UNIVERSITY OF CALGARY

# The Curry-Howard Correspondence

by

Husna Farooqui

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF ARTS

GRADUATE PROGRAM IN PHILOSOPHY

CALGARY, ALBERTA

AUGUST, 2021

# Abstract

Outside of logic, computer science, and mathematics, the Curry-Howard correspondence is ordinarily described as a deep connection between proofs and programs. However, without sufficient requisite background knowledge, such a description can be mystifying and contribute to the correspondence's general obscurity. In this thesis, we attempt to introduce the Curry-Howard correspondence by presenting an elementary correspondence between the extended simply-typed lambda calculus and intuitionistic natural deduction. Our introduction aims to provide the necessary theoretical background to understand the basic correspondence, which can help bridge the gap between non-specialists and the more advanced literature on the topic. Such an introduction also serves to better demonstrate and clarify the correspondence's significance, which is a topic we explore towards the end.

To introduce the correspondence, we introduce the lambda calculus and the simply-typed lambda calculus. Moreover, we introduce natural deduction and subsequently develop a novel sequent-style natural deduction for reasons philosophical as well as logically advantageous. As a result, we methodically prove a full isomorphism between our system of natural deduction and the extended simply-typed lambda calculus.

# Preface

This thesis is original, unpublished, independent work by the author, Husna Farooqui.

# Acknowledgments

Though I cannot give proper thanks to everyone who inspires my gratitude, which is a good and sorry thing, I try my best and so have the following:

First, I give thanks to the members of the department—professors, students, and administrators alike—for enriching, memorable time spent on the 12th floor of the SS Tower. My committee, Nicole Wyatt and Jeremy Fantl, deserve special named thanks and recognition. I am further obliged to Jeremy and my peers, Chloe Stephenson and Weidong Sun, for reading perilous, early drafts of this thesis. Nothing is possible without fun—a bold metaphysical statement—and I have Yoshiki Kobasigawa and fellow grad students to thank for fun.

Second, a 'thank you' to friends and family for (long-distance) love as always. To Chloe and Baudelaire, I reserve extra honourable mention for their cozy company in our year shared together.

Third and most of all, I am supremely grateful to my supervisor, Richard Zach, who was unbelievably patient with me. I owe these two years to him, for I could not have done so much without his humour, support, encouragement, dedication, knowledge, guidance, q.e.d. etc.

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

As the title suggests, this thesis is about the Curry-Howard correspondence. The correspondence often appears under many different names, and some of its other guises are *the Curry-Howard isomorphism, the Curry-Howard-Lambek correspondence, the proofs-as-programs paradigm, propositions-as-types,* and *formulae-as-types,* among others. This thesis is about those too.

As with all specialist subjects, an obstacle presented by the current literature is that the correspondence is discussed in ways that are often epistemically inaccessible to non-specialists. So though our focus will be on the correspondence, our approach will be such that non-specialists with some basic logical background are able to follow along. Our hope is that by presenting an elementary version of the correspondence, we provide a small stepping stone that will help readers approach more advanced literature. We also hope our relatively light-handed treatment helps to bring this fascinating correspondence out of the purely scientific domain so that it can be more widely appreciated.

A final hope is that 'light-handed treatment' does not necessarily mean imprecise or inadequate. Throughout this thesis, we attempt to describe the correspondence without solely recoursing to basic slogans like *'proofs are programs'* and expecting such maxims make the subject obvious and apparent. We cannot contest that the slogans that frequently appear in relation to the correspondence are cognitively useful—after all, we use them too—but we prefer to expand a bit more on the theory so that readers can see the mechanics of the correspondence for themselves.

Our planned program then is to begin with a brief intellectual history. The inclusion of history is not strictly necessary, but it situates the correspondence between how we got here and where we are going, which we feel is important. The 'where we are going' is of particular note since it motivates much of the current interest in the isomorphism. This brief intellectual history appears in the next section.

In Chapter 2, we cover the conceptual background needed to establish the correspondence, which includes a presentation of the untyped lambda calculus, the simply-typed lambda calculus, and intuitionistic natural deduction. We try to explain and motivate the technicalities as much as possible.[1] One distinguishing feature of the chapter is that we present our own system of intuitionistic natural deduction. That particular system differs from usual presentations of intuitionistic deduction by incorporating labels for *every* assumption (not just discharged ones) and by identifying different

[1]For better or worse, one still usually needs to use several sources to understand how a kind of formalism works. If we think of the phrase 'lambda calculus' as denoting a kind, and a particular system as a member of a group of systems that share features in common, it makes sense that we need varied exposure to different systems of the same class to generate understanding of the class or kind at large. At the end of this introduction, we list some suggested readings to supplement our later presentation.

kinds of introductions (vacuous against strict). We give reasons why we view these divergences as necessary improvements to regular natural deduction.

Using the background provided in Chapter 2, we are able to illustrate the correspondence in Chapter 3. The question of whether a particular correspondence between two formal systems is also an isomorphism depends on the systems involved. Confusingly, many ‘Curry-Howard isomorphisms’ are not actually exact and are more like deep correspondences than isomorphisms. However, we are able to prove a full isomorphism between intuitionstic natural deduction and the simply-typed lambda calculus, which is a central contribution of this thesis. The isomorphism we prove is largely a result of modifying natural deduction according to the proposals in Chapter 2. We are quite fortunate in that a fortified natural deduction ends up perfectly aligning with the simply-typed lambda calculus.

In our final chapter, Chapter 4, we explore the significance of the Curry-Howard Correspondence. First we turn a critical eye to the popular phrases such as ‘proofs-as-programs’, ‘proofs are programs’, and ‘proofs and programs are two sides of the same coin’. Such language proliferates in the epistemic landscape, and this makes for a good opportunity to revisit the relationship between metaphor (or figurative language generally) and theory. Second, we return to the theme of ‘where we are going’ by exploring programming type-safety as well as the uses of proof assistants. This last, forward-looking excursion concludes the thesis.

## 1.1 A Brief History

Gottlob Frege's publication of his *Begriffsschrift* in 1879 is usually credited as the birth of modern symbolic logic. However, such a story on the origins of modern logic is very simplistic and more mythical than factual. In reality, there were *many* more figures involved in the early days of modern logic, all working in multifarious schools and traditions. For some simplification, we can say that the end of the nineteenth century saw roughly three schools that all contributed to modern logic.[2] There was the algebraic tradition associated with Boole, Peirce, Jevons, Schröder, and Venn, and it focused its efforts on uncovering the relationship between reason and algebraic-calculi with operations like addition or multiplication.[3] In a way, one can view their projects as abstracting from instances of reason to logical principles.

Another school preferred the inverse perspective: they viewed logical principles as underlying all thought, including scientific thought.[4] These were the 'logicists', and they saw the subject of logic as foundational and as concerned with systemizing underpinning principles that all thought shares in common. The 'logicists' were mostly comprised of Russell, early Wittgenstein, and Frege. Finally, there were mathematicians who were concerned about axiomatizing different mathematical systems like arithmetic, geometry, set theory, and analysis.[5] This group included mathematicians such as Peano, Dedekind, Huntington, Pasch, Heyting, Hilbert, Veblen, and Zermelo.[6]

[2] Stewart Shapiro and Peter King. "The History of Logic". In: *The Oxford Companion to Philosophy*. Ed. by Ted Honderich. Oxford University Press, 1995, pp. 496–500, pp. 3–5.
[3] Ibid., p. 4.
[4] Ibid., p. 4.
[5] Ibid., p. 5.
[6] Ibid., p. 5.

Frege's attempt at providing a foundations of arithmetic, though assiduous and interesting, was unsuccessful. It was Russell who informed Frege of the paradox, now known as Russell's paradox, inherent to the system that Frege developed. This naturally led to a scramble of sorts to somehow resolve the paradox or find a way around it.[7] During this episode, Russell worked with Whitehead, and together they developed a new theory, the Ramified Theory of Types, in their *Principia Mathematica*. The theory was highly complex and was developed to similarly defend the view that mathematics, or at least arithmetic, could be grounded on its logical principles.

However, its highly complex nature along with controversial choices of 'logical principles' and the seemingly contrived, artificial character of the resultant arithmetic meant that few others took up Russell and Whitehead's research. Despite the fact that their theory was not immediately popular, type theory has now become extraordinarily useful, especially as it pertains to the construction of programming languages. By assigning programming terms a type, we can better identify programming bugs (sometimes, even before the program executes): all we have to do is check the type at each step to ensure things are running as we expect them. This is starting to become a more widespread feature in programming languages as programming safety becomes an increasing concern. As for our present focus, the Curry-Howard correspondence similarly draws from Russell and Whitehead's theory.

Around the same time that Frege, Russell, and Whitehead were working on their logicist projects, others were also working on their own theories

[7]Frege tried to find a way around the paradox by attempting to restrict Basic Law V, which was thought to be the cause of the paradox. See Edward N. Zalta. *Gottlob Frege*. The Stanford Encyclopedia of Philosophy. Aut. 2020. URL: `https://plato.stanford.edu/archives/fall2020/entries/frege/`, sec. 1, sec. 2.4.1

about the nature of mathematics and mathematical practise. Brouwer, for instance, was developing *intuitionism*, which became a highly influential view not only within the philosophy of mathematics, but also logical practise and—for some time—mathematical practise itself. Because Brouwer viewed the truth of a mathematical statement as dependent on the existence of some kind of corresponding mental construct that *proves* it, a few classically valid logical principles became invalid by Brouwer's lights. In crude summary, *truth is truth* for the classical perspective, but for the intuitionistic perspective, *truth is provability*.

Brouwer's views found some reception among other mathematicians and logicians, which resulted in a logical interpretation of those views called the **BHK** interpretation. The system of logic formed in accordance to this interpretation is now called intuitionistic logic. Intuitionistic logic is important for our purposes because we prove the isomorphism between intuitionistic logic and the simply-typed lambda calculus. In fact, there seems to be a deep connection shared between the two: for example, the **BHK** interpretation says that a proof of $A \to B$ is really just a method of transforming a proof of $A$ into a proof of $B$, but this is exactly what we do in the lambda calculus when creating a function type.[8] Eventually though, the Curry-Howard isomorphism was proved for classical logic too, but this took a bit more time to accomplish.[9] This is in contrast with the seemingly more natural, straightforward connection between intuitionistic logic and the

[8] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism.* 1st ed. Amsterdam: Elsevier, 2006, p. v.

[9] Michel Parigot. "Classical Proofs as Programs". In: *Computational Logic and Proof Theory.* KGC 1993. Ed. by Georg Gottlob, Alexander Leitsch, and Daniele Mundici. Vol. 713. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer-Verlag, 1993, pp. 263–276. DOI: 10.1007/BFb0022575.

simply-typed lambda calculus.

With all these developments on the nature of mathematics and mathematical practise, the question of *what is a proof* was bound to be asked. In striking insight, Hilbert thought that we can "consider the proof itself to be a mathematical object," which marked the start of Hilbert's formal investigation of proof.[10] Over time and with many research developments, this became its own respected branch of mathematical logic called *proof theory*. Hilbert's goal was to provide metamathematical consistency proofs for infinitary mathematics, like geometry and arithmetic, and he worked extensively with Bernays to develop these consistency proofs using proof-theoretical methods.[11] At the time, Hilbert and Bernays used the epsilon calculus—which they developed themselves—and like other available logics, the epsilon calculus was an axiomatic-style logic.[12]

A limitation of the axiomatic method is that it did not intuitively represent mathematical reasoning, nor did mathematicians actually proceed using the method.[13] Łukasiewicz issued an open challenge to logicians to find another method that would better represent actual mathematical reasoning (i.e., from assumptions), but nevertheless prove the same amount of theorems as axiomatic logics.[14] Jaśkowski developed two types of natural

[10] Paolo Mancosu. *The Adventure of Reason: Interplay between Philosophy of Mathematics and Mathematical Logic, 1900-1940*. Oxford: Oxford University Press, 2014, p. 129.

[11] Michael Rathjen and Wilfried Sieg. *Proof Theory*. The Stanford Encyclopedia of Philosophy. Aut. 2020. URL: `https://plato.stanford.edu/archives/fall2020/entries/proof-theory/`, sec. 2.1.

[12] Jeremy Avigad and Richard Zach. *The Epsilon Calculus*. The Stanford Encyclopedia of Philosophy. Aut. 2020. URL: `https://plato.stanford.edu/archives/fall2020/entries/epsilon-calculus/`, sec. 2.

[13]Usually, an axiomatic formalization of a proof would be appear after the fact.

[14] Francis Jeffry Pelletier. "A Brief History of Natural Deduction". In: *History and Philosophy of Logic* 20.1 (1999), pp. 1–31. DOI: `10.1080/014453499298165`, p. 3.

deduction that met Łukasiewicz's challenge.[15,16] In the same year, Gentzen also formulated natural deduction systems where assumptions play a central role, independent of Łukasiewicz and Jaśkowski.[17] These systems of natural deduction were huge advancements in proof theory, and the correspondence depends on the construction of such logical calculi and the advancements that subsequently occurred after their development.

Roughly a decade before the development of natural deduction, Schönfinkel became the first to develop a combinatory logic, which is a formal system that was meant to explore function abstraction, application, and substitution.[18] More specifically, combinatory logic was developed by defining function abstraction with basic combinators (i.e., kinds of primitive operators).[19] The lambda calculus—which was developed later—is another formal system that explores those concepts, but the lambda calculus instead treats function abstraction as primitive.[20] The two logics, of course, are nonetheless very closely related.

Some years later, Curry independently developed his own combinatory logic, and Church developed his lambda calculus around the same time. After some improvements to his original system, Church conjectured that all effectively computable functions are lambda definable. In other words, Church's thesis was that everything computable can be written in terms of

[15] Ibid., p. 3.

[16] Stanisław Jaśkowski. "On the Rules of Suppositions in Formal Logic". In: Storrs McCall. *Polish Logic 1920–1939*. Oxford: Oxford University Press, 1967, pp. 232–258.

[17] Pelletier, op. cit., p. 4.

[18] Felice Cardone and J. Roger Hindley. *History of Lambda-Calculus and Combinatory Logic*. MRRS-05-06. Swansea University Mathematics Department, 2006, pp. ii+1–93. URL: https://www.researchgate.net/publication/228386842_History_of_lambda-calculus_and_combinatory_logic, pp. 1, 3.

[19] Ibid., p. 1.

[20] Ibid., p. 1.

the lambda calculus. With Turing's development of *Turing machines*, this later became the Church-Turing T nhesis. Both Curry and Church eventually developed typed versions of their systems: the introduction of types to combinatory logic and the lambda calculus was the result of wanting to avoid inconsistencies.[21] In the midst of all these exciting developments, we finally arrive at our anticipated, pivotal moment: Curry notices that there is a correspondence between aspects of combinatory logic and propositional logic.[22]

But this is really only one part of it (after all, it is called the Curry-*Howard* correspondence). In fact, Curry's observation didn't immediately stimulate wide interest and deep investigation. Moreover, his observation did not yet constitute the full correspondence we know today. The full correspondence actually came about three decades later when Howard combined Curry's observation with Tait's observation that there is a correspondence between cut elimination and lambda reduction.[23] [24]

Howard's paper is largely responsible for the subsequent interest in the correspondence.[25] Martin-Löf, in particular, directly learned about the correspondence from Howard, and this deeply influenced his own work.[26]

[21] Ibid., pp. 9, 12.

[22] Haskell B. Curry. "Functionality in Combinatory Logic". In: *Proceedings of the National Academy of Sciences* 20.11 (1934), pp. 584–590. DOI: `10.1073/pnas.20.11.584`, p. 584.

[23] William A. Howard. *The Formulae-as-Types Notion of Construction*. Chicago: Department of Mathematics, University of Illinois, 1969, pp. 1–12, p. 2.

[24] Howard's mimeographed notes were later published as William Alvin Howard. "The Formulae-as-Types Notion of Construction". In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Ed. by Jonathan P. Seldin and J. Roger Hindley. London and New York: Academic Press, 1980, pp. 480–490.

[25] Furthermore, Howard's paper is probably responsible for the predominant use of the lambda calculus when establishing the correspondence, rather than Curry's combinatory logic. He writes "It is convenient to use $\lambda$-terms rather than combinators. This corresponds to the sequent formulation of propositional logic." See Howard, loc. cit.

[26] Cardone and Hindley, op. cit., p. 34.

