for verification — the reader will hopefully be understanding of the reluctance to explicitly create any more proof trees of that magnitude! □

**Theorem 4.4.** *Let $\phi$ be a sentence. Then $\vdash_I \mathsf{K}(\phi)$ iff $\vdash_C \phi$.*

*Proof.* Suppose $\vdash_I \mathsf{K}(\phi)$. Since everything provable intuitionistically is provable classically, we have $\vdash_C \mathsf{K}(\phi)$, which in turn gives us $\vdash_C \phi$ by theorem 4.2.

Now suppose that $\vdash_C \phi$. We will give a recursive procedure for turning a proof of $\vdash_C \phi$ into a proof of $\vdash_I \mathsf{K}(\phi)$. To formalize this, begin by letting $\mathfrak{T}$ be a proof tree for $\vdash_C \phi$. Now, we argue by cases of the root of $\mathfrak{T}$. On the one hand, it could be of the form:

$$\dfrac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \to \beta}\,(\text{DP})$$

$$\dfrac{\dfrac{\dfrac{}{\Delta \vdash \neg\neg(\neg\neg\alpha \to \neg\neg\beta)}\,(\mathrm{AX}) \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash \neg\beta}\,(\mathrm{AX}) \qquad \dfrac{\dfrac{}{\Gamma \vdash \neg\neg\alpha \to \neg\neg\beta}\,(\mathrm{AX}) \qquad \dfrac{}{\Gamma \vdash \neg\neg\alpha}\,(\mathrm{AX})}{\Gamma \vdash \neg\neg\beta}\,(\mathrm{MP})}{\Gamma \vdash \bot}\,(\mathrm{MP})}{\Delta \vdash \neg(\neg\neg\alpha \to \neg\neg\beta)}\,(\mathrm{DP})}{\dfrac{\Delta \vdash \bot}{\Theta \vdash \neg\neg\beta}\,(\mathrm{DP})}\,(\mathrm{MP})$$

$$\dfrac{\Theta \vdash \neg\neg\beta}{\neg\neg(\neg\neg\alpha \to \neg\neg\beta) \vdash \neg\neg\alpha \to \neg\neg\beta}\,(\mathrm{DP})$$

In order to save space, we define:

$$\begin{aligned}
\Gamma &= \{\neg\neg\alpha, \neg\beta, \neg\neg\alpha \to \neg\neg\beta, \neg\neg(\neg\neg\alpha \to \neg\neg\beta)\} \\
\Delta &= \{\neg\neg\alpha, \neg\beta, \neg\neg(\neg\neg\alpha \to \neg\neg\beta)\} \\
\Theta &= \{\neg\neg\alpha, \neg\neg(\neg\neg\alpha \to \neg\neg\beta)\}
\end{aligned}$$

Figure 4.3: Proof for Part 2 of Lemma 4.3

By induction, we have $\mathsf{K}(\Gamma), \mathsf{K}(\alpha) \vdash \mathsf{K}(\beta)$, so we can apply lemma 4.3 item 1 and replace the root with:

$$\cfrac{\cfrac{\cfrac{\mathsf{K}(\Gamma), \mathsf{K}(\alpha) \vdash \mathsf{K}(\beta)}{\mathsf{K}(\Gamma) \vdash \mathsf{K}(\alpha) \to \mathsf{K}(\beta)}\,(\text{DP})}{\vdots}}{\mathsf{K}(\Gamma) \vdash \mathsf{K}(\alpha \to \beta)}$$

where the dots represent the necessary deduction steps that we'd have to fill in to show the relevant result from lemma 4.3 part 2.

The root of $\mathfrak{T}$ could also be of the form:

$$\frac{\Gamma \vdash \alpha, \alpha \to \beta}{\Gamma \vdash \beta}\,(\text{MP})$$

Then, by induction, we can assume $\mathsf{K}(\Gamma) \vdash \mathsf{K}(\alpha), \mathsf{K}(\alpha \to \beta)$. Again, we can apply lemma 4.3 and replace the root of the tree as follows:

$$\cfrac{\cfrac{\cfrac{\mathsf{K}(\Gamma) \vdash \mathsf{K}(\alpha), \mathsf{K}(\alpha \to \beta)}{\vdots}}{\mathsf{K}(\Gamma) \vdash \mathsf{K}(\alpha), \mathsf{K}(\alpha) \to \mathsf{K}(\beta)}}{\mathsf{K}(\Gamma) \vdash \mathsf{K}(\alpha \to \beta)}\,(\text{MP})$$