Thereafter, Martin-Löf developed intuitionistic type theory which exploited the correspondence between propositions and types.[27] Martin-Löf's intuitionistic type theory can be thought of as a formalization of intuitionistic, or constructive, mathematics (i.e., it makes clear the character of a constructive mathematics).[28] This intuitionistic type theory ends up being the basis for Nuprl, a proof assistant developed by Constable and his colleagues.[29]

Howard's work also influenced Coquand and Huet—via Girard, rather than Martin-Löf—and they subsequently developed their Calculus of Constructions that later became the basis for Coq, another proof assistant.[30] Yet another early proof assistant that depends on the correspondence is de Bruijn's Automath, but de Bruijn's work was developed independently of Howard.[31] These proof assistants marked incredible progress in the domains of logic, mathematics, and computer science: for the first time, computers could verify formal proofs or help search for a proof (sometimes using human interaction). Proof assistants remain a very hot topic of research and development: new proof assistants like Lean have been created, and popular proof assistants and their library of theorems are constantly updated. Currently, there is a feeling of hopeful excitement in the air: proof assistants may one day represent the future of mathematics.[32,33]

[27] Ibid., p. 33.

[28] Bengt Nordström and Jan Smith. "Propositions and Specifications of Programs in Martin-Löf's Type Theory". In: *BIT* 24.3 (1984), pp. 288–301. DOI: 10.1007/BF02136027, p. 289.

[29] Jonathan P Seldin. *The Logic of Curry and Church*. Department of Mathematics and Statistics, University of Lethbridge, 2008, pp. 1–77. URL: https://people.uleth.ca/~jonathan.seldin/CCL.pdf, p. 8.

[30] Ibid., p. 8.

[31] Philip Wadler. "Propositions as Types". In: *Communications of the ACM* 58.12 (2015), pp. 75–84. DOI: 10.1145/2699407, p. 79.

[32] Herman Geuvers. "Proof Assistants: History, Ideas and Future". In: *Sadhana* 34.1 (2009), pp. 3–25. DOI: 10.1007/s12046-009-0001-5.

[33] Kevin Hartnett. *Building the Mathematical Library of the Future*. Quanta Maga-

To be sure, the journey to this point has been one marked by prolific, exciting periods. But as we saw, even quiet contributions and observations—like in Curry's paper—may be early ripples that later rise into a great wave. In that paper where Curry first notes the correspondence, he begins by writing

> In an attempt to resolve the foundations of logic and mathematics into their elements, it has occurred to several persons that certain notions, ordinarily taken as primitive, could be analyzed into constituents of much simpler nature. Among such notions are, on the one hand, various processes of substitution, and the use of variables generally; and, on the other hand, the categories of logic—such as proposition, propositional function and the like—together with the intuitions by which we tell what entities belong to them.[34]

It's a beginning in a beginning, and so we shall begin from this point too.

## Suggested Readings

Barendregt, Henk and Erik Barendsen. *Introduction to the Lambda Calculus*. 1998, pp. 1–55. URL: `https://www.researchgate.net/publication/215458960_Introduction_to_lambda_calculus`.

Girard, Jean-Yves, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge: Cambridge University Press, 1989.

zine. URL: `https://www.quantamagazine.org/building-the-mathematical-library-of-the-future-20201001/`.

[34] Curry, loc. cit.

Hindley, J. Roger. *Lambda-Calculus and Combinators: An Introduction*. In collab. with Jonathan P. Seldin. Cambridge: Cambridge University Press, 2008.

Rojas, Raul. *A Tutorial Introduction to the Lambda Calculus*. 2015. arXiv: 1503.09060 [cs]. URL: http://arxiv.org/abs/1503.09060.

Selinger, Peter. *Lecture Notes on the Lambda Calculus*. Halifax: Department of Mathematics, Dalhousie University, pp. 1–120. URL: https://arxiv.org/abs/0804.3434.

Sørensen, Morten Heine and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. 1st ed. Amsterdam: Elsevier, 2006.

# Chapter 2

# Conceptual Background for the Correspondence

The Curry-Howard isomorphism presents a significant and fascinating connection between proofs, on one hand, and programs, on the other hand.[1] For this reason, the isomorphism is often referred to as the "proofs-as-programs" paradigm. But how exactly does this connection, or rather, this 'paradigm', obtain?

In this chapter, we will attempt for now to cover the conceptual basics required to understand the isomorphism in full detail. To do this, however, it will be necessary to review the untyped and simply-typed $\lambda$-calculus, which will cover the programs aspect of the paradigm. To then account for the proofs aspect of the paradigm, we will also need to review intuitionistic natural deduction. In the following chapter, we will show how the two come together under the isomorphism. In Chapter 4, we will then explore the

[1]Sometimes, this connection is presented as obtaining between *propositions* and programs.

isomorphism's significance.

## 2.1 The Untyped $\lambda$-Calculus

The $\lambda$-calculus is a formal system and programming language that describes function abstraction and application in general. First sketched out by Alonzo Church in 1932, the $\lambda$-calculus was originally intended to be a new foundation for logic that depended upon the notion of *function* rather than *set*.[2] However, Church's first presentation of the $\lambda$-calculus turned out to be inconsistent, and thus failed in its aspirations to supersede Bertrand Russell and Alfred Whitehead's type theory. Yet, this failure bore little on the $\lambda$-calculus' eventual prospects: in 1936, having distilled his formal system, Church demonstrated how the untyped $\lambda$-calculus circumscribes the scope of all effectively-computable functions. Few enjoy a similar distinction.

As one may later realize, it is wonderfully striking that the $\lambda$-calculus is a model for computation. Such a fact entails not only that computable functions are reducible to $\lambda$-functions but also that the whole scope of computability is definable within the means of a very *small* language. We turn to this very small language now.

### $\lambda$-Calculus Terms

The $\lambda$-calculus contains only three basic kinds of terms. There are variables, abstractions $(\lambda x.M)$ that define functions, and applications $(MN)$ that apply $\lambda$-term $M$ to $\lambda$-term $N$.

[2] Cardone and Hindley, *History of Lambda-Calculus and Combinatory Logic*, p. 6.

**Definition 1** (The Set of $\lambda$-Terms)**.** We first suppose that there is a countably infinite set of variables $V$. Then the set of $\lambda$-terms $\Lambda$ can be defined inductively as follows.[3]

1. If $x \in V$, then $x$ is a $\lambda$-term (i.e., $x \in \Lambda$).
2. If $M, N \in \Lambda$, then $(MN)$ is a $\lambda$-term (i.e., $(MN) \in \Lambda$).
3. If $x \in V$ and $M \in \Lambda$, then $(\lambda x.M)$ is a $\lambda$-term (i.e., $(\lambda x.M) \in \Lambda$).

Some notes about these terms are in order. The variable $x$ in an arbitrary function abstraction $\lambda x.M$ represents the function's argument and is often called the function's 'formal parameter'. Moreover, $x$ may but need not occur in $M$, which is called the 'body' of the function. If $x$ appears as the formal parameter of an abstraction, it can be said to bind the $x$'s in $M$. That is, supposing there were a free occurrence of $x$ in the body $M$, the abstraction $\lambda x.M$ effectively binds that free occurrence in $M$. In comparison to abstraction, function application is much more straightforward. Interestingly, there is no restriction in the untyped calculus on which terms can be used in function application: any lambda term will do. Yet for all their ease, we will soon see that the matter of *reducing* these applications is where the trouble lies.

Having alluded to free and bound variables, we can formally define the set of free variables of $M_0$, FV($M_0$).

**Definition 2** (The Set of Free Variables of a Term $M_0$)**.** The set of free variables of $M_0$, FV($M_0$), can be defined inductively as follows.

1. If $M_0$ is equivalent to $x$, then FV($x$) $= \{x\}$,

[3]The Greek letter $\Lambda$ is the uppercase form of $\lambda$.

2. If $M_0$ is equivalent to $MN$, then $\text{FV}(MN) = \text{FV}(M) \cup \text{FV}(N)$,

3. If $M_0$ is equivalent to $\lambda x.M$, then $\text{FV}(\lambda x.M) = \text{FV}(M) - \{x\}$.

Though the above definitions formally exhaust what can be said about $\lambda$-terms, there are several conventions to be aware of.

**Convention 3.** The following conventions serve to simplify $\lambda$-terms when there is little opportunity for ambiguity.

1. When a $\lambda$-term is equivalent to an application or abstraction, we may omit the outside brackets and obtain $\lambda x.M$ or $MN$.[4]

2. Iterations of applications associate to the left when eliminating brackets. That is, $M_0 M_1 \dots M_n$ abbreviates $(\dots((M_0 M_1) M_2) \dots M_n)$.

3. Relatedly, iterations of abstractions associate to the right when eliminating brackets. That is, $\lambda x_1 \dots x_n.M$ abbreviates $\lambda x_1.(\lambda x_2.(\dots(\lambda x_n.M)\dots))$. Moreover, we may omit repeated lambdas and write $\lambda x y z.M$ in place of $\lambda x.(\lambda y.(\lambda z.M))$.

4. When eliminating brackets for a single abstraction, the abstraction is given the widest possible scope. That is, $\lambda x.MNL$ abbreviates $(\lambda x.((MN)L))$.

## Reduction

To complete the formal presentation of the untyped $\lambda$-calculus, there is one remaining piece to cover. That is, once we have a $\lambda$-term like $(\lambda x y.y)(MN)$,

[4] Recall from Page 14 that terms of the form $(\lambda x.M)$ are abstractions and terms of the form $(MN)$ are applications.

can this term be simplified? In other words, is there a formal procedure that allows us to *evaluate* or *reduce* this term to a point where it cannot be reduced or simplified further?

In the $\lambda$-calculus, the answer is *yes*. To evaluate $\lambda$-terms so as to reduce them to their simplest form, we use $\beta$-reduction, which is the principal transformation rule of the calculus. The rule depends on the notion of substitution, which we shall review first.

**Definition 4** (Substitution)**.** We represent the substitution of $\lambda$-term $N$ for $x$ in $\lambda$-term $M$ by $M[N/x]$. In other words, $M[N/x]$ formally represents the informal, intuitive process of replacing every free instance of $x$ in $M$ with the term $N$.

It is important to note that substitution is defined *if and only if* no free instance of $x$ in $M$ occurs in some $(\lambda y.L)$ where $y$ happens to be a free variable of $N$. Without this stipulation, substitution would permit the placement of a free $y$ into the body of a function that has $y$ as a formal parameter and thereby confuse free and bound variables, which is undesirable.

Having defined substitution, we can now define $\beta$-reduction. As we will see, it is confoundingly simple despite the fact that the rule is the only essential one to the calculus.

**Definition 5** ($\beta$-Reduction)**.** Let $M$, $N$ be arbitrary $\lambda$-terms and $x$ an arbitrary variable. Then we define $\beta$-reduction with the following schema.

$$(\lambda x.M)N \to_\beta M[N/x]$$

So $\beta$-reduction merely relates *redex* $(\lambda x.M)N$ to *contractum* $M[N/x]$.

Related to $\beta$-reduction is a notion called $\beta$-conversion. Note that the reduction of $\lambda x.M)N$ to $M[N/x]$ can occur anywhere in a given term that contains multiple abstractions as subterms. So in contrast to the single step in $\beta$-reduction, $\beta$-conversions can consist of any number of $\beta$-reductions to the subterms of a given $\lambda$ term.

**Definition 6** ($\beta$-Conversion)**.** A $\lambda$-term $M$ $\beta$-converts to a $\lambda$-term $N$ if there is a sequence of $\beta$-reduction steps between $M$ and $N$.

We can $\beta$-convert a term until it is in normal form, which is defined below.

**Definition 7** ($\beta$-Normal Form)**.** A term is in $\beta$-normal form when there are no longer occurences of $(\lambda x.M)N$ as a subterm (i.e., the $\lambda$ term cannot be simplified further). Usually, the $\beta$-normal form of a term is obtained by using $\beta$-conversion.

In fact, it is here that we may begin to see how $\lambda$ terms are programs. In the untyped $\lambda$-calculus, there are normal terms that represent numbers (e.g., Church numerals). When we apply a $\lambda$ terms to one such 'number', the process of $\beta$-reducing that application is akin to *executing a program*, whose result is another normal term representing a number. In other words, a term of the form $(\lambda x.M)$ is a program that takes an input $N$. The execution of this program with the input is akin to reducing the term $(\lambda x.M)N$, and the resultant normal form term is the output. We can also represent numbers using $\lambda$ terms (and other special terms and rules not covered here) in the simply-typed $\lambda$-calculus, which is a calculus that we will discuss in the next section.

Though this officially completes our presentation of the untyped $\lambda$-calculus, for the sake of total clarity, it is worth mentioning $\alpha$-equivalence. As might have been suggested, we should strive always to separate our free variables from our bound ones. For this reason, it is useful to assume that our terms' bound variables are always different from their free variables. Notice that in certain cases, we can indicate their difference and maintain the distinction only by *renaming bound variables*, which is precisely what $\alpha$-equivalence allows us to do.

**Definition 8** ($\alpha$-Equivalence)**.** Let $x$ and $y$ be arbitrary variables and $M_x$, $M_y$ be two arbitrary lambda terms containing $x$ and $y$ respectively. Suppose that $M_x$ and $M_y$ are the same saving for the fact that $x$ appears in $M_x$ where $y$ would appear in $M_y$, and vice versa. Then two $\lambda$-expressions are $\alpha$-equivalent if and only if

The result of the process of replacing every $x$ with $y$ in $\lambda x.M_x$ (i.e., $\lambda y.M_x[y/x]$) is equivalent to $\lambda y.M_y$ where $y \notin M_x$.

In other words, $\lambda x.M_x \equiv_\alpha \lambda y.M_y$ (i.e., are $\alpha$-equivalent), and thus we identify expressions that differ only in the names of their bound variables. As such, $\lambda x.x \equiv_\alpha \lambda y.y$, and we can have the following.

$$(\lambda x.x)a \equiv_\alpha (\lambda y.y)a$$

And the two alpha-equivalent terms $\beta$-reduce to the same term, as follows.

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

$$(\lambda y.y)a \rightarrow_\beta a$$

Since $\beta$-reduction is a very important rule, it is no surprise that a very important property of the calculus involves it. We end our presentation of the untyped calculus here and shall proceed to our discussion of the *Church-Rosser property.*

## Normalization

One important question to ask is whether every $\lambda$-term has a $\beta$-normal term, or in other words, a term which cannot be further $\beta$-reduced. In tandem, we can also ask whether these $\beta$-normal terms will be unique. In answering this question, it turns out that this is one major site where the untyped and simply-typed $\lambda$-calculus will diverge: the untyped calculus is weakly normalizing since not every term can be $\beta$-reduced to a (unique) $\beta$-normal term, whereas the simply-typed calculus is strongly normalizing in this exact sense. This is just one way in which the two calculi differ, but we will review the simply-typed calculus later.

For now, it is important to recognize the significance of having a *unique* $\beta$-normal term, which is what the Church-Rosser property is about.

**Theorem 9** (Church-Rosser Property)**.** *Let $M$, $N_1$, and $N_2$ be arbitrary $\lambda$-terms, and suppose that $M$ is $\beta$-reducible to $N_1$ and $N_2$. That is, suppose*

$$M \twoheadrightarrow_\beta N_1 \text{ and } M \twoheadrightarrow_\beta N_2.$$

*Then there is a $L \in \Lambda$ such that*

$$N_1 \twoheadrightarrow_\beta L \text{ and } N_2 \twoheadrightarrow_\beta L.$$

**Corollary 10.** If $M \in \Lambda$ is reducible to a $\beta$-normal term, its $\beta$-normal term is *unique*. That is, every $M$ has at most one $\beta$-normal form.

It may seem unimpressive that the Church-Rosser Property implies that every reducible $M$ has a single $\beta$-normal $L$, however it implies that how we reduce our terms does not matter: eventually, we will reach the same $\beta$-normal term.[5] An example will best illustrate the property in action.

Consider the following $\lambda$-term.

$$(\lambda x.(\lambda y.y\,x)z)v$$

Notice that we may reduce this term in different ways. If we begin the reduction using the innermost application, we get

$$(\lambda x.z\,x)v.$$

However, if we begin the reduction on the outermost application, we instead get

$$(\lambda y.y\,v)z.$$

Now despite this initial difference, notice that another round of reduction results in the same term:

$$(\lambda x.z\,x)v \rightarrow z\,v$$
$$(\lambda y.y\,v)z \rightarrow z\,v.$$

As one might imagine, this property is very desirable from a programming perspective. Were it otherwise, the order of reduction would significantly alter final output, and we would have needed to define new rules and conventions to appropriately deal with such added complexity. Moreover, for a program, we would want there to be a unique output for every input.

[5]In computer science, this property is often known as *confluence*.

But as we saw, the Church-Rosser property guarantees this, which is another reason why we can view the $\lambda$-calculus as a kind of programming language.

It's worth reiterating, however, that in the untyped calculus a term may not have a normal form even if the term is reducible. For instance, an oft-cited example is the term $(\lambda x.xx)(\lambda x.xx)$:

$$\begin{aligned}(\lambda x.xx)(\lambda x.xx) &\to_\beta (\lambda x.xx)(\lambda x.xx)\\ &\to_\beta (\lambda x.xx)(\lambda x.xx)\\ &\to_\beta (\lambda x.xx)(\lambda x.xx)\\ &\dots\end{aligned}$$

The simply-typed $\lambda$-calculus does not suffer from the same limitation: every term has its $\beta$-normal term. As such, unlike the untyped $\lambda$-calculus, there are no reducible terms that remain reducible forever. This stems, in part, from the fact that $(\lambda x.xx)(\lambda x.xx)$ becomes an illegal term in the simply-typed calculus.

## 2.2 The Simply-Typed $\lambda$-Calculus

The essential difference between the untyped and simply-typed $\lambda$-calculi is that whereas the untyped calculus allows self-application, the simply-typed calculus does not allow a term to be applied to itself.[6] This is because *every term now has a type*. A *type* can be thought of as a sort of syntactic object since we don't have to consider the contents or elements contained in it. In some sense, one can think of a term's type as sorting that term into a

[6]The simply-typed $\lambda$-calculus is often referred to as the $\lambda^{\to}$ calculus.

particular *kind*. For example, to assign the type $\mathsf{Nat}$ to the number 2 is to say that 2 belongs to the kind 'natural number'.

This may appear unnecessarily pedantic, but there is an intuitive motivation to assign specific types to terms: the appropriate type of terms are always applied to functions of matching type arguments. That is to say, types guarantee that a function expecting an integer always receives an integer as input. In contrast, as we saw earlier with the term $(\lambda x.xx)(\lambda x.xx)$, the untyped calculus allows one to apply a function to a function *of the same type*, though this differs from many mathematical functions and can have unintended consequences.[7]

Another difference between the two calculi is that in a simply-typed context, the Church-Rosser Property becomes a kind of consistency result: since every well-formed term has only one $\beta$-normal form in the simply-typed $\lambda$-calculus, we know that for every simply-typed term $M$, if $M$ has the $\beta$-normal form of $L$, then $Q$ is not the $\beta$-normal form of $M$ where $L \not\equiv Q$.

We will now proceed to the simply-typed $\lambda$-calculus.[8] It will be similar to the untyped calculus and will re-employ many of the same notions, with the

[7]In other words, one cannot apply a function of type $A \to A$ to a function also of type $A \to A$ in the simply-typed $\lambda$-calculus. Instead, one can apply a function of type $(A \to B) \to C$ to a function of type $A \to B$. Notice, however, that we can apply one to the other in the simply-typed $\lambda$-calculus because *they are different types*.

[8]Our presentation will be a Curry-style simply-typed $\lambda$-calculus, which is *implicit* in its typing. But it's worth noting that there are also Church-style simply-typed $\lambda$-calculi, which are *explicit* in their typing. Explicit mention of types always makes a term's type decidable. In contrast, there are some Curry systems where the type of a term is undecidable. It has been noted that there is some confusion where *explicit* has sometimes been understood as *explicit in syntax*. See Benjamin C. Pierce. *Types and Programming Languages*. Cambridge, Mass: MIT Press, 2002, p. 111. However, what is meant by *explicit* is explicitness of the type of a term *semantically*. That is, in Church-style calculi, the typing is prior to semantics: the type of a term forms a part of that term's meaning. However, in Curry-style calculi, semantics is prior to typing: thus, terms are well-defined first before a typing system is then superimposed. As we will see, the final superimposition of a typing system in a Curry-style calculi ensures that terms behave as we intend.

exception of the added types. Of particular note, the simply-typed calculus has typing rules, which will play an extremely important role in establishing the isomorphism later on.

## Substitution, $\beta$-Reduction, and $\alpha$-Equivalence

It is not worth precisely rehashing out the notions of substitution, free and bound variables, and $\alpha$-equivalence since these will look almost identical to their untyped $\lambda$-calculus counterparts. But it is worth mentioning that we will reuse these notions, albeit with slight changes to account for the introduction of types.

In contrast, $\beta$-reduction requires some careful review, which we will do later. Until then, the following definition serves our current purposes.

**Definition 11** (Simply-Typed $\beta$-Reduction)**.** Let $M$, $N$ be arbitrary simply-typed $\lambda$-terms and $x$ an arbitrary variable of type $A$. Then we define $\beta$-reduction with the following schema.

$$(\lambda x^A.M)N \to_\beta M[N/x^A]$$

In the above definition, we use a superscript on the variable $x$ to indicate that it is of type $A$. The above definition also invokes the notion of simply-typed $\lambda$-terms, so we should now focus on making this notion clear.

## Types and Simply-Typed $\lambda$-Terms

**Definition 12** (The Set of Simple Types)**.** Suppose there is a set of basic types $\tau$, which may include the type of booleans, natural numbers, integers, etc. Then the set of simple types, $\mathbb{T}$, may be inductively defined as follows.

1. $A \in \tau \Rightarrow A \in \mathbb{T}$,

2. $A, B \in \mathbb{T} \Rightarrow A \rightarrow B \in \mathbb{T}$,

3. $A, B \in \mathbb{T} \Rightarrow A \times B \in \mathbb{T}$,

4. $A, B \in \mathbb{T} \Rightarrow A + B \in \mathbb{T}$, and

5. $1 \in \mathbb{T}$,

6. $0 \in \mathbb{T}$.

To be clear, the type $A \rightarrow B$ is meant to represent the type of functions that map from $A$ to $B$. Moreover, $A \times B$ is meant to be taken as the type of pairs $\langle x, y \rangle$ with $x$ of type $A$ and $y$ of type $B$, whereas $A + B$ is meant to be the sum type. Finally, 1 is the unit type that has one element and 0 the empty type that contains no elements.[9]

Now that we've defined our types, we may proceed to defining our simply-typed $\lambda$-terms. This will require defining *raw simply-typed $\lambda$-terms* and *typing rules.*[10]

**Definition 13** (The Set of Raw Simply-Typed $\lambda$-Terms)**.** Suppose there is a set of typed variables, $V_{\mathbb{T}}$, which contains, as an example, $x^A$ (i.e., variable $x$ of type $A$). The set of raw simply-typed $\lambda$-terms, $\lambda_{\mathbb{T}}$, may then be inductively defined as follows.[11]

[9]This singular element is $*$.

[10]More precisely, what we present here is an extended version of the simply-typed $\lambda$-calculus. To construct a 'minimal' simply-typed calculus, we need only 1, 2, and 3. Later on, we will see that these raw terms correspond to the typing rules *var*, *app*, and *abs*, which also constitute the 'minimal' calculus. Further, we would need only the types that correspond to those three typing rules. See Sørensen and Urzyczyn, *Lectures on the Curry-Howard Isomorphism*, pp. 57, 88.

[11]In our definition of $\lambda^{\rightarrow}$ terms, we use abort and case. Others use $\varepsilon$ and $\delta$ respectively. See Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science 7. Cambridge: Cambridge University Press, 1989.

1. $x^A \in V_{\mathbb{T}} \Rightarrow x^A \in \lambda_{\mathbb{T}}$,
2. $M, N \in \lambda_{\mathbb{T}} \Rightarrow (MN) \in \lambda_{\mathbb{T}}$,
3. $M, x^A \in \lambda_{\mathbb{T}} \Rightarrow (\lambda x^A.M) \in \lambda_{\mathbb{T}}$,
4. $M, N \in \lambda_{\mathbb{T}} \Rightarrow \langle M, N \rangle \in \lambda_{\mathbb{T}}$,
5. $M \in \lambda_{\mathbb{T}} \Rightarrow \pi_1 M \in \lambda_{\mathbb{T}}$,
6. $M \in \lambda_{\mathbb{T}} \Rightarrow \pi_2 M \in \lambda_{\mathbb{T}}$,
7. $M \in \lambda_{\mathbb{T}} \Rightarrow \text{ inj}_1 M \in \lambda_{\mathbb{T}}$,
8. $M \in \lambda_{\mathbb{T}} \Rightarrow \text{ inj}_2 M \in \lambda_{\mathbb{T}}$,
9. $M, N, O \in \lambda_{\mathbb{T}} \Rightarrow \text{ case } M \text{ of } (x)N; (y)O \in \lambda_{\mathbb{T}}$, and
10. $* \in \lambda_{\mathbb{T}}$,
11. $M \in \lambda_{\mathbb{T}} \Rightarrow \text{ abort}_A M \in \lambda_{\mathbb{T}}$.

Although this set of terms looks a little like our definition of the set $\Lambda$ from before, there are some major differences. Notice first that our $\lambda$-abstraction now requires a *typed* formal parameter. That is, where before we simply wrote $(\lambda x.M)$, we must now write $x$'s type in a superscript as in $(\lambda x^A.M)$.[12] The next major change is that we now have extra clauses that deal specifically with *pairs*. We've defined $\langle M, N \rangle$ as a pair term, and we've added projections $\pi_i M$ that are intended to extract a single element from any given pair in $\Lambda_{\mathbb{T}}$. That is, if we have a pair $\langle N_1, N_2 \rangle$, then $\pi_i \langle N_1, N_2 \rangle = N_i$.

[12]Later on, however, we drop the superscript because context usually makes the type of the variable sufficiently clear. We only note it now because it should be understood that all terms in $\lambda^{\rightarrow}$ are assigned types: of course, bound variables of abstraction are no exception.

We also now have (1) the left and right injections into a sum type, which are labelled as $\mathrm{inj}_1$ and $\mathrm{inj}_2$ respectively, (2) the case rule, (3) an $*$, which is the single element of type 1, and (4) the abort rule. The injections into a sum type essentially work to create a sum type $A+B$ from the term $A$, in the case of $\mathrm{inj}_1$, or from the term $B$, in the case of $\mathrm{inj}_2$. A sum type can be thought of as a disjoint union with 'tags'. Essentially, the tags allow us to track the type of each term in the disjoint union. The case rule will end up being the elimination rule for these sum types.[13] Finally, because deriving a void type can be thought of as reaching an error, the abort rule allows us to create a term of any type from a void type term. [14]

However, the raw simply-typed $\lambda$-terms do not yet suffice to define proper, well-formed simply-typed terms. So far, we are able to form the raw terms $\pi_2 x^A$, $\pi_1(\lambda x^A.M)$, and $(MN)(MN)$, but if we stop here, they would be ill-formed terms that should be illegitimate in our calculus.[15] To rectify this situation, we must now introduce *typing rules* that will appropriately structure how we use each term.

**Definition 14** (The Typing Rules)**.** We present the typing rules in the following collection of sequents. Note the labels positioned on both left-hand and right-hand sides of the sequents.

[13]The case rule can be thought of as a follows: *if* we can make a program $\Gamma$ that (1) ends in a sum type term $M$, and (2) executing $\Gamma$ with a term of type $A$ ends in a term of type $C$, and (3) executing $\Gamma$ with a term of type $B$ also ends in a term of type $C$, *then* executing $\Gamma$ alone can result in a program that ends with a term (case $M$ of $(x)N$; $(y)O$) whose type is $C$.

[14]However, we must note beforehand the type of the term we create from the abort rule using a subscript (e.g., we write $\mathrm{abort}_A$ and not simply abort).

[15]To be clear, these would be ill-formed terms for the reason that they are *intuitively nonsensical.* Projections should apply only to pairs, and we want it so that applications always result in a different term. For instance, consider that in the untyped $\lambda$-calculus, $(MN)(MN) \Rightarrow (MN)(MN)$. In comparison, $\lambda x.xM \Rightarrow M$.

First, we have the var rule for any $x^A$.

$$\text{var}\ \frac{}{\Gamma, x : A \vdash x : A}$$

The remaining rules are as follows.

$$\text{app}\ \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \to \text{E}$$

$$\text{abs}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \to \text{I}$$

$$\text{pair}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B} \times \text{I}$$

$$\pi_1\ \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_1 M : A} \times \text{E}_1$$

$$\pi_2\ \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_2 M : B} \times \text{E}_2$$

$$\text{inj}_1\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \text{inj}_1 M : A + B} + \text{I}_1$$

$$\text{inj}_2\ \frac{\Gamma \vdash M : B}{\Gamma \vdash \text{inj}_2 M : A + B} + \text{I}_2$$

$$\text{case}\ \frac{\Gamma \vdash M : A + B \qquad \Gamma, x : A \vdash N : C \qquad \Gamma, y : B \vdash O : C}{\Gamma \vdash (\text{case}\ M\ \text{of}\ (x)N; (y)O) : C} + \text{E}$$

$$*\ \frac{}{\Gamma \vdash * : 1}\ 1\ \text{I}$$

$$\text{abort}\ \frac{\Gamma \vdash M : 0}{\Gamma \vdash \text{abort}_A\ M : A}\ 0\ \text{E}$$

Each of these typing rules invoke *typing judgments*, whose basic form is the following:

$$x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \vdash M : A$$

.

We will deconstruct the meaning of the above expression in parts. Notice that each sequent of the form $\Gamma \vdash M$ is composed of two parts, the left-hand and right-hand sides, that are visually separated with a turnstile, $\vdash$. Colons are used to explicitly mention the type of each term: for example, $M : A$ indicates that $M$ has type $A$. The left-hand side of each sequent, indicated in the above typing rules by an arbitrary $\Gamma$, details each typing judgment's sequence of assumptions. So in the above expression, $x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n$ can be read as 'assuming that $x_i$ has type $A_i$ (for $i = 1, \ldots, n$) ...'

This left-hand sequence of assumptions is called a *typing context*. The typing context of a typing judgment is very important because it tells us what the free variables of the right-hand side terms are. To use the above expression again, $x_1 : A_1, x_2 : A_2, \ldots, x_n : A_n \vdash M : A$ means that 'assuming $x_i$ is of type $A_i$ (for $i = 1, \ldots, n$), *M is a term whose only possible free variables are* $x_1, \ldots x_n$, and thus, M is well-typed term whose type is A'. In a sense, knowing what the types of the free variables of $M$ are helps us to determine the type of $M$ itself.

To return to the typing rules alone, one might observe that each rule corresponds to exactly one kind of term from our $\lambda_{\mathbb{T}}$ set. This ensures that each step in a derivation is *uniquely determined*. Finally, as indicated above, our typing rules can be construed as a set of introduction and elimination rules, with the exception of $*$ and abort as well as var, which is treated as an axiom or tautology. As we will see, this interpretation of the rules into pairs of introductions and eliminations will be one part of the bridge that connects the simply-typed $\lambda$-calculus to **NJ** natural deduction.

Through Definitions 12, 13 and 14, we developed various simply-typed $\lambda$-terms and how they interact together. However, notice that we still do not technically know what happens when we project into a pair.[16] We also do not know the result of a case acting on an injection. When we defined $\beta$-reduction in Definition 11, we did not have these other terms yet. Thus Definition 11 is an incomplete definition for $\beta$-reduction in the simply-typed $\lambda$-calculus. Now that we have all the resources needed available to us, we continue where we left off and fully define $\beta$-reduction.