This covers the rules for introducing and eliminating the $\to$ connective — the cases for $\vee$ and $\wedge$ follow similarly from lemma 4.3. The one tricky case is that of $\vee\mathsf{E}$, since $\vee$ doesn't behave *quite* like $\wedge$. Suppose that the root of $\mathfrak{T}$ takes the form:

$$\frac{\Gamma \vdash \alpha \vee \beta \quad \Gamma, \alpha \vdash \delta \quad \Gamma, \beta \vdash \delta}{\Gamma \vdash \delta}\,(\vee\text{E})$$

Now, by induction, we may assume that all of the following hold:

$$\begin{gathered} \mathsf{K}(\Gamma) \vdash \mathsf{K}(\alpha \vee \beta) \\ \mathsf{K}(\Gamma), \mathsf{K}(\alpha) \vdash \mathsf{K}(\delta) \\ \mathsf{K}(\Gamma), \mathsf{K}(\beta) \vdash \mathsf{K}(\delta) \end{gathered}$$

This means that we can also use $\vee$E to derive:

$$\mathsf{K}(\Gamma), \mathsf{K}(\alpha) \vee \mathsf{K}(\beta) \vdash \mathsf{K}(\delta)$$

by application of our induction hypotheses. Further reworking and application of lemma 4.3 item 6 (noting that $\mathsf{K}(\delta)$ has the form $\neg\gamma$ for some $\gamma$) gives us:

$$\mathsf{K}(\Gamma) \vdash \neg\neg(\mathsf{K}(\alpha) \vee \mathsf{K}(\beta)) \to \mathsf{K}(\delta)$$

which, by rewriting $\neg\neg(\mathsf{K}(\alpha) \vee \mathsf{K}(\beta))$ as $\mathsf{K}(\alpha \vee \beta)$ and an application of MP, gives us the result we are looking for.

This leaves applications of AX, PC, and DN to take care of. AX is trivial. For PC, suppose that $\mathfrak{T}$ has a root of the form:

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \alpha}\ (\text{PC})$$

Then, by induction, we may assume $K(\Gamma) \vdash K(\bot)$. Now just note that $K(\bot) = ((\bot \to \bot) \to \bot)$. Furthermore, we have $\vdash_I \bot \to \bot$, so we can derive $K(\bot) \vdash_I \bot$. Perform this derivation, and then just replace the root of $\mathfrak{T}$ with:

$$\frac{K(\Gamma) \vdash \bot}{K(\Gamma) \vdash K(\alpha)}\ (\text{PC})$$

Instances of DN are handled as another special case. Suppose the root of $\mathfrak{T}$ is of the form:

$$\frac{\Gamma \vdash \neg\neg\alpha}{\Gamma \vdash \alpha}\ (\text{DN})$$

Then, by induction, we have:

$$K(\Gamma) \vdash K(\neg\neg\alpha)$$

which, by selective expansion of some abbreviations and definitions, gives us:

$$\begin{aligned} K(\neg\neg\alpha) &= K((\alpha \to \bot) \to \bot) \\ &\vdash_I K(\alpha \to \bot) \to K(\bot) \end{aligned}$$

By repeated application of lemma 4.3 item 2 and other facts:

$$\vdash_I K(\alpha) \to K(\bot) \to K(\bot)$$

Since $\bot$ and $K(\bot)$ are intuitionistically equivalent, we may perform the following derivation (the details of which are omitted):

$$\begin{aligned} &\vdash_I K(\alpha) \to \bot \to \bot \\ &= \neg\neg K(\alpha) \end{aligned}$$

Now, just note that $\neg\neg K(\alpha)$ is of the form $\neg\neg\neg\beta$ for some sentence $\beta$. In general, we have $\vdash_I \neg\neg\neg\beta \to \neg\beta$ by one of our examples from section 1.3, so we can derive $\neg\neg K(\alpha) \vdash_I K(\alpha)$. After plugging in all of the necessary work, we can replace the root of $\mathfrak{T}$ with the final step in our derivation of $K(\Gamma) \vdash K(\alpha)$.

The covers the last of the possible cases, completing the proof. □

## 4.3 Inter-Definability of Connectives

One distinguishing feature of classical logic is that we can define conjunction and disjunction in terms of the conditional and negation.

**Definition 4.5.** We may take our language for our formal system of logic and modify the alphabet by removing $\vee$ and $\wedge$, re-introducing them in the form of abbreviations as follows:

1. We introduce $p \vee q$ as an abbreviation for $\neg(\neg p \to \neg q)$.

2. Similarly, we use $p \wedge q$ as an abbreviation for $\neg(p \to \neg q)$.

Now we just need to check that conjunction and disjunction behave properly when defined as such.

**Theorem 4.6.** *The following are all theorems of classical logic with our newly redefined notions of conjunction and disjunction:*

*1.* $p \to p \vee q$

*2.* $q \to p \vee q$

*3.* $p \wedge q \to p$

*4.* $p \wedge q \to q$

*Proof.* We will prove 1 and leave the rest for routine verification by the reader. Removing our abbreviations, we see that we are trying to prove $p \to ((p \to \bot) \to ((q \to \bot) \to \bot))$. We may do this with the following tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p, p \to \bot \vdash p}\text{(AX)} \qquad \dfrac{}{p, p \to \bot \vdash p \to \bot}\text{(AX)}}{p, p \to \bot \vdash \bot}\text{(MP)}}{p, p \to \bot \vdash (q \to \bot) \to \bot}\text{(PC)}}{p \vdash (p \to \bot) \to (q \to \bot) \to \bot}\text{(DP)}}{\vdash p \to ((p \to \bot) \to (q \to \bot) \to \bot)}\text{(DP)}$$

The other proofs proceed in a similar fashion. □

# Chapter 5

# Continuations and Continuation Passing Style

At this point, the reader (hopefully!) expects some computational analog to the discussion in the previous chapter. We want something that is to classical logic as the lambda calculus is to intuitionistic logic. There are two levels on which we hope to make a correspondence. For starters, we wish to establish a connection like the Curry-Howard Isomorphism between classical logic and a related notion of computation. Second, we wish to find a computational meaning for the Kolmogorov Double Negation Translation, i.e. a relation between our original lambda calculus and whatever new system we develop that mirrors the translation between systems of logic. As it turns out, we can deliver on both fronts by introducing a notion of control operators based on *continuations*.

## 5.1 Motivation: Control Flow

Ideally, our augmented version of the lambda calculus will have some relatively intuitive computational interpretation. As it turns out, we can easily motivate the additions we are going to make with some very concrete examples. The lambda calculus on its own is sufficiently powerful, but it lacks some key features that would allow it to emulate idiomatic computer code. Consider, for example, situations in which we may wish to explicitly move execution from one point in our code to another. Overuse of such operators (in the case of unrestricted "goto" statements found in some early languages) quickly leads to spaghetti code that is nigh impossible to reason about, but there are situations in which some form of "jump" may in fact be tasteful. One such example may be if we wish to raise and handle exceptions in our code. There may be instances where there is some particular occurrence, such as a division by zero, that may occur within a complex computation. In such a case, we may wish to immediately abort the current computation and return control to the part of the program that invoked our code so that the condition can be handled in some appropriate way.

```
(define (f throw)
  (let ((fst (get-number-from-user))
        (snd (get-number-from-user)))
    (if (eq? snd 0)
        (throw 'div-by-zero-exception)
        (/ fst snd))))

(call/cc f)
```

Listing 5.1: Example of Continuation-Based Exception Handling

### 5.1.1 Example: Scheme

One instantiation of the sort of control flow we are discussing can be found in the programming language Scheme [Spe07]. Suppose that we wish to implement something akin to the aforementioned exception handling mechanism. An example of such usage can be seen in listing 5.1.

This hypothetical example takes two user-supplied numbers and divides them, handling the case where the divisor may be zero. The function f being defined takes an argument called throw. When the built-in function `call/cc` is called on `f`, the function `f` itself is called and supplied with another function as argument. This other function is a device called a *continuation* that encapsulates the current control state of the program, and returns to its previous position when called, abandoning the current flow of execution. The argument supplied to the continuation when it is called is then returned by `call/cc`. If the call to `f` terminates normally, then the result obtained from that evaluation is returned by `call/cc` instead. So, if our user dutifully gives two numbers, the final S-expression in our program will evaluate to their quotient. If, on the other hand, the divisor is zero, we will get back the symbol `'div-by-zero-exception`.