**Definition 15** (Simply-Typed $\beta$-Reduction)**.** Let $M$, $N$ be arbitrary simply-typed $\lambda$-terms that are well-typed (i.e., are not merely raw terms) and $x$ an arbitrary variable of type $A$. Then we define $\beta$-reduction with the following schemas.

$$
\begin{gathered}
(\lambda x.M)N \to_\beta M[N/x] \\
\pi_1\langle M, N\rangle \to_\beta M \\
\pi_2\langle M, N\rangle \to_\beta N \\
(\text{case inj}_1 M \text{ of } (x)N; (y)O) \to_\beta N[M/x] \\
(\text{case inj}_2 M \text{ of } (x)N; (y)O) \to_\beta O[M/y]
\end{gathered}
$$

In this definition of $\beta$-reduction, we present schemas of evaluation for terms immediately related to each other: projections work on pairs and cases work on injections. But what happens when we want to evaluate

[16]Of course, the typing rules give the intuition that we should have the first object of the pair as the result. However, without indicating this in our definition of $\beta$-reduction, that intuition is not yet properly formalized.

terms that are not immediately related but, with some careful evaluation of subformulas, could be? For instance, suppose we had the following program.

$$
\dfrac{
  \dfrac{
    \Gamma \vdash M : A+B
    \qquad
    \dfrac{
      \begin{array}{c}\Gamma, x:A \vdash x:A\\ \vdots \\ \Gamma, x:A \vdash N:C\end{array}
      \qquad
      \begin{array}{c}\Gamma, x:A \vdash x:A\\ \vdots \\ \Gamma, x:A \vdash O:D\end{array}
    }{\Gamma, x:A \vdash \langle N,O\rangle : C\times D}\times\mathrm{I}
    \qquad
    \begin{array}{c}\Gamma, y:B \vdash y:B\\ \vdots \\ \Gamma, y:B \vdash L:C\times D\end{array}
  }{\Gamma \vdash (\text{case } M \text{ of } (x)\langle N,O\rangle;(y)L) : C\times D}+\mathrm{E}
}{\Gamma \vdash \pi_1(\text{case } M \text{ of } (x)\langle N,O\rangle;(y)L) : C}\times\mathrm{E}_1
$$

On the final line, we have the $\lambda^{\rightarrow}$ term $\pi_1(\text{case } M \text{ of } (x)\langle N,O\rangle;(y)L)$. But to what exactly does that reduce? Our definition of $\beta$-reduction only defines an evaluation for projections when they work on pairs, not cases. If we look closely enough, there are pairs hiding within the case term: the pair $\langle N,O\rangle$ whose type is $C\times D$ and the term $L$ whose type is also $C\times D$. Recall from Definition 14 that so long as our arbitrary term $M_0$ is of type $A\times B$, we can use $\pi_1$ to get the term $\pi_1 M_0$ of type $A$ (i.e., use the $\times\mathrm{E}_1$ rule). So although projections do not routinely work with cases, it seems that it should work with a case containing pairs. Ideally then, we would like to have the following.

$$\pi_1(\text{case } M \text{ of } (x)\langle N,O\rangle;(y)L) \rightarrow_\beta (\text{case } M \text{ of } (x)\pi_1\langle N,O\rangle;(y)\pi_1 L)$$

That is, we would like to permute the order of the terms so that $\pi_1$ appears inside case rather than outside of it. We can do this by defining *permutation conversions.*[17]

[17] Girard, Taylor, and Lafont, op. cit., pp. 79–80.

**Definition 16** (Permutation Conversions for Sum Types in $\lambda^{\rightarrow}$)**.** Let $M$, $N$, $O$, and $P$ be arbitrary simply-typed $\lambda$-terms that are well-typed and $x$ and $y$ be arbitrary variables. Finally let $A$ and $B$ be arbitrary types.

1. When $N : A \times B$ and $O : A \times B$

$$\pi_1(\text{case } M \text{ of } (x)N; (y)O) \rightarrow_\beta (\text{case } M \text{ of } (x)\pi_1 N; (y)\pi_1 O)$$

2. When $N : A \times B$ and $O : A \times B$

$$\pi_2(\text{case } M \text{ of } (x)N; (y)O) \rightarrow_\beta (\text{case } M \text{ of } (x)\pi_2 N; (y)\pi_2 O)$$

3. When $N : A \rightarrow B$, $O : A \rightarrow B$, and $P : A$

$$(\text{case } M \text{ of } (x)N; (y)O)P \rightarrow_\beta (\text{case } M \text{ of } (x)NP; (y)OP)$$

4. When $N : 0$ and $O : 0$

$$\text{abort}_B(\text{case } M \text{ of } (x)N; (y)O) \rightarrow_\beta$$
$$(\text{case } M \text{ of } (x)\text{abort}_\text{B} N; (y)\text{abort}_\text{B} O)$$

5. When $N : A + B$, $O : A + B$, $N' : A' + B'$, and $O' : A' + B'$

$$(\text{case } (\text{case } M \text{ of } (x)N; (y)M) \text{ of } (x')N'; (y')O') \rightarrow_\beta$$
$$(\text{case } M \text{ of } (x)(\text{case } N \text{ of } (x')N'; (y')O'); (y)(\text{case } O \text{ of } (x')N'; (y')O'))$$

We also have permutation conversions for empty types.

**Definition 17** (Permutation Conversions for Empty Types in $\lambda^{\rightarrow}$)**.** Let $M$, $N$, and $O$ be arbitrary simply-typed $\lambda$-terms that are well-typed and $x$ and $y$ be arbitrary variables. Finally let $A$ and $B$ be arbitrary types.

1. $\pi_1(\text{abort}_{A\times B}M) \rightarrow_\beta \text{abort}_A M$

2. $\pi_2(\text{abort}_{A\times B}M) \rightarrow_\beta \text{abort}_B M$

3. $(\text{abort}_{A\rightarrow B}M)N \rightarrow_\beta \text{abort}_B M$

4. $\text{abort}_A(\text{abort}_0 M) \rightarrow_\beta \text{abort}_A M$

5. $(\text{case}\ (\text{abort}_{A+B}M)\ \text{of}\ (x)N;(y)O) \rightarrow_\beta \text{abort}_A M$

These permutation conversions serve to make our programs simpler as we are able to evaluate terms more quickly. For instance, permutation conversions allow us to convert from our example program

$$\dfrac{\dfrac{\Gamma \vdash M : A+B \qquad \dfrac{\begin{matrix}\Gamma, x:A \vdash x:A \\ \vdots \\ \Gamma, x:A \vdash N:C\end{matrix} \qquad \begin{matrix}\Gamma, x:A \vdash x:A \\ \vdots \\ \Gamma, x:A \vdash O:D\end{matrix}}{\Gamma, x:A \vdash \langle N,O\rangle : C\times D}\times\mathrm{I} \qquad \begin{matrix}\Gamma, y:B \vdash y:B \\ \vdots \\ \Gamma, y:B \vdash L:C\times D\end{matrix}}{\Gamma \vdash (\text{case}\ M\ \text{of}\ (x)\langle N,O\rangle;(y)L) : C\times D}+\mathrm{E}}{\Gamma \vdash \pi_1(\text{case}\ M\ \text{of}\ (x)\langle N,O\rangle;(y)L) : C}\times\mathrm{E}_1$$

to the following one.

$$\dfrac{\Gamma \vdash M : A+B \qquad \dfrac{\dfrac{\begin{matrix}\Gamma, x:A \vdash x:A \\ \vdots \\ \Gamma, x:A \vdash N:C\end{matrix} \qquad \begin{matrix}\Gamma, x:A \vdash x:A \\ \vdots \\ \Gamma, x:A \vdash O:D\end{matrix}}{\Gamma, x:A \vdash \langle N,O\rangle : C\times D}\times\mathrm{I}}{\Gamma, x:A \vdash \pi_1\langle N,O\rangle : C}\times\mathrm{E}_1 \qquad \dfrac{\begin{matrix}\Gamma, y:B \vdash y:B \\ \vdots \\ \Gamma, y:B \vdash L:C\times D\end{matrix}}{\Gamma, y:B \vdash \pi_1 L : C}\times\mathrm{E}_1}{\Gamma \vdash (\text{case}\ M\ \text{of}\ (x)\pi_1\langle N,O\rangle;(y)\pi_1 L)}+\mathrm{E}$$

If one compares the two programs, one will notice that the difference is simply that the order of +E and $\times\mathrm{E}_1$ are reversed. In the first program, it is

$+$E then $\times$E$_1$, whereas in the second program, it is $\times$E$_1$ then $+$E. Perhaps this does not seem significant, and it would be fair to judge the difference as too subtle. Despite what appears to be a vanishingly small change, we are better off with our second program than with our first. This is because we cannot directly simplify the first one, but in comparison, we are in the position to directly simplify the second using a few techniques. Since we haven't discussed those techniques yet, we are forced to stop here. Even though we do not explicitly cover them presently, the general idea is to permute eliminations upwards in a program. When those eliminations finally appear directly beneath introductions of the same operator, we can simplify the program using substitution methods.[18]

## 2.3 Intuitionistic Natural Deduction

Intuitionistic Natural Deduction, or **NJ** for short, is a derivational (i.e., deductional) calculi that uses intuitionistic logic as its underlying framework.[19] That is, with **NJ** one can construct proof trees that contain only intuitionistically-valid derivations. As is implied, the fundamental motivational force of **NJ** stems from its *derivational* capacity and efficacy. Prior to its invention in 1934–35 by Gerhard Gentzen and Stanisław Jaśkowski independently, for-

[18]In the next section, we will define detours, cut segments, and proof substitution for **NJ**. These notions and processes defined for **NJ** have their application in $\lambda^{\to}$ too and can be repurposed to execute (i.e., simplify) programs.

[19]Intuitionistic logic can be thought of as a subsystem of classical logic. It accepts all the axioms of classical logic *except for the excluded middle*, $A \vee \neg A$. This means only that $A \vee \neg A$ is not a tautology, which is not tantamount to saying that every particular instantiation of $A \vee \neg A$ is invalid in intuitionistic logic. Nevertheless, the result of rejecting the excluded middle as an axiom is that proof by contradiction ceases to be a valid proof pattern in intuitionistic logic. The principle of explosion (i.e., $\bot \to A$), however, remains valid.

mal proofs in logic and mathematics were done in an axiomatic style.[20] In the axiomatic-style of deduction, derivations are ordinarily constructed by substitutions of logical formulas into axiom schemas and applications of modus ponens. In fact, modus ponens is usually the only 'derivational rule' that axiomatic-style systems allow. In contrast, natural deduction, and its sibling system, the sequent calculus, invert this emphasis: they have only one axiom and many derivational rules.

Such an inversion turns out to be highly fruitful and significant. It was never easy—and in fact, it was usually extremely difficult— to construct proofs using the axiomatic-style, and a part of the reason why so is that ordinary, informal mathematical and logical reasoning rarely proceeds from axioms. Prior to the invention of natural deduction and the sequent calculus, Jan Łukasiewicz also noted that ordinary mathematical proofs rarely relied on axiomatic theory and, instead, normally proceeded and reasoned *from assumptions.*[21] But natural deduction and the sequent calculus make use of that observation and do precisely that: they track our initial assumptions by placing them either at the top or on the left-hand side of a proof-tree, and in the rest of the tree, they map out our *paths of reasoning* that follow from the application of derivational rules.[22] Thus, the notion of a derivation becomes essential as it is the process by which we proceed and reason from a set of a assumptions.[23]

For a concrete example of how this might look, consider the following

[20] Pelletier, "A Brief History of Natural Deduction", pp. 3–4.
[21] Ibid., p. 3.
[22]Derivational rules are sometimes also called *inference rules.*
[23]Interestingly, a derivation can also be viewed as the resultant proof (i.e., the output of the process of deriving). Here, a parallel can be drawn to any $\lambda$-term since every $\lambda$-term can be read as both an algorithm and the resultant output of that algorithm.

conditional proof.

$$\cfrac{\begin{matrix}A\\ \vdots\\ B\end{matrix}}{A \to B}\,{\to}\mathrm{I}$$

Here, it's clear that we have created a proof of $A \to B$ because we were able to take $A$ as an assumption and prove $B$ from it. In other words, our written derivation clearly illustrates our path of reasoning: *from A we may derive B and hence, $A \to B$.*

Now this is not the only way to derive $A \to B$. An alternate way of deriving $A \to B$ is to introduce the conditional vacuously:

$$\cfrac{\begin{matrix}\vdots\\ B\end{matrix}}{A \to B}\,{\to}\mathrm{I}$$

And since a conditional can be introduced from assumptions, including copies of assumptions, as well as no assumptions, we can derive the following.

$$\cfrac{\cfrac{A}{A \to A}\,{\to}\mathrm{I}}{A \to (A \to A)}\,{\to}\mathrm{I}$$

But such a derivation conceals a puzzle. Though we have successfully derived $A \to (A \to A)$, by looking at the derivation alone, we cannot exactly tell how each conditional was introduced. That is, was $A \to A$ introduced vacuously or was it derived from the beginning assumption $A$? It is for this reason that the conditional rule in **NJ** is normally presented as follows.

$$x\;\cfrac{\begin{matrix}[A]^x\\ \vdots\\ B\end{matrix}}{A \to B}\,{\to}\mathrm{I}$$

The square brackets around the formula $A$ indicate that to derive a conditional, we do not actually require an antecedent (i.e., deriving from an assumption is permissible but not required). However, should we use an assumption for the purpose of a derivation, the above method requires us to mark when we have discharged that assumption by writing a variable.

Using this convention, it becomes evident that there are two distinct derivations of $A \to (A \to A)$. On the one hand, we have a derivation where the assumption is discharged at the first conditional introduction.

$$\dfrac{x\ \dfrac{A^x}{A \to A}\,{\to}\mathrm{I}}{A \to (A \to A)}\,{\to}\mathrm{I}$$

And on the other hand, we have a derivation where the assumption is discharged at the second conditional introduction.

$$y\ \dfrac{\dfrac{A^y}{A \to A}\,{\to}\mathrm{I}}{A \to (A \to A)}\,{\to}\mathrm{I}$$

Without these labels, we are unable to distinguish these unique derivations, which is why we must treat labels as essential to **NJ**. Nevertheless, this way of labelling still faces limitations. This is because this limited method of labelling doesn't tell us how many copies of each assumption were discharged and whether there are remaining open assumptions in cases where assumptions are repeated.

But this seems like important information to know when we are trying to identify and distinguish derivations. As we already witnessed, there is a significant difference between the derivation of $A \to (A \to A)$ that discharges the assumption $A$ at the first conditional introduction and the one that discharges at the second. Likewise, there is a significant difference between derivations of the same end-formula that discharge all assumptions and those that discharge only some copies of those assumptions but not all. Put shortly, vacuous introductions are clearly different from strict introductions. That a single introduction rule can do both—as in usual **NJ** systems—is unreasonable given that there is a difference and we know it. What we really have are two introductions rules, one vacuous and the other strict, and we should treat them as such.[24]

Fortunately, we can make these distinctions between derivations—and between vacuous and strict introductions—by requiring every assumption, discharged or not, be labelled. For this reason, we argue that labelled assumptions are an indispensable part of a full presentation of **NJ**. Moreover, to sufficiently manage these assumptions, we must differentiate instances of rules which discharge their assumptions from those that do not. Though many presentations of natural deduction systems omit these requirements, we consider these to be more minimalistically-styled systems even if sound and complete.

Below we present a full sequent-style **NJ** (i.e., with assumption labels).

**Definition 18** (**NJ** Derivation Rules)**.** It should be known that in **NJ** we may have one axiom.

$$\frac{}{x : A \vdash A}\,\text{Ax}$$

As is usual for sequent-style calculi, we regard the left-hand side of sequents as sets of formula assumptions.[25] Notice that one instance of

[24]We argue for two separate conditional introduction rules, but the same considerations about vacuous and strict introductions apply to or-introduction as well. Thus, we will have four separate or-introduction rules.

[25]We do this so as to avoid having structural rules common to sequent-calculi such as weakening and contraction.

the formula $A$, namely the left-hand one, is separated from a variable $x$ with a colon.[26] We use the variable $x$ to mark that particular instance of $A$ as an assumption, but of course, any variable from the set of variables $\{x, y, z, \ldots\}$ will do. Different assumptions, including distinct tokens of the same assumption formula, must employ different variables. If there are more assumptions than available variables, one may use variables with subscripts or natural numbers instead.

The rest of the rules are *derivational* rules that indicate how we may introduce and eliminate logical connectives such as $\to$ with arbitrary logical formulas $A$ and $B$ along with the potentially empty set of formulas represented by $\Gamma$, $\Delta$, and $\Theta$.

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \to B} \to \mathrm{I_v}$$

$$x \; \frac{x : A, \Gamma \vdash B}{\Gamma \vdash A \to B} \to \mathrm{I_s}$$

$$\frac{\Gamma \vdash A \to B \qquad \Delta \vdash A}{\Gamma, \Delta \vdash B} \to \mathrm{E}$$

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B} \wedge \mathrm{I}$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge \mathrm{E}_1$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge \mathrm{E}_2$$

[26] In tree-style **NJ** like the one earlier, we indicate assumptions by writing variable superscripts onto the formulas in question. In sequent-style **NJ**, assumptions are written to the left of the formula before a colon. Here we are still purely working with **NJ** so this notation should be read as assigning an assumption label to a formula. In particular, we are not assigning the type $A$ to a variable $x$ even though this labelling style looks identical to type assignments in a typing context. Of course, the overlap comes as no surprise since under the isomorphism, the typing context in $\lambda^{\to}$ is mapped to the assumptions of a derivation in **NJ**. Despite this link, it's important to keep these two notions distinct when working with systems individually.

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \vee\mathrm{I}_1$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \vee\mathrm{I}_2$$

$$\frac{\Gamma \vdash A \vee B \qquad \Delta \vdash C \qquad \Theta \vdash C}{\Gamma, \Delta, \Theta \vdash C} \vee\mathrm{E}_1$$

$$x \; \frac{\Gamma \vdash A \vee B \qquad \Delta, x : A \vdash C \qquad \Theta \vdash C}{\Gamma, \Delta, \Theta \vdash C} \vee\mathrm{E}_2$$

$$y \; \frac{\Gamma \vdash A \vee B \qquad \Delta \vdash C \qquad \Theta, y : B \vdash C}{\Gamma, \Delta, \Theta \vdash C} \vee\mathrm{E}_3$$

$$x\,y \; \frac{\Gamma \vdash A \vee B \qquad \Delta, x : A \vdash C \qquad \Theta, y : B \vdash C}{\Gamma, \Delta, \Theta \vdash C} \vee\mathrm{E}_4$$

$$\frac{}{\Gamma \vdash \top} \top\,\mathrm{I}$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A} \bot\,\mathrm{E}$$

Note that there are two introduction rules for $\vee$ since the disjunction may be formed from either the right or left disjunct. We also have two elimination rules for $\wedge$ since either conjunct may be inferred from a conjunction. It's also worth noting that in **NJ**, there is a hidden connection between conjunctions and pairs. The **BHK** interpretation stipulates that a derivation of $A \wedge B$ is equivalent to having a pair $\langle \mathcal{A}, \mathcal{B} \rangle$, where $\mathcal{A}$ is a construction of conjunct $A$ and $\mathcal{B}$ is a construction of conjunct $B$.[27] Acknowledging this connection between conjuncts and pairs will be necessary later on when establishing the isomorphism.

[27]The **BHK** interpretation, or the Brouwer–Heyting–Kolmogorov interpretation, provides an informal intuitionistic interpretation of logical connectives and thereby provides the criteria for intuitionistically acceptable proofs.

Besides the serious philosophical motivation to distinguish the two types of introduction, there are other advantages to the system we are proposing here. Firstly, when compared to usual **NJ**, we can prove just as many propositions with the benefits of clarity that come from knowing how each assumption was introduced. In other words, the precision we demand does not harm the strength ordinarily had by intuitionistic natural deduction. And if we ever desire to have a proof of a proposition in usual **NJ**, from our system, it would simply be a matter of removing the subscripts on rule labels to convert the proof into usual **NJ** style. In contrast, if we have a proof in usual **NJ** involving either or-eliminations or conditional introductions, it would be difficult to know which or-elimination or conditional introduction to pick in our system if we don't know how each assumption was introduced. This means our system preserves more 'information', and we can always choose to discard that information later if we want usual **NJ** proofs.

Secondly, as a result of formulating **NJ** in this way, it becomes easier to relevantly prove propositions. In particular, the subsystem represented by $\rightarrow \mathrm{I}_s/\mathrm{E}$, $\wedge\, \mathrm{I}/\mathrm{E}$, $\vee\, \mathrm{I}_{1,2}/\mathrm{E}_4$, $\top\, \mathrm{I}$, and $\bot\, \mathrm{E}$ is relevantly-acceptable. In relevance logic, vacuous introductions violate the organizing principle that the logic is founded upon: valid proofs should consist of propositions that are sufficiently *relevant* to each other.[28] Different relevance logics have different ways of formalizing this principle, but at the very least, it's clear that respecting *relevance* requires discarding $\rightarrow \mathrm{I}_v$ and other vacuous introductions since antecedent formulas would not be obviously relevant to consequent ones. The advantage that our **NJ** presents is that we know that any proof contained

[28] Arnon Avron. "Whither Relevance Logic?" In: *Journal of Philosophical Logic* 21.3 (1992), pp. 243–281. DOI: 10.1007/BF00260930.

by the subsystem is relevantly valid. In a sense then, we get knowledge of a proof's relevance validity for free.

In summary, the two advantages just discussed both indicate that it is easy to shift between logics using our system of **NJ**. For it's a simple matter to remove subscripts so that we end up with usual **NJ**, to add rules so that we end up with **NK**, or to remove some rules and add others to get a full relevance logic.[29] We have this flexibility because our system puts all restrictions in the rules, and it is relatively straightforward to add or discard rules to accommodate different logics.

## Proof Normalization

Before proceeding at last to the isomorphism, we have one final subject to discuss with respect to **NJ**. Since natural deduction derivations elucidate paths of reasoning from a set of assumptions, a set that can also be empty, these derivations may also serve to convey the *structure* of proofs.[30] An interesting question that has been taken up in recent decades is what makes a proof, mathematical or otherwise, informative, insightful, beautiful, or elegant. Somewhat related to the logical structure of proofs, we can always investigate what makes a proof structurally complex or simple (and perhaps simple is what we sometimes mean by 'elegant'). Proof normalization is just one formal means of capturing that notion of structural simplicity.

Essentially, we call a proof 'normal' if the proof is direct: that is, for a particular logical connective, say $\vee$, we do not apply its elimination rule im-

[29]**NK** is natural deduction for classical logic.

[30]In fact, natural deduction derivations are used in proof theory as a tool in investigating the structure of proofs.

mediately after having applied its introduction rule in a derivation. Normalization is the process of making proofs direct (i.e., the process of eliminating any portion of a proof that contains applications of an introduction rule of a certain kind that are followed immediately by applications of an elimination rule of the same kind). It is a significant finding in proof theory that we can always normalize a proof.[31] That is, it's always possible to construct a normal proof for something that was originally derived non-directly.[32]

But just how are 'normal' and 'non-direct' defined exactly? These terms can be defined using the notion of detour, which can be defined like so.

**Definition 19** (Detour)**.** Let $\pi_n$ represent a subderivation. A detour in an **NJ** proof occurs when an elimination rule is applied immediately after an introduction for the same logical connective or immediately after falsum elimination (i.e., $\bot$E). A detour formula is the right-hand formula that appears after an introduction rule (or falsum elimination) and before an elimination rule of the same kind. For instance,

$$y\,x\ \dfrac{\dfrac{\begin{array}{c}\vdots\,\pi_1\\ \Gamma \vdash B\end{array}}{\Gamma \vdash A \vee B}\vee \mathrm{I}_2 \qquad \begin{array}{c}\vdots\,\pi_2\\ \Delta, y:A \vdash C\end{array} \qquad \begin{array}{c}\vdots\,\pi_3\\ \Theta, x:B \vdash C\end{array}}{\Gamma,\Delta,\Theta \vdash C}\vee \mathrm{E}_4$$

is a proof of $C$ from $\Gamma,\Delta,\Theta$ that ends in a detour. In other words, the elimination rule for $\vee$ was applied after the introduction rule for $\vee$. Notice that a proof of $C$ would have been more directly obtained by stitching together

[31]For more details about normalization and proof theory, see Paolo Mancosu, Richard Zach, and Sergio Galvan. *An Introduction to Proof Theory: Normalization, Cut-Elimination, and Consistency Proofs*. Oxford, New York: Oxford University Press, 2021.

[32]We use the term 'non-directly' so as to avoid using 'indirectly' since we are not referring to indirect proofs (i.e., proofs by contradiction).

the subproofs that end in $\Gamma \vdash B$ and $\Theta, x : B \vdash C$.[33] Having defined 'detour', we say that a proof is normal when it contains no instances of a detour. A proof is 'non-direct' or 'non-normal' otherwise.

Though already briefly described above, it will be helpful to provide an example of normalization. Let us reuse our example detour: if we suppose that none of the preceding subproofs contain a detour, the proof would be normalized as follows.

$$\begin{array}{c} \vdots\, \pi_1 \\ \Gamma \vdash B \\ \vdots\, \pi_3[\pi_1/x : B \vdash B] \\ \Gamma, \Theta \vdash C \end{array}$$

This may appear confusing, but bear in mind that the normalization involves a number of steps that are not entirely obvious at first. To normalize the detour, we appeal to proof substitution, and we shall expound the process. As a reminder, the detour we wish to normalize is the following.

$$y\,x\;\dfrac{\dfrac{\begin{array}{c}\vdots\,\pi_1\\ \Gamma\vdash B\end{array}}{\Gamma\vdash A\vee B}\vee \mathrm{I}_2 \qquad \begin{array}{c}\vdots\,\pi_2\\ \Delta, y:A\vdash C\end{array}\qquad \begin{array}{c}\vdots\,\pi_3\\ \Theta, x:B\vdash C\end{array}}{\Gamma,\Delta,\Theta\vdash C}\vee \mathrm{E}_4$$

Note that we would like to derive $\Gamma, \Delta, \Theta \vdash C$. So far, to derive $C$ from the assumption(s) contained in $\Gamma, \Delta, \Theta$, we have only the two sequents $\Delta, y : A \vdash C$ and $\Theta, x : B \vdash C$. Let us choose the latter, which gives us the following incomplete derivation.

[33] It's worth noting that this method would result in a derivation ending with sequent $\Gamma, \Theta \vdash C$ rather than $\Gamma, \Delta, \Theta \vdash C$. This illustrates a general pattern where normal derivations tend to use fewer assumptions than non-normal ones.

$$\begin{array}{c} \vdots\,\pi_3 \\ \Theta, x : B \vdash C \end{array}$$

Since the formula $B$ appears on the left side of the turnstile, we can deduce that the axiom $x : B \vdash B$ must appear from the beginning because there are no other rules that can introduce a formula on the left of the turnstile. We have then the same incomplete derivation with the initial axiom:

$$\begin{array}{c} x : B \vdash B \\ \vdots\,\pi_3 \\ \Theta, x : B \vdash C \end{array}$$

At this point, we need to create a method that will allow us to eliminate the formula $B$ in the end-sequent $\Theta, x : B \vdash C$. Recall that the formula $B$ appears on the left side only for the reason that it originally appears on the left side of the axiom $x : B \vdash B$. Thus, one idea might be to remove that first occurrence of $B$ and replace it with something else. But what to replace it with? If we revisit what we can derive using our various $\pi_n$, we see that $\pi_1$ derives $\Gamma \vdash B$. Here we've found something to replace $B$: the set of assumptions $\Gamma$. In other words, we can substitute the proof $\pi_1$ for every occurrence of the axiom $x : B \vdash B$ as in the following.

$$\begin{array}{c} \vdots\,\pi_1 \\ \overset{\Gamma}{\cancel{x : B}} \vdash B \\ \vdots\,\pi_3[\pi_1/x : B \vdash B] \\ \Theta, \overset{\Gamma}{\cancel{x : B}} \vdash C \end{array}$$

This last step consists essentially of proof substitution where we substitute the proof $\pi_1$ for $x : B \vdash B$, and $\pi_3[\pi_1/x : B \vdash B]$ signifies this particular substitution within the proof $\pi_3$. Since we already stipulated that the formulas $\Gamma, \Theta$ on the left hand side of the turnstile should be understood as a set, we obtain the expected result.

$$\begin{array}{rl} & \vdots\, \pi_1 \\ \Gamma \vdash & B \\ & \vdots\, \pi_3[\pi_1/x : B \vdash B] \\ \Gamma,\Theta \vdash & C \end{array}$$

In this way, we normalize the proof of $\Gamma, \Delta, \Theta \vdash C$ to obtain a new but nevertheless related proof of $\Gamma, \Theta \vdash C$ where that detour is eliminated.

Having briefly demonstrated what proof substitution looks like, it's important to know that the method of proof substitution works generally.

**Theorem 20** (Proof Substitution)**.** *Let $B$ be a formula $\notin \Gamma$, and let $x$ be the variable assumption label assigned to $B$ when it occurs as an assumption. If $\Pi_1$ is a proof of formula $A$ from the set of assumptions $\Gamma \cup \{B\}$ and $\Pi_2$ is a proof of formula $B$ from the set of assumptions $\Delta$, then $\Pi_1[\Pi_2/x : B \vdash B]$ is a proof of formula $A$ from a subset of $\{\Gamma \cup \Delta\}$.*

*Proof.* See Paolo Mancosu, Richard Zach, and Sergio Galvan. *An Introduction to Proof Theory: Normalization, Cut-Elimination, and Consistency Proofs*. Oxford, New York: Oxford University Press, 2021, Lemma 4.8, p. 113. □

Though the proof substitution works generally to eliminate detours, there is nevertheless a limitation to the method. Consider the following derivation.[34]

[34] To some, the derivation may strike as suspiciously familiar. This is because it is familiar: it is the corresponding **NJ** proof of our example program on Page 31.

$$x\,y\;\dfrac{\begin{array}{c}\vdots\,\pi_0\\ \Gamma \vdash A\vee B\end{array} \quad \dfrac{\begin{array}{c}\Gamma, x:A\vdash A\\ \vdots\,\pi_1\\ \Gamma,x:A\vdash C\end{array} \quad \begin{array}{c}\Gamma, x:A\vdash A\\ \vdots\,\pi_2\\ \Gamma,x:A\vdash D\end{array}}{\Gamma,x:A\vdash C\wedge D}\wedge\mathrm{I} \quad \begin{array}{c}\Gamma,y:B\vdash B\\ \vdots\,\pi_3\\ \Gamma,y:B\vdash C\wedge D\end{array}}{\dfrac{\Gamma\vdash C\wedge D}{\Gamma\vdash C}\wedge\mathrm{E}_1}\vee\mathrm{E}_4$$

Strictly speaking, no detour occurs in the above derivation since no immediately successive applications of I/E appear for a single logical operator. Despite that, we can intuit that this proof is not yet in normal form since $\wedge\mathrm{E}_1$ follows $\wedge$I: it is just that $\vee\mathrm{E}_4$ inconveniently appears between them. This fact prevents us from being able to straightforwardly normalize the proof by using proof substitution.

Intuitively speaking, this should count as a kind of detour even though it doesn't exactly fit the definition. No matter our wishes, it still falls short of the definition, so we cannot proceed to normalize as usual. Faced with this limitation, we can generalize the notion of a detour by defining *cut segments*. We can then find a method that will eliminate cut segments from our derivations.

**Definition 21** (Cut-Segment in Sequent-Style **NJ**). [35] A cut segment of length $n$ in a proof $\Pi$ is a sequence $A_1, \ldots, A_n$ of right-hand formulas (i.e., formulas right of $\vdash$) appearing in $\Pi$ that satisfy the following conditions.

1. $A_1$ is the conclusion of an introduction rule.

2. $A_i$ (where $1 \leq i < n$) is a minor premise of $\vee$E, and $A_{i+1}$ is the conclusion of $A_i$.

[35] For details on usual **NJ**, see Mancosu, Zach, and Galvan, op. cit.

3. $A_n$ is the major premise of an elimination rule.

4. All $A_i$ are the same formula.

The minor premises of $\vee \mathrm{E}_i$ are the right-hand formulas of the middle sequent and right-most sequent, which appear above the inference line in the rule. For single-sequent eliminations rules (i.e., $\wedge \mathrm{E}_i$ and $\bot\mathrm{E}$), there are no minor premises, and the major premise is simply the right-hand formula above the inference line. For multi-sequent elimination rules (i.e., $\to\mathrm{E}$ and $\vee \mathrm{E}_i$), the major premise is the right-hand formula of the left-most sequent appearing above the inference line, and the remaining right-hand formulas are minor premises.