In addition to providing a means by which usual ideas of control flow can be expressed in functional languages (as discussed in [Wad92]), continuations play a larger role. In particular, they provide a means by which such languages can be more readily compiled into conventional machine language or bytecode. See, for example, [App92].

There is one important problem with our example, namely that Scheme lacks a type system of the sophistication that we need. In particular, functions don't have types, which leaves us without a way of forming a Curry-Howard correspondence. We can, however, discuss how we might assign `call/cc` a type in a hypothetical language like Scheme. In fact, there is a typed variant of the Scheme dialect Racket. We may look at how it handles the matter; an interactive session at the Racket REPL is shown in listing 5.2.

The type of `call/cc` reported back is that of a function whose argument is itself a function and whose return type is something corresponding roughly to an $\alpha \vee \beta$ variant type. The function that `call/cc` takes has some $\alpha$ to `Nothing` (roughly anal-

```
> racket -I typed/racket
Welcome to Racket v5.3.3.
-> (:print-type call/cc)
(All (a b) (((a -> Nothing) -> b) -> (U a b)))
```

Listing 5.2: Typed Racket Handling `call/cc`

ogous to our $\bot$) function (the continuation) as its argument and gives back something of type $\beta$. Let's dissect this a little bit. The continuation is something of type $\alpha \to \bot$, and since it returns type $\bot$, it never finishes evaluating if called; instead the flow of the program is altered. An invocation of the continuation with an argument causes evaluation of the continuation-using function to stop and the argument that it was called with to be given back by `call/cc`. The `call/cc` function then returns either the value that the continuation was invoked with or, if the continuation wasn't called, the result of the evaluation of the function `call/cc` was given.

## 5.2 Control Operators

We will now develop an extension of the lambda calculus that incorporates the features we need, roughly corresponding to the `call/cc` mechanism in Scheme outlined above. The original development of such a system and its relation to classical logic is due to Griffin [Gri90]. Various such systems have been developed and formalized since Griffin published his paper, but it will be most convenient for us to follow the system developed in [SU98, p. 132–141].

The one new symbol that we add to the lambda calculus is $\Delta$. We extend the language $\Lambda$ as a whole to a new language $\Lambda_\Delta$ by saying that, for any term $M$, we also have a term $(\Delta x_i M)$.

**Definition 5.1** ($\Delta$ Reduction)**.** We define the $\to_\Delta$ relation to be the minimal relation on $\Lambda_\Delta$ terms satisfying:

1. $(\Delta x.M)N \to_\Delta (\Delta z.M[x \coloneqq (\lambda y.z(yN))])$
2. $(\Delta x.xM) \to_\Delta M$ whenever $x \notin FV(M)$
3. $(\Delta x.x(\Delta y.M)) \to_\Delta (\Delta z.M[x \coloneqq z][y \coloneqq z])$

We also need to extend the $\to_\beta$ relation slightly. Let our extended $\to_\beta$ be the minimal relation satisfying all of the properties outlined in definition 2.7, in addition to the stipulation that $(\Delta x_i M) \to_\beta (\Delta x_i N)$ whenever $M \to_\beta N$. This just means that we can $\beta$-reduce within $\Delta$ terms in the usual manner, just as we can $\beta$-reduce within $\lambda$ terms.

Finally, we define a rule for typing $\Delta$ terms, shown in figure 5.1. This extends the type inference rules given in figure 2.2. Note that this allows us to perform the

$$\frac{\Gamma, x_i : \tau \to \bot \vdash M : \bot}{\Gamma \vdash (\Delta x_i M) : \tau} \text{(Ctrl)}$$

Figure 5.1: Type Inference Rule for Δ Terms

analog of double negation elimination in our type system. Suppose we have some term $N$ with $N : (\tau \to \bot) \to \bot$. Then, under the assumption that there is some $x$ with $x : (\tau \to \bot)$, we can get $Nx : \bot$. Therefore, we may conclude that $\Delta x.Nx : \tau$. Note that this final term doesn't directly $\Delta$ reduce to anything, but $Mx$ will $\beta$ reduce in such a way that we eventually get something of the form $xN$, in which case our $\Delta$ reduction rules will be applicable. This allows us to extend the Curry-Howard Isomorphism in theorem 5.2.

The introduction of $\Delta$ is meant to provide our calculus with something corresponding to `call/cc`. In order to clarify and motivate what's going on, we can provide a $\Lambda_\Delta$ analog to the earlier example given in Scheme. Suppose that we have some EInt (extended integer) type whose values are either normal integers or a special **!** value, which we'll use to indicate an exception. Furthermore, suppose that we have a div function for integer division, a term eq0 that tests whether an EInt is $0$, and an if term. These can all be constructed in our calculus from the ground up, but for the purposes of this example we will treat them as given. Now, consider the following term:

$$f \equiv \lambda m.\lambda n.\Delta t.t(\text{if}\,(\text{eq0}\,n)(\Delta x.t\,\mathbf{!})(\text{div}\,m\,n))$$

Suppose that we give $f$ two values, $i_1$ and $i_2$, by taking the term $fi_1i_2$ and reducing it. In the case where $i_2$ is nonzero, reduction should proceed normally and give us the value of $i_1$ divided by $i_2$:

$$\begin{aligned} fi_1i_2 &\to_\beta \Delta t.t(\text{if}\,(\text{eq0}\,i_2)(\Delta x.t\,\mathbf{!})(\text{div}\,i_1\,i_2)) \\ &\twoheadrightarrow_\beta \Delta t.t(\text{div}\,i_1\,i_2) \end{aligned}$$

By $\Delta$ reduction rule 2:

$$\to_\Delta (\text{div}\,i_1\,i_2)$$

On the other hand, if $i_2$ is zero, the reduction is different:

$$\begin{aligned} fi_1i_2 &\to \Delta t.t(\text{if}\,(\text{eq0}\,i_2)(\Delta x.t\,\mathbf{!})(\text{div}\,i_1\,i_2)) \\ &\twoheadrightarrow_\beta \Delta t.t(\Delta x.t\,\mathbf{!}) \end{aligned}$$

By $\Delta$ reduction rule 3:

$$\to_\Delta \Delta z.z\,\mathbf{!}$$

By $\Delta$ reduction rule 2:

$$\to_\Delta \mathbf{!}$$

## 5.3 The Curry-Howard Isomorphism for Classical Logic

Our newly defined calculus with the $\Delta$ operator allows us to give a Curry-Howard Isomorphism for classical logic.

**Theorem 5.2** (The Curry-Howard Isomorphism for Classical Logic)**.** *In a type context $\Gamma$, there is a $\Lambda_\Delta$ term $M$ such that $\Gamma \vdash M : \phi$ iff* $\mathrm{rg}(\Gamma) \vdash_C \phi$.

*Proof.* All that we need to do is slightly generalize theorem 3.4. As with our proof of theorem 3.1, we will show that $\mathrm{rg}(\Gamma) \vdash_C \phi$ implies the existence of a $\Lambda_\Delta$ term $M$ such that $\Gamma \vdash M : \phi$.

Let $\mathfrak{T}$ be a minimal depth proof tree of $\mathrm{rg}(\Gamma) \vdash_C \phi$. The one new case that we need to consider is that where the root of $\mathfrak{T}$ is of the form:

$$\frac{\mathfrak{T}'}{\mathrm{rg}(\Gamma) \vdash \phi}\ (\mathrm{DN})$$

where $\mathfrak{T}'$ is a proof tree of $\mathrm{rg}(\Gamma) \vdash \neg\neg\phi$. By induction, we have some term $M$ such that $\Gamma \vdash M : ((\phi \to \bot) \to \bot)$. Since $M$ has a function type, it must be of the form $\lambda x.N$, with the type inference going as follows:

$$\frac{\Gamma, x : (\phi \to \bot) \vdash N : \bot}{\Gamma \vdash (\lambda x.N) : ((\phi \to \bot) \to \bot)}\ (\mathrm{Abs})$$

Then, we can type the term $\Delta x.N$ as follows:

$$\frac{\Gamma, x : \phi \to \bot \vdash N : \bot}{\Gamma \vdash (\Delta x.N) : \phi}$$

which gives us a $\Lambda_\Delta$ term of type $\phi$ in $\Gamma$, as desired. □

## 5.4 Continuation Passing Style

As it turns out, the original lambda calculus can express anything that our newly-enhanced system can. Not only is that the case, but there is an easy recursive algorithm for turning a term in our new system with instances of $\Delta$ into a pure lambda term.