Notice that this definition is a generalization of a detour. In fact, detour formulas turn out to be cut segments of length 1 where $A_1 = A_n$. Having defined the notion of a cut segment, we now know where our example proof goes wrong. We rewrite the proof and highlight its cut segment below.

$$
x\,y\;\dfrac{\begin{array}{c}\vdots\,\pi_0\\ \Gamma \vdash A\vee B\end{array} \qquad \dfrac{\begin{array}{c}\Gamma, x:A\vdash A\\ \vdots\,\pi_1\\ \Gamma,x:A\vdash C\end{array}\qquad \begin{array}{c}\Gamma,x:A\vdash A\\ \vdots\,\pi_2\\ \Gamma,x:A\vdash D\end{array}}{\Gamma,x:A\vdash C\wedge D}\,\wedge\mathrm{I} \qquad \begin{array}{c}\Gamma,y:B\vdash B\\ \vdots\,\pi_3\\ \Gamma,y:B\vdash C\wedge D\end{array}}{\dfrac{\Gamma\vdash C\wedge D}{\Gamma\vdash C}\,\wedge\mathrm{E}_1}\,\vee\mathrm{E}_4
$$

Our general method to deal with cut segments of length > 1 will be to permute applications of eliminations up the proof tree until they are directly beneath the appropriate introduction inference. In which case, they become a cut segment of length 1 (i.e., they become a detour), and we normalize the proof using proof substitution (or simply use an existing subproof) as is usually done to detours.

In the example proof above, we have a cut segment of length 2. To normalize, first we apply a permutation conversion to push the $\land E_1$ over the $\lor E_4$ like so. This results in a cut segment of length 1.

$$x\,y\ \dfrac{\begin{array}{c}\vdots\,\pi_0\\ \Gamma\vdash A\lor B\end{array} \qquad \dfrac{\dfrac{\begin{array}{c}\Gamma,x:A\vdash A\\ \vdots\,\pi_1\\ \Gamma,x:A\vdash C\end{array} \qquad \begin{array}{c}\Gamma,x:A\vdash A\\ \vdots\,\pi_2\\ \Gamma,x:A\vdash D\end{array}}{\Gamma,x:A\vdash \color{blue}{C\land D}}\land\mathrm{I}}{\Gamma,x:A\vdash C}\land\mathrm{E}_1 \qquad \dfrac{\begin{array}{c}\Gamma,y:B\vdash B\\ \vdots\,\pi_3\\ \Gamma,y:B\vdash C\land D\end{array}}{\Gamma,y:B\vdash C}\land\mathrm{E}_1}{\Gamma\vdash C}\lor\mathrm{E}_4$$

Now supposing that none of the subproofs $\pi_0$, $\pi_1$, $\pi_2$, or $\pi_3$ contain any cut segments, we can finish normalizing the proof by simply using the existing subproof $\pi_1$, as shown below.

$$x\,y\ \dfrac{\begin{array}{c}\vdots\,\pi_0\\ \Gamma\vdash A\lor B\end{array} \qquad \begin{array}{c}\Gamma,x:A\vdash A\\ \vdots\,\pi_1\\ \Gamma,x:A\vdash C\end{array} \qquad \dfrac{\begin{array}{c}\Gamma,y:B\vdash B\\ \vdots\,\pi_3\\ \Gamma,y:B\vdash C\land D\end{array}}{\Gamma,y:B\vdash C}\land\mathrm{E}_1}{\Gamma\vdash C}\lor\mathrm{E}_4$$

As was just shown, cut segment elimination can effectively normalize a proof. Fortunately, the method of cut segment elimination works in general.

**Theorem 22** (Cut Segment Elimination for **NJ**)**.** *Every proof $\Pi$ of $A$ containing cut segments can be converted into a proof $\Pi'$ of $A$ without cut segments. In other words, every non-normal proof in **NJ** has a corresponding normal proof.*

*Proof.* See ibid., Theorem 4.29, p. 134. □

# Chapter 3

# What is the Curry-Howard Correspondence?

At last we have the conceptual background necessary to see exactly how the correspondence between proofs and programs is made. In fact, the correspondence is drawn on two levels: First, the typing judgments of the typing rules in Definition 14 (particularly, the ones on the right-hand side) correspond to the **NJ** derivation rules in Definition 18. Second, the process of $\beta$-reduction in the simply-typed $\lambda$-calculus corresponds to proof normalization. These two lines of correspondence will become clearer with some concrete illustration.

## 3.1 Approaching the Correspondence

### The First Level of Correspondence

Below, we put the typing rules of the simply-typed lambda calculus. We can see that the simply-typed $\lambda$-calculus typing judgments correspond to **NJ** rules, and in particular, we can see how formulas of **NJ** proofs correspond to the types in type derivations. To better illustrate this, we colour the typing judgments—and other aspects that align with **NJ**—in blue. We also write the label of the **NJ** rule on the right side.

$$\text{var}\ \frac{}{\Gamma, x : A \vdash x : A}\ \text{Ax}$$

$$\text{app}\ \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}\ \to \text{E}$$

$$^{1}\ \text{abs}_{\text{v}}\ \frac{\Gamma \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ \to \text{I}_{\text{v}}$$

$$^{2}\ \text{abs}_{\text{s}}\ x\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ \to \text{I}_{\text{s}}$$

$$\text{pair}\ \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \times B}\ \times \text{I}$$

$$\pi_1\ \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_1 M : A}\ \times \text{E}_1$$

$$\pi_2\ \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_2 M : B}\ \times \text{E}_2$$

$$\text{inj}_1\ \frac{\Gamma \vdash M : A}{\Gamma \vdash \text{inj}_1 M : A + B}\ +\text{I}_1$$

$$\text{inj}_2\ \frac{\Gamma \vdash M : B}{\Gamma \vdash \text{inj}_2 M : A + B}\ +\text{I}_2$$

[1]The $\text{abs}_{\text{v}}$ rule works for any $A$ and on the condition that $x$ is not free in $M$.
[2]This $\text{abs}_{\text{s}}$ rule works on the condition that $x$ is free in $M$.

$$\text{case}\ \frac{\Gamma \vdash M : A+B \qquad \Gamma \vdash N : C \qquad \Gamma \vdash O : C}{\Gamma \vdash (\text{case}\ M\ \text{of}\ (x)N;(y)O) : C}\ +\text{E}_1$$

$$\text{case}\ x\ \frac{\Gamma \vdash M : A+B \qquad \Gamma, x : A \vdash N : C \qquad \Gamma \vdash O : C}{\Gamma \vdash (\text{case}\ M\ \text{of}\ (x)N;(y)O) : C}\ +\text{E}_2$$

$$\text{case}\ y\ \frac{\Gamma \vdash M : A+B \qquad \Gamma \vdash N : C \qquad \Gamma, y : B \vdash O : C}{\Gamma \vdash (\text{case}\ M\ \text{of}\ (x)N;(y)O) : C}\ +\text{E}_3$$

$$\text{case}\ x\,y\ \frac{\Gamma \vdash M : A+B \qquad \Gamma, x : A \vdash N : C \qquad \Gamma, y : B \vdash O : C}{\Gamma \vdash (\text{case}\ M\ \text{of}\ (x)N;(y)O) : C}\ +\text{E}_4$$

$$*\ \frac{}{\Gamma \vdash * : 1}\ 1\ \text{I}$$

[3] $$\text{abort}\ \frac{\Gamma \vdash M : 0}{\Gamma \vdash \text{abort}_A M : A}\ 0\ \text{E}$$

The above illustration may reveal some discrepancies between the two systems, however these discrepancies are quite superficial. Though the symbols $\times$ and $+$ are written, we can read these as $\wedge$ and $\vee$ respectively. Thus, for instance, the $\times$ E rule is understood as the $\wedge$ E rule, and the $+$ I rule is understood as the $\vee$ I rule.

As for 1 and 0, 1 should be understood as $\top$ (i.e., as verum or top), and 0 should be understood as $\perp$ (i.e., falsum or bottom). Hence the typing judgment of $*$ corresponds to the $\top$ I rule, which is written below.

$$\frac{}{\Gamma \vdash \top}\ \top\ \text{I}$$

Likewise, the typing judgment of abort corresponds to the $\perp$ E rule:

[3]The problem with the abort rule is that we actually have infinitely many $\text{abort}_A$ constructors (i.e., we should have $\text{abort}_B$, $\text{abort}_{A\to B}$, $\text{abort}_{A\times B}$, and so on). For other constructors such as $\pi$ or case, we defined differerent rules to capture the differences in output or input. To be consistent with this practise, we would need to define every rule for every abort constructor, but this is evidently impossible. Moreover, it seems that we would need to define infinitely many $\perp$ E rules to truly preserve the correspondence between **NJ** and $\lambda^{\to}$. Though we acknowledge these remaining problems, we presently have no solution besides leaving the abort rule as it is and glossing over these subtleties.

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A} \bot \mathrm{E}$$

Notice as well that compared to the typing rules in Definition 14, we now have more than one abs rule and case rule. Our reasons for doing so are identical to those when we introduced various →I and ∨E inference rules in **NJ**. To reiterate, we maintain that a fundamental distinction must be made between vacuous and strict discharging of assumptions. By introducing various versions of abs and case, we are merely ensuring that this distinction is similarly respected in $\lambda^{\rightarrow}$. This is important if we would like to establish a correspondence that is also isomorphic.

Infrequently, one might find the correspondence illustrated thus:

| $\lambda^{\rightarrow}$ | **NJ** |
|---|---|
| var | Ax |
| app | $\rightarrow$ E |
| abs | $\rightarrow$ I |
| pair | $\wedge$ I |
| $\vdots$ | $\vdots$ |

As helpful as this may be, such an illustration eschews the real line of correspondence being made. That is, the correspondence being drawn here is not between sets of rules, or even between type constructors and logical connectives, but rather between the typing judgments in $\lambda^{\rightarrow}$ and **NJ** derivations. Of course, to draw this primary correspondence between typing judgments and derivations, we require the other correspondences to hold as well. It's just that we are trying to emphasize first and foremost the correspondence between typing judgments and **NJ** derivations. Our goal is not to merely pattern-match constituents of $\lambda^{\rightarrow}$ to constituents of **NJ** though we require these patterns be present in any case.

## The Second Level of Correspondence

As was already suggested, the correspondence between the simply-typed $\lambda$-calculus and **NJ** goes deeper than the discovery of matching terms in both calculi. In particular, the two calculi have processes that correspond in structure: normalizing proofs correspond to $\beta$-reducing lambda terms, and vice versa. In other words, not only does the simply-type $\lambda$-calculus and **NJ** 'look alike', they 'act alike' as well.

That normalization of detours is isomorphic to $\beta$-reduction can be shown as follows.

**Theorem 23** (Isomorphism Between $\beta$-Reduction and Detour Normalization)**.** *If* $\Pi$ *is a proof of* $t : A$ *from a set of assumptions* $\Gamma$ *and contains a detour, then removing that detour results in a proof* $\Pi'$ *of* $t' : A$ *from a subset of* $\Gamma$ *where* $t \rightarrow_{\beta} t'$.

*Proof.* Suppose that $\Pi$ ends in the detour. We know that in this case, $t$ must be a $\beta$-redex (i.e., a $\beta$-reducible term). If $\Pi$ doesn't end in a detour, then a subproof ends in a detour. The result of that subproof must be a subterm of $t$ where $t$ is a $\beta$-redex.

**Case 1a.** Suppose the proof ends in the detour $\rightarrow \mathrm{I}_s \,/ \rightarrow \mathrm{E}$. Then $t$ is $(\lambda x.t)s$. We would have for instance the following.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\pi_1\\ x : A, \Gamma \vdash t : B\end{array}}{\Gamma \vdash \lambda x.t : A \to B}\to \mathrm{I_s} \qquad \begin{array}{c}\vdots\,\pi_2\\ \Delta \vdash s : A\end{array}}{\Gamma, \Delta \vdash (\lambda x.t)s : B}\to \mathrm{E}$$

To normalize this proof, we again employ the method of proof substitution. To start we have the following:

$$\begin{array}{c}\Gamma, x : A \vdash x : A\\ \vdots\,\pi_1\\ x : A, \Gamma \vdash t : B\end{array}$$

Then we begin to eliminate occurrences of $x : A$ and substitute $\pi_2$ for $x : A \vdash x : A$.

$$\begin{array}{c}\vdots\,\pi_2\\ \Gamma, \overset{\Delta}{\cancel{x : A}} \vdash \overset{s:A}{\cancel{x : A}}\\ \vdots\,\pi_1[\pi_2/x : A \vdash x : A]\\ \Gamma, \Delta \vdash t[s/x] : B\end{array}$$

The final result is the following.

$$\begin{array}{c}\vdots\,\pi_2\\ \Delta \vdash s : A\\ \vdots\,\pi_1[\pi_2/x : A \vdash x : A]\\ \Gamma, \Delta \vdash t[s/x] : B\end{array}$$

Notice that prior to normalization, the type of $B$ was $(\lambda x.t)s$ whereas after normalization, the type became $t[s/x]$. That is, $(\lambda x.t)s \to_\beta t[s/x]$.

**Case 1b.** Suppose the proof ends in the detour $\to \mathrm{I_v}$ / $\to \mathrm{E}$ . Then $t$ is $(\lambda x.t)s$.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\pi_1\\ \Gamma \vdash t : B\end{array}}{\Gamma \vdash \lambda x.t : A \to B}\to \mathrm{I_v} \qquad \begin{array}{c}\vdots\,\pi_2\\ \Delta \vdash s : A\end{array}}{\Gamma, \Delta \vdash (\lambda x.t)s : B}\to \mathrm{E}$$

In this case, no substitution is required. To normalize the proof, we simply use the subproof $\pi_1$.

$$\begin{array}{c} \vdots\,\pi_1 \\ \Gamma \vdash t : B \end{array}$$

Moreover, we know that $(\lambda x.t)s \rightarrow_\beta t[s/x]$. But since $x$ is not free for $t$, $t[s/x]$ is just $t$. In other words, for this case we have $(\lambda x.t)s \rightarrow_\beta t$.

**Case 2a.** Suppose the proof ends in the detour $\times\mathrm{I} \,/ \times \mathrm{E}_1$. Then $t$ is $\pi_1\langle M, N\rangle$. For instance, $\Pi$ may be like the following.

$$\cfrac{\cfrac{\begin{array}{c}\vdots\,\pi_1\\ \Gamma \vdash M : A\end{array} \qquad \begin{array}{c}\vdots\,\pi_2\\ \Delta \vdash N : B\end{array}}{\Gamma, \Delta \vdash \langle M, N\rangle : A \times B}\times\mathrm{I}}{\Gamma, \Delta \vdash \pi_1\langle M, N\rangle : A}\times\mathrm{E}_1$$

In fact, we already have a derivation of $A$, namely, $\pi_1$. Therefore, $\Pi'$ is simply $\pi_1$ and $\pi_1\langle M, N\rangle \rightarrow_\beta M$.

$$\begin{array}{c} \vdots\,\pi_1 \\ \Gamma \vdash M : A \end{array}$$

**Case 2b.** Suppose the proof ends in the detour $\times\mathrm{I} \,/ \times \mathrm{E}_2$. Then $t$ is $\pi_2\langle M, N\rangle$. We may reuse our last example for $\Pi$.

$$\cfrac{\cfrac{\begin{array}{c}\vdots\,\pi_1\\ \Gamma \vdash M : A\end{array} \qquad \begin{array}{c}\vdots\,\pi_2\\ \Delta \vdash N : B\end{array}}{\Gamma, \Delta \vdash \langle M, N\rangle : A \times B}\times\mathrm{I}}{\Gamma, \Delta \vdash \pi_2\langle M, N\rangle : B}\times\mathrm{E}_2$$

This case proceeds similarly to **Case 2a**. That is, $\Pi'$ is $\pi_2$ and $\pi_2\langle M, N\rangle \rightarrow_\beta N$.

**Case 3a.** Suppose the proof ends in the detour $+\mathrm{I}_i/+\mathrm{E}_1$.[4] Then $t$ is (case $\mathrm{inj}_i M$ of $(x_1)N;(x_2)O)$ where $x_1$ and $x_2$ do not appear free in $M$.