**Definition 5.3** (Continuation Passing Style Translation)**.** The *continuation passing style translation*, or *CPS translation* for short, $t$, is an operator that takes $\Lambda_\Delta$ terms to pure $\Lambda$ terms, defined recursively as follows:

1. If $M = x_i$, when $t(M) = (\lambda k.x_i k)$.
2. If $M = (\lambda x_i N)$, then $t(M) = \lambda k.k(\lambda x_i.t(N))$.
3. If $L = MN$, then $t(L) = \lambda k.t(M)(\lambda x.xt(N)k)$.
4. If $M = (\Delta x_i.M)$, then $t(M) = \lambda k.(\lambda x_i.t(M))(\lambda a.a(\lambda b.\lambda c.c(bk)))(\lambda z.z)$.

### 5.4.1 Type of Continuation Passing Style Translations

As it turns out, the type of a term $M$ and the type of $t(M)$ are related by the Kolmogorov Translation.

**Theorem 5.4.** *Suppose that $\Gamma \vdash M : \tau$ in the $\Lambda_\Delta$ calculus. Then $K(\Gamma) \vdash t(M) : K(\tau)$ in the simply typed lambda calculus. Here $K(\Gamma)$ is to be understood as the type context derived from $\Gamma$ by taking each type assignment of the form $N : \tau$ and replacing it with $N : K(\tau)$.*

Proving this result in its entirety would be laborious and just result in getting bogged down with technical details. However, we may illustrate the procedure with a specific example.

Suppose that we have a term of the form $MN$, where we have an environment $\Gamma$ with $M : \sigma \to \tau$ and $N : \sigma$, so that $MN$ has type $\tau$ overall. Now we need to take $t(MN)$, as given in the definition above, and type it in the environment $K(\Gamma)$, where we have $t(M) : \neg\neg(\neg\neg\sigma \to \neg\neg\tau)$ and $t(N) : \neg\neg\sigma$. However, we have already done this! Well, not quite… but figure 4.3 with the very last step removed is the logical analog of the type inferences for $t(MN)$ under the Curry-Howard Isomorphism.[1] Sure enough, we get that $K(\Gamma) \vdash t(MN) : \neg\neg\beta$.

This is a huge upshot. All we need to do is see that the CPS translation can be viewed not only as a translation between terms, but also as a translation between proofs. Then, if a term $L$ encodes a proof of $\phi$, we will have $t(L)$ encoding a proof of $K(\phi)$. Doing the proof in full detail would simply involve confirming that $t$ performs exactly the transformation on proofs that we developed in order to prove theorem 4.4.

## 5.5 Definability of Pair and Variant Types

Just as classical logic allowed us to define conjunctions and disjunctions in terms of negation and the material conditional in section 4.3, the addition of a control operator to the lambda calculus allows us to define pairs and variants in simpler terms.

**Definition 5.5.** We may remove pair and variant terms from the lambda calculus, replacing them as abbreviations of the following, as described in [SU98, p. 135]:

1. We write $\lambda x.xPQ$ in place of $\langle P, Q\rangle$.
2. We write $\Delta k.M(\lambda x_1.\lambda x_2.kx_i)$ for $\pi_i(M)$.
3. We write $\lambda x_1.\lambda x_2.x_iM$ for $\mathrm{in}_i(M)$.
4. We write $\Delta k.M(\lambda x.kB)(\lambda y.kC)$ in place of **case** $M$ **of** $x \Rightarrow B$ **or** $y \Rightarrow C$.

[1] In fact, the proof in figure 4.3 was developed by working backwards from the CPS translation.