We have for instance then the following.

$$\dfrac{\dfrac{\begin{matrix}\vdots\,\pi_0\\ \Gamma \vdash M : A_i\end{matrix}}{\Gamma \vdash \mathrm{inj}_i M : A_1 + A_2}\,\vee\mathrm{I}_i \qquad \begin{matrix}\vdots\,\pi_1\\ \Gamma \vdash N_1 : C\end{matrix} \qquad \begin{matrix}\vdots\,\pi_2\\ \Gamma \vdash N_2 : C\end{matrix}}{\Gamma \vdash (\text{case } \mathrm{inj}_i M \text{ of } (x_1)N_1;(x_2)N_2) : C}\,+\mathrm{E}_1$$

Since neither $N_1$ or $N_2$ of type $C$ were introduced with an assumption (i.e., were introduced vacuously), we can eliminate the detour in the following way.

$$\begin{matrix}\vdots\,\pi_i\\ \Gamma \vdash N_i : C\end{matrix}$$

Recall that (case $\mathrm{inj}_i M$ of $(x_1)N_1;(x_2)N_2) \rightarrow_\beta N_1[N_2/x_1]$ or $N_2[N_1/x_2]$ depending on which inj rule was used. But since both $x_1$ and $x_2$ are not free in $\mathrm{inj}_i M$, $N_1[N_2/x_1]$ simply becomes $N_1$ and $N_2[N_1/x_2]$ simply becomes $N_2$. Either way, (case $\mathrm{inj}_i M$ of $(x_1)N_1;(x_2)N_2) \rightarrow_\beta N_i$.

**Case 3b.** Suppose the proof ends in the detour $+\mathrm{I}_i/+\mathrm{E}_2$. Then $t$ is (case $\mathrm{inj}_i M$ of $(x_1)N;(x_2)O)$ where $x_1$ is free in $M$ but $x_2$ is not.

We have for instance then the following.

$$x_1\ \dfrac{\dfrac{\begin{matrix}\vdots\,\pi_0\\ \Gamma \vdash M : A_i\end{matrix}}{\Gamma \vdash \mathrm{inj}_i M : A_1 + A_2}\,\vee\mathrm{I}_i \qquad \begin{matrix}\Gamma, x_1 : A_1 \vdash x_1 : A_1\\ \vdots\,\pi_1\\ \Gamma, x_1 : A_1 \vdash N : C\end{matrix} \qquad \begin{matrix}\vdots\,\pi_2\\ \Gamma \vdash O : C\end{matrix}}{\Gamma \vdash (\text{case } \mathrm{inj}_i M \text{ of } (x_1)N;(x_2)O) : C}\,+\mathrm{E}_1$$

[4]Here $i$ can be 1 or 2, and this also applies to cases **3b**, **3c**, and **3d**. We do this to avoid having to write two copies of almost identical proofs in each of these four cases. That is, it is simpler to cover the general case $+\mathrm{I}_i/+\mathrm{E}_1$ than to cover both $+\mathrm{I}_1/+\mathrm{E}_1$ and $+\mathrm{I}_2/+\mathrm{E}_1$, for example.

Suppose we are normalizing the proof where $\text{inj}_1$ appears. Then $\Pi'$ would be obtained by following method. First we take $\pi_1$.

$$\begin{array}{c} \Gamma, x_1 : A_1 \vdash x_1 : A_1 \\ \vdots\, \pi_1 \\ \Gamma, x_1 : A_1 \vdash N : C \end{array}$$

Then we use proof substitution to replace every occurrence of $x_1 : A_1 \vdash x_1 : A_1$ with $\pi_0$.

$$\begin{array}{c} \vdots\, \pi_0 \\ \Gamma, \overset{\Gamma}{\cancel{x_1 : A_1}} \vdash \overset{M:A_1}{\cancel{x_1 : A_1}} \\ \vdots\, \pi_1[\pi_0 / x_1 : A_1 \vdash x_1 : A_1] \\ \Gamma \vdash N : C \end{array}$$

Ultimately $\Pi'$ is the following proof, and we have (case $\text{inj}_1 M$ of $(x_1)N;(x_2)O) \rightarrow_\beta N$.

$$\begin{array}{c} \vdots\, \pi_0 \\ \Gamma, \Gamma \vdash M : A_1 \\ \vdots\, \pi_1[\pi_0 / x_1 : A_1 \vdash x_1 : A_1] \\ \Gamma \vdash N : C \end{array}$$

Suppose, however, we are normalizing the proof where $\text{inj}_2$ appears instead. Then we proceed similarly to **Case 3a**. $\Pi'$ becomes the following proof and (case $\text{inj}_2 M$ of $(x_1)N;(x_2)O) \rightarrow_\beta O$.

$$\begin{array}{c} \vdots\, \pi_2 \\ \Gamma \vdash O : C \end{array}$$

**Case 3c.** Suppose the proof ends in the detour $+\text{I}_i / +\text{E}_3$. Then $t$ is (case $\text{inj}_i M$ of $(x_1)N;(x_2)O)$ where $x_1$ is not free in $M$ but $x_2$ is.

We have for instance then the following.

$$x_2\ \dfrac{\dfrac{\begin{array}{c}\vdots\,\pi_0\\ \Gamma \vdash M : A_i\end{array}}{\Gamma \vdash \mathrm{inj}_i M : A_1 + A_2}\vee\mathrm{I}_i \qquad \begin{array}{c}\vdots\,\pi_1\\ \Gamma \vdash N : C\end{array} \qquad \begin{array}{c}\Gamma, x_2 : A_2 \vdash x_2 : A_2\\ \vdots\,\pi_2\\ \Gamma, x_2 : A_2 \vdash O : C\end{array}}{\Gamma \vdash (\text{case inj}_i M \text{ of } (x_1)N;(x_2)O) : C}+\mathrm{E}_1$$

Suppose we are normalizing the proof where $\mathrm{inj}_1$ appears. Then we proceed similarly to **Case 3a**. $\Pi'$ would be the following proof and $(\text{case inj}_1 M \text{ of } (x_1)N;(x_2)O) \rightarrow_\beta N$.

$$\begin{array}{c}\vdots\,\pi_1\\ \Gamma \vdash N : C\end{array}$$

If, however, we are normalizing the proof where $\mathrm{inj}_2$ appears, then $\Pi'$ is obtained by using proof substitution and $(\text{case inj}_2 M \text{ of } (x_1)N;(x_2)O) \rightarrow_\beta O$.

First we isolate the proof $\pi_2$.

$$\begin{array}{c}\Gamma, x_2 : A_2 \vdash x_2 : A_2\\ \vdots\,\pi_2\\ \Gamma, x_2 : A_2 \vdash O : C\end{array}$$

Then we replace all instances of $x_2 : A_2 \vdash x_2 : A_2$ with the proof $\pi_0$ like so.

$$\begin{array}{c}\vdots\,\pi_0\\ \Gamma, \overset{\Gamma}{\cancel{x_2 : A_2}} \vdash \overset{M:A_2}{\cancel{x_2 : A_2}}\\ \vdots\,\pi_2[\pi_0/x_2 : A_2 \vdash x_2 : A_2]\\ \Gamma, \overset{\Gamma}{\cancel{x_2 : A_2}} \vdash O : C\end{array}$$

$\Pi'$ then becomes the following derivation.

$$\begin{array}{c}\vdots\,\pi_0\\ \Gamma \vdash M : A_2\\ \vdots\,\pi_2[\pi_0/x_2 : A_2 \vdash x_2 : A_2]\\ \Gamma \vdash O : C\end{array}$$

**Case 3d.** Suppose the proof ends in the detour $+\mathrm{I}_i/+\mathrm{E}_4$. Then $t$ is (case $\mathrm{inj}_i M$ of $(x_1)N_1;(x_2)N_2)$ where both $x_1$ and $x_2$ are free in $M$.

We then have for instance the following.

$$x_1\,x_2\;\dfrac{\dfrac{\begin{matrix}\vdots\,\pi_0\\ \Gamma\vdash M:A_i\end{matrix}}{\Gamma\vdash \mathrm{inj}_i M : A_1+A_2}\vee\mathrm{I}_i \qquad \begin{matrix}\Gamma, x_1:A_1\vdash x_1:A_1\\ \vdots\,\pi_1\\ \Gamma\vdash N_1:C\end{matrix} \qquad \begin{matrix}\Gamma, x_2:A_2\vdash x_2:A_2\\ \vdots\,\pi_2\\ \Gamma\vdash N_2:C\end{matrix}}{\Gamma\vdash (\text{case } \mathrm{inj}_i M \text{ of } (x_1)N_1;(x_2)N_2):C}+\mathrm{E}_1$$

Since both $x_1$ and $x_2$ are free in $M$, we use proof substitution to normalize the proofs where $\mathrm{inj}_1$ and $\mathrm{inj}_2$ appear. Let us generalize and normalize the proof where $\mathrm{inj}_i$ appears so that we cover both cases in a single attempt. First we take $\pi_i$ as per usual.

$$\begin{matrix}\Gamma, x_i:A_i\vdash x_i:A_i\\ \vdots\,\pi_i\\ \Gamma, x_i:A_i\vdash N_i:C\end{matrix}$$

Then we replace every occurrence of $x_i:A_i\vdash x_i:A_i$ with $\pi_0$

$$\begin{matrix}\vdots\,\pi_0\\ \Gamma, \overset{\Gamma}{\cancel{x_i:A_i}}\vdash \overset{M:A_i}{\cancel{x_i:A_i}}\\ \vdots\,\pi_i[\pi_0/x_i:A_i\vdash x_i:A_i]\\ \Gamma, \overset{\Gamma}{\cancel{x_i:A_i}}\vdash N_i:C\end{matrix}$$

We ultimately then have the following proof for $\Pi'$ and (case $\mathrm{inj}_i M$ of $(x_1)N_1;(x_2)N_2) \rightarrow_\beta N_i$

$$\begin{array}{l} \qquad \vdots\, \pi_0 \\ \Gamma \vdash M : A_i \\ \qquad \vdots\, \pi_i[\pi_0/x_i : A_i \vdash x_i : A_i] \\ \Gamma \vdash N_i : C \end{array}$$

**Case 4.** Suppose the proof ends in the detour $\bot$E/ $x$E where '$x$E' stands for any elimination rule.

As an example, let us suppose $t$ is $\pi_1(\mathrm{abort}_{A\times B} M)$ and we have the following $\Pi$.

$$\dfrac{\dfrac{\begin{array}{l}\qquad \vdots\, \pi_1 \\ \Gamma \vdash M : 0\end{array}}{\Gamma \vdash \mathrm{abort}_{A\times B} M : A \times B}\,0\mathrm{E}}{\Gamma \vdash \pi_1(\mathrm{abort}_{A\times B} M) : A}\,\times\mathrm{E}_1$$

To normalize this proof, we simply have to take $\pi_1$ and apply 0 E, but with the correct type. Thus $\Pi'$ is the following proof, and we have $\pi_1(\mathrm{abort}_{A\times B} M) \to_\beta \mathrm{abort}_A M$.

$$\dfrac{\begin{array}{l}\qquad \vdots\, \pi_1 \\ \Gamma \vdash M : 0\end{array}}{\Gamma \vdash \mathrm{abort}_A M : A}\,0\mathrm{E}$$

Notice that the term $\pi_1(\mathrm{abort}_{A\times B} M)$ has $\beta$-reduced to $\mathrm{abort}_A M$, which exactly correlates to the permutation conversion as defined in Definition 17.

In fact, any $\bot$E/$x$E detour will be normalized by using a similar method, and the corresponding $\beta$-reductions will be as described in Definition 17.

$\square$

Detour formulas, as one might recall, are a specific type of cut segment. In other words, a cut segment is a generalized version of a detour. Perhaps it is similarly possible to generalize Theorem 23? As a matter of fact, we can prove that normalization of proofs containing cut segments is isomorphic to $\beta$-reduction inclusive of Definition 15, Definition 16, and Definition 17.

**Theorem 24** (Isomorphism Between $\beta$-Reduction and Cut Segment Normalization)**.** *If $\Pi$ is a proof of $t : A$ from a set of assumptions $\Gamma$ and contains cut segments, then removing those cut segments results in a proof $\Pi'$ of $t' : A$ from a subset of $\Gamma$ where $t \to_\beta t'$.*

*Proof.* Using induction on the length of cut segments of $\Pi$.

**Base:** Cut segment length is 1. This means that $\Pi$ contains simple detours. But we already proved the claim for detours in Theorem 23.

**Inductive Step:** Cut segment length is $> 1$. If the length of the cut-segment we would like to eliminate is $> 1$, then we simply apply permutation conversions until the length is equal to 1. The case thereby falls under the base case, and we eliminate the detour as described above. We do this for all cut-segments that appear in $\Pi$, the result of which is succesful normalization of $\Pi$ to $\Pi'$.

For example, consider the following $\Pi$ with cut segment length of 2.

$$
\dfrac{
x\,y\;\dfrac{
\begin{matrix} \vdots\,\pi_0 \\ \Gamma \vdash p : A+B \end{matrix}
\qquad
\dfrac{
\begin{matrix} \Gamma, x : A \vdash x : A \\ \vdots\,\pi_1 \\ \Gamma, x : A \vdash t : C \end{matrix}
\qquad
\begin{matrix} \Gamma, x : A \vdash x : A \\ \vdots\,\pi_2 \\ \Gamma, x : A \vdash s : D \end{matrix}
}{\Gamma, x : A \vdash \langle t, s\rangle : C \times D}\times\mathrm{I}
\qquad
\begin{matrix} \Gamma, y : B \vdash y : B \\ \vdots\,\pi_3 \\ \Gamma, y : B \vdash u : C \times D \end{matrix}
}{\Gamma \vdash (\text{case } p \text{ of } (x)\langle t, s\rangle ; (y)u) : C \times D}+\mathrm{E}_4
}{\Gamma \vdash \pi_1(\text{case } p \text{ of } (x)\langle t, s\rangle ; (y)u) : C}\times\mathrm{E}_1
$$

In the example above, $t$ is $\pi_1(\text{case } p \text{ of } (x)\langle t, s\rangle ; (y)u)$. Our first step will be to permute $\times\mathrm{E}_1$ upwards to get the following proof, $\Pi_1$.

$$
x\,y\;\dfrac{\begin{array}{c}\vdots\,\pi_0\\ \Gamma\vdash p:A+B\end{array}\qquad \dfrac{\dfrac{\begin{array}{c}\Gamma,x:A\vdash x:A\\ \vdots\,\pi_1\\ \Gamma,x:A\vdash t:C\end{array}\qquad\begin{array}{c}\Gamma,x:A\vdash x:A\\ \vdots\,\pi_2\\ \Gamma,x:A\vdash s:D\end{array}}{\Gamma,x:A\vdash\langle t,s\rangle:C\times D}\times\mathrm{I}}{\Gamma,x:A\vdash\pi_1\langle t,s\rangle:C}\times\mathrm{E}_1\qquad \dfrac{\begin{array}{c}\Gamma,y:B\vdash y:B\\ \vdots\,\pi_3\\ \Gamma,y:B\vdash u:C\times D\end{array}}{\Gamma,y:B\vdash\pi_1u:C}\times\mathrm{E}_1}{\Gamma\vdash(\text{case }p\text{ of }(x)\pi_1\langle t,s\rangle;(y)\pi_1u):C}+\mathrm{E}_4
$$

Notice that $t$ became $t_1$ in $\Pi_1$. That is, $\pi_1(\text{case } p \text{ of } (x)\langle t,s\rangle;(y)u) \longrightarrow_\beta$ $(\text{case } p \text{ of } (x)\pi_1\langle t,s\rangle;(y)\pi_1 u)$, which means that $t$ $\beta$-reduced to $t_1$ in accordance to the permutation conversion defined in Definition 16. Now that we have a cut segment of length 1, we remove the detour as usual to get $\Pi'$.

$$
x\,y\;\dfrac{\begin{array}{c}\vdots\,\pi_0\\ \Gamma\vdash p:A+B\end{array}\qquad \begin{array}{c}\Gamma,x:A\vdash x:A\\ \vdots\,\pi_1\\ \Gamma,x:A\vdash t:C\end{array}\qquad \dfrac{\begin{array}{c}\Gamma,y:B\vdash y:B\\ \vdots\,\pi_3\\ \Gamma,y:B\vdash u:C\times D\end{array}}{\Gamma,y:B\vdash\pi_1u:C}\times\mathrm{E}_1}{\Gamma\vdash(\text{case }p\text{ of }(x)t;(y)\pi_1u):C}+\mathrm{E}_4
$$

At the very end, we see that $t_1 \longrightarrow_\beta t'$. That is, $(\text{case } p \text{ of } (x)\pi_1\langle t,s\rangle;(y)\pi_1 u) \longrightarrow_\beta$ $(\text{case } p \text{ of } (x)t;(y)\pi_1 u)$ in accordance to Definition 15.