We need to verify that these definitions do, in fact make sense and cohere with our existing rules for the reduction of pair and variant terms. For example, we should be able to reduce $\pi_1\langle P, Q\rangle$ to $P$. Using the definitions above, we can write $\pi_1\langle P, Q\rangle = \Delta k.(\lambda x.xPQ)(\lambda x_1.\lambda x_2.kx_1)$, and then reduce:

$$\begin{aligned}
\Delta k.(\lambda x.xPQ)(\lambda x_1.\lambda x_2.kx_1) &\to_\beta \Delta k.(\lambda x_1.\lambda x_2.kx_1)PQ \\
&\to_\beta \Delta k.(\lambda x_2.kP)Q \\
&\to_\beta \Delta k.kP \\
&\to_\Delta P
\end{aligned}$$

Furthermore, we may note that these definitions also match definition 4.5 from the perspective of our type system; if we assign $P : \tau$ and $Q : \sigma$, we get

$$\langle P, Q\rangle : (\tau \to (\sigma \to \bot)) \to \bot$$

which can also be written as $\neg(\tau \to \neg\sigma)$.

What we have, then, is an interesting result in programming languages motivated by the Curry-Howard Isomorphism — introducing control operators to your language somehow gives it the ability to express pairs and variants without adding them explicitly. Similarly, the intuition behind the control flow interpretation of the $\Delta$ operator may be applied back to our thinking about classical logic and how it is that it can be defined entirely in terms of $\to$ and $\neg$.

# Conclusion

This thesis has provided a basic outline of the Curry-Howard Isomorphism and explored some of the subtleties of the relationship. Our main study case has been that of classical logic, and the connection between the Kolmogorov translation presented in chapter 4 and the CPS translation from chapter 5 is one of the main points that the reader should come away understanding and appreciating. This example really shows how the Curry-Howard Isomorphism can go deeper than a simple relationship between proofs in one system of logic and terms in one computational calculus.

There are several different directions in which the subject can be taken further. One particularly interesting example is that of automated theorem proving and proof assistant systems. Coq is a prominent example of a proof assistant; it relies on a Curry-Howard Isomorphism with a system called the calculus of inductive constructions [BC04]. Coq is notable in that it has been used to formalize a complete proof of the Four Color Theorem (rather than using a computer to simply verify some cases in an ad-hoc manner), as described in a paper by Georges Gonthier from Microsoft Research [Gon08]. Isabella is another prominent example of a proof assistant, and is inspired heavily by an earlier proof system based on the type system of the ML programming language [Pau88].

In addition, there are further questions at the basic level of logic that can be posed. For example, we studied how the fact that classical logic can be formulated with only $\neg$ and $\rightarrow$ in section 4.3, and saw that this has a Curry-Howard interpretation in section 5.5. However, it turns out that we can go even further, defining all of classical logic in terms of one connective, the Sheffer stoke, which is also known as the nand connective and sometimes written $\uparrow$ [End72, p. 51]. Presumably this has some sort of Curry-Howard interpretation as well, but there does not appear to be any reference to such a development in the literature.

# References

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, Cambridge, 1992.

[Bar84] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., Amsterdam, revised edition, 1984.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, Berlin, 2004.

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, April 1932.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.

[End72] Herbert B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, New York, 1972.

[FO10] Gilda Ferreira and Paulo Oliva. On various negative translations. In Steffen van Bakel, Stefano Berardi, and Ulrich Berger, editors, *CL&C*, volume 47 of *EPTCS*, pages 21–33, 2010.

[Gon08] Georges Gonthier. Formal proof—the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008.

[Gri90] Timothy G. Griffin. A formulae-as-type notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '90, pages 47–58, New York, NY, USA, 1990. ACM.

[Pau88] Lawrence C. Paulson. The foundation of a generic theorem prover. Technical Report UCAM-CL-TR-130, University of Cambridge, Computer Laboratory, March 1988. Online at `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-130.pdf`.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, 2002.

[Spe07] Michael Sperber. *Revised$^6$ Report on the Algorithmic Language Scheme*, 2007. Online at `http://www.r6rs.org/final/html/r6rs/r6rs.html`.

[SU98] Morten Heine B. Sørensen and Paweł Urzyczyn. Lectures on the Curry-Howard isomorphism, 1998. Draft version of [SU06] online at `http://folli.loria.fr/cds/1999/library/pdf/curry-howard.pdf`.

[SU06] Morten Heine Sørensen and Paweł Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematica*. Elsevier, Amsterdam, 2006.

[Wad92] Philip Wadler. The essence of functional programming, 1992. Online at `http://homepages.inf.ed.ac.uk/wadler/papers/essence/essence.ps`.