□

**Theorem 25** (Strong Normalization)**.** *Every reduction and conversion sequence for* $\Pi$ *eventually terminates. Moreover, the resultant normal form* $\Pi'$ *is unique.*

*Proof.* See Theorem 3.5.3 in Dag Prawitz. "Ideas and Results in Proof Theory". In: *Studies in Logic and the Foundations of Mathematics.* Vol. 63. Elsevier, 1971, p. 256. See also section 10.5 in Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types.* Cambridge Tracts in Theoretical Computer Science 7. Cambridge: Cambridge University Press, 1989, p. 78. □

## 3.2 Is the Correspondence Isomorphic?

So far, we have used the words 'correspondence' and 'isomorphism' somewhat interchangeably. These notions are not identical, of course, since the notion of an isomorphism is stronger than that of a correspondence. That is, every isomorphism is a correspondence, but not every correspondence is an isomorphism.

In the early days of the correspondence—and rest assured, we do have at least a correspondence—it was mostly referred to as something akin to a correspondence. In fact, in Howard's privately distributed notes—the very same Howard whose name is appended to the name of the correspondence—there are mentions of 'correspondence', but nowhere does he call it 'an isomorphism'.[5] Curry—also the very same Curry—described corresponding terms as *analogues*.[6,7] Christopher Goad who wrote two early influential works describing the computational uses of formal proofs also did not use the word 'isomorphism'.[8,9] He does gesture towards something like an isomorphism when he writes, "For natural deduction proofs of pure implicational logic, the correspondence to the typed $\lambda$-calculus is exact ..."[10] The computer scientist Robert L. Constable, known for developing Nuprl with his research

[5] Howard, *The Formulae-as-Types Notion of Construction*.

[6] Haskell B. Curry. "Some Properties of Equality and Implication in Combinatory Logic". In: *Annals of Mathematics* 35.4 (1934), pp. 849–860. DOI: `10.2307/1968498`. JSTOR: `1968498`, pp. 855–6.

[7] Haskell B. Curry and Robert Feys. *Combinatory Logic*. Studies in Logic and the Foundations of Mathematics. Amsterdam: North-Holland, 1958. URL: `https://catalog.hathitrust.org/Record/001918752`, 312–5, sec. 9E.

[8] Christopher Alan Goad. "Proofs as Descriptions of Computation". In: *5th Conference on Automated Deduction Les Arcs, France, July 8–11, 1980* 87 (1980), pp. 39–52. DOI: `10.1007/3-540-10009-1_4`.

[9] Christopher Alan Goad. "Computational Uses of the Manipulation of Formal Proofs". PhD dissertation. Stanford University, 1980.

[10] Ibid., p. 31.

group on automated programming at Cornell, was familiar with Goad's work. With similar sentiments, he writes "Programs are like constructive proofs of their specifications. This analogy is a precise equivalence for certain classes of programs."[11] It is interesting that although they share intuitions of some kind of limited equivalence, they approach it from opposite perspectives: Goad approaches it from proofs, whereas Constable approaches it from programs.

Later on, and this continues to the present day, some identified proofs with programs, which led to some resistance where the distinction between proofs and programs was reasserted and defended.[12] The phrase "Curry-Howard Isomorphism" then begins to appear in the early 1990s.[13,14] But the appearance of the word "isomorphism" did not settle the matter conclusively. A mere decade later Giuseppe Longo writes "Therefore thanks to the biunivocal correspondence between [lambda] terms and proofs ..."[15] But we know that a bijection is still not yet an isomorphism, so we are left with a question: is it only a correspondence or something more?

The simple answer is that it depends on the systems one uses. As mentioned before, the systems used and developed here ensure we have a full isomorphism. In fact, the bijection illustrated on Page 51 and the proofs for Theorem 23 and Theorem 24 all taken together prove the full isomorphism.

[11] Robert L. Constable. "Programs as Proofs: A Synopsis". In: *Information Processing Letters* 16.3 (1983), pp. 105–112. DOI: 10.1016/0020-0190(83)90060-1, p. 1.

[12] Bengt Nordström. "Terminating General Recursion". In: *BIT* 28.3 (1988), pp. 605–619. DOI: 10.1007/BF01941137, pp. 608–9.

[13] Thierry Boy de la Tour and Christoph Kreitz. "Building Proofs by Analogy via the Curry-Howard Isomorphism". In: *Logic Programming and Automated Reasoning*. Ed. by Andrei Voronkov. Lecture Notes in Computer Science 624. Berlin, Heidelberg: Springer, 1992, pp. 202–213. DOI: 10.1007/BFb0013062.

[14] Parigot, "Classical Proofs as Programs".

[15] Giuseppe Longo. "Proofs and Programs". In: *Synthese* 134.1/2 (2003), pp. 85–117. DOI: 10.1023/A:1022135614184, p. 95.

In contrast, the systems used by Sorensen and Urzyczyn do not amount to a full isomorphism.[16] Using their systems, a single proof can correspond to different $\lambda$-terms. Consider the following derivation, which is a legitimate derivation according to their natural deduction rules.

$$\dfrac{\dfrac{p \vdash p}{p \vdash p \to p}}{\vdash p \to p \to p}$$

According to them, such a derivation poses a problem since it corresponds both to $\lambda x^p(\lambda y^p.x)$ and $\lambda x^p(\lambda y^p.y)$.[17] And it is a genuine problem because to count as an isomorphism, the systems need to be bijective (i.e., one-to-one) and preserve structures and relations among their elements.

The usual problem is that $\lambda$-terms usually 'encode' more information than a usual **NJ** system. Put in other words, simply-typed lambda-terms are more fine-grained than their counterpart proofs in conventional **NJ** systems. But we've already seen reasons why such **NJ** systems are limiting and misguided in their treatment of assumptions and introductions. There are serious philosophical reasons to fortify these systems with further machinery capable of handling usual shortfalls, and as we saw, this machinery ends up being advantageous too. Serendipitously, once **NJ** is fitted with our proposed modifications, we get a system that is fully isomorphic with the extended simply-typed $\lambda$-calculus.

[16] Sørensen and Urzyczyn, op. cit., p. 81.
[17] Ibid., p. 81.

# Chapter 4

# Significance

Now that we have seen the full details of the Curry-Howard isomorphism, in this chapter we may finally consider its significance. Here I mean 'significance' in both senses of the word: meaning and importance. We will first investigate the meaning of expressions such as 'proofs-as-programs', 'a proof's computational content', and 'proofs and programs are two sides of the same coin'. Such expressions are frequently used to refer to the correspondence or describe it, which naturally raises the question of their meaning. The significance of such expressions has yet to be properly explored, and so we devote the first section of this chapter to beginning that task. It is my hope that doing so generates more philosophical interest in the correspondence.

We will then very briefly consider the significance of the correspondence in the second sense of the word. That is, we will consider the correspondence's importance as it relates to its applications present and future. This second and final section will conclude our exploration on significance.

## 4.1 Investigating ‘Proofs-as-Programs’

As mentioned previously, the correspondence is often referred to by a number of slogans. That set of expressions includes phrases like *proofs-as-programs, the proofs-as-programs paradigm, propositions-as-types, the propositions-as-types paradigm, formulae-as-types,* and so on. These expressions are quite striking: the use of ‘as’ in this sense typically marks an analogy indicating likeness, but the two objects likened—proofs and programs—are seemingly so disparate that they resist comparison. This is a hallmark of metaphorical language: “A metaphor makes us attend to some likeness, often a novel or surprising likeness, between two or more things.”[1]

In metaphor making, we connect categories whose edges are not immediate, which is the source of our surprise, awe, or insight. Put differently, the power of a metaphor “derives from how massively and conspicuously different its two subject matters are.”[2] The phrase ‘proofs as programs’, along with its various counterparts, is striking because the word ‘as’ draws a direct edge between proof and programs that we hadn’t noticed or imagined before. In so doing, it sidesteps our ordinary associations and with it, our ordinary picture of reality and the ordinary possibilities contained therein.[3]

Moreover, these slogans do not exhaust the striking language in relation

[1] Donald Davidson. “What Metaphors Mean”. In: *Critical Inquiry* 5.1 (1978), pp. 31–47. JSTOR: 1342976, p. 33.

[2] David Hills. *Metaphor*. The Stanford Encyclopedia of Philosophy. Aut. 2017. URL: https://plato.stanford.edu/archives/fall2017/entries/metaphor/, sec. 1.

[3]Alternatively, rather than sidestep, such comparisons *cut across* the ordinary. Whichever it is—and it could be one or the other depending on the context—I do not claim that these comparisons *transcend* our ordinary categories or relations. Arguably, the surprise and power of a metaphor arises from its direct conflict with our ordinary categories. But conflicts would be impossible if such comparisons somehow go completely beyond the fore, as would be the case with transcending our ordinary frame of relations.

to the correspondence. The literature on the Curry-Howard isomorphism is replete with claims identifying proofs with programs and, even more curiously, claims of identity but *only to a degree.* The list below contains a small selection of such claims.

1. "[P]roofs and programs are more or less the same objects . . . "[4]
2. "Proof theory and the theory of computation turn out to be two sides of the same field."[5]
3. "Thus, [. . . ] it was finally possible to see the work of Gentzen and Church as two sides of the same coin."[6]
4. "[P]roofs and programs are really the same thing . . . "[7]
5. "Propositions as Types is a notion with mystery. Why should it be the case that intuitionistic natural deduction . . . and simply typed lambda calculus . . . should be discovered 30 years later to be essentially identical?"[8]

In a longer description, Martin-Löf writes

> The difference, then, between constructive mathematics and programming does not concern the primitive notions of the one or the other, because they are essentially the same, but lies in the

[4] Sørensen and Urzyczyn, op. cit., p. vi.
[5] Ibid., p. v.
[6] Philip Wadler. *Proofs Are Programs: 19th Century Logic and 21st Century Computing.* Avaya Labs, 2000, pp. 1–15. URL: `https://homepages.inf.ed.ac.uk/wadler/papers/frege/frege.pdf`, p. 10.
[7] Ibid., p. 1.
[8] Idem, "Propositions as Types", p. 2.

> programmer's insistence that his programs be written in a formal notation so that they can be read and executed by a machine, whereas, in constructive mathematics [. . . ] the computational procedures (programs) are normally left implicit in the proofs, so that considerable further work is needed to bring them into a form which makes them fit for mechanical execution.[9]

Moreover, from the very early days, proofs were described to have *computational content*. For instance, Constable writes "We can see the computational content of the proof rather clearly in terms of recursive procedures."[10] A little earlier, Goad writes "In the former, formal proofs serve as vessels from which computational contents of a standard kind are extracted."[11] But in this context, what exactly do the words 'contain' and 'contents' mean? The meaning of such language has never been made very clear.[12]

The question I intend to ask here is the same question but broadly put: *just what does such language really mean?* Or, perhaps a better question, just what exactly is this language *meant to do*?

To start with, and this should come as no surprise, I argue that such language is not to be taken literally.[13] I believe that slogans like 'proofs-as

[9] Per Martin-Löf. "Constructive Mathematics and Computer Programming". In: (1982), pp. 153–175, p. 156.

[10] Constable, op. cit., p. 106.

[11] Goad, "Proofs as Descriptions of Computation", p. 51.

[12] Richard Zach. "The Significance of the Curry-Howard Isomorphism". In: *Philosophy of Logic and Mathematics. Proceedings of the 41st International Ludwig Wittgenstein Symposium.* Ed. by Gabriele M. Mras, Paul Weingartner, and Bernhard Ritter. Publications of the Austrian Ludwig Wittgenstein Society, New Series 26. Berlin: De Gruyter, 2019, pp. 313–325. DOI: 10.1515/9783110657883-018, p. 315.

[13]Now this is not to say that the meaning of the words involved are not the same as the literal meanings. Davidson may be right that the meaning of a metaphor does not go beyond the literal. I am claiming only that they should not be *understood* literally, or rather, they should not be understood as ordinary, declarative sentences. There is a certain kind of intent—a certain kind of intended emotive or special cognitive effect—involved with

programs' are kinds of similes, identity claims like 'proofs and programs are more or less the same objects ...' are kinds of metaphors, and talk of 'computational contents' is really a stand-in for talk of meaning. In other words, the literature on the correspondence uses figurative language for the purposes of theory. Now this is not really novel: scientific theory regularly invokes figurative language for various reasons.[14] However, the use of figurative language in the context of the correspondence has rarely been recognized, and so the application of philosophical theories of metaphor here is novel.

At this point, it's worth clarifying why I take the language under question *figuratively* and not literally. I think we can reject the view that such language is ordinary (as opposed to metaphorical) rather easily. Let's take each type in turn, starting from the notion of 'a proof's computational content'. When considering an **NJ** proof, exactly where are those purported contents? If those contents are on the page, where on the page are they? Surely not in the written token of the proof because the written token does not contain anything beyond its subparts. To make this clear, consider the word 'and'. The word 'and', which to most humans in all of history is nothing more than some strange strokes, contains, 'a', 'n', 'd', and various subparts of these. It also contains 'an' as well as 'and', but notice that it does not contain anything beyond its subparts.

One might object with *the word 'and' also contains a vowel*, and this wouldn't be entirely wrong. But precisely where is the vowel? If by 'vowel'

metaphorical language that is not had with ordinary sentences.

[14] Jennifer Runke. "Towards an Adequate Theory of Scientific Metaphor". PhD dissertation. Calgary: University of Calgary, 2008. URL: http://www.proquest.com/pqdtlocal1006267/docview/304693593/abstract/A318288088C4917PQ/30, p. 1.

we mean *vowel sound*, then the vowel is located nowhere amongst the set of strokes ‘and’ because there are no sounds in it. Alternatively, if by ‘vowel’ we mean *letter representing a vowel sound*, then the vowel contained is ‘a’, and that was already noted to be a contained subpart. This reply applies to not only vowels, but also to consonants, letters, strokes, and so on. No matter what other thing one purports to be contained, when said thing is analysed enough, it becomes clear that it is either not there at all or it is just another descriptor for a subpart.

Now if the purported content of a proof can’t be on the page (or screen, sand, etc.), then the next guess would be that those contents are cognitive. In which case, however, we are still left with the question of *where in a cognitive proof object are the computational contents*? We cannot reasonably claim that computation is conceptually contained by the concept of proof (or contained within a cognitive proof object) since we know that the notion of computation is not analytic to proof.[15] After all, generations of mathematicians and logicians made proofs before the development of computation. Moreover, there were at least three decades of natural deduction and sequent calculi before the correspondence between proof and computation was widely recognized owing to Howard’s work.

And it’s worth adding that I think analyticity is the kind of thing intended when claiming that there are computational contents of a proof.[16] Firstly, the way of employing the word ‘content’ brings to mind notions such as *semantic content*, *conceptual content*, and *propositional content*, which are

[15]In other words, a computation is not inherently part of the meaning of proof.

[16]That is, I believe that when someone says ‘there are computational contents of a proof’, they are trying to appeal to a notion like analyticity. This is because the claim of computational content seems to me like a claim of computation forming part of the meaning of a proof.

all related. Secondly, I think that early dispensers of 'computational content' already inadvertently reveal that analyticity and semantic content are what they intend. For example, Constable writes "We will see shortly that it makes sense to treat constructive proofs computationally because the *meaning* of a constructive proof is computational."[17]

Thus, I believe that the considerations so far give firm grounds for the idea that 'computational content' means something like semantic or conceptual content. But as we have seen, there are good reasons to suspect that computation is not analytic to proof, and if so, it would be wrong to say that computation is semantically or conceptually contained by proof as a notion (or proof as a cognitive object).

Despite the fact that 'computational contents' strictly means something false, there is another way to think of the relationship between proof and computation. Rather than think of computation as part of proof, we can regard computation as a frame to view proofs by or a perspective adjacent to that of a proof. In other words, the notion of a computation is something superimposed upon or directly adjacent to a proof. The use of metaphorical language then is particularly apt as it is often described as a superimposition of images and concepts upon one another or even placing them adjacently.[18] Then again, maybe the use of figurative language is not merely apt: perhaps metaphors and the like are used *because* of their aptness.[19] This is a thought we shall revisit later.

[17] Constable, op. cit., 107, emphasis added.

[18]For an example of a metaphor where images are placed adjacently, consider the following: "The apparition of these faces in the crowd;/Petals on a wet, black bough." Hills, op. cit., sec. 1, quoting Ezra Pound, *In a Station of The Metro*.

[19]That is, metaphors don't just happen to fit, but rather the metaphors appear because they fit.

Returning to our reductio, we can reject the simple identification of proofs with computation—as seen in 4 and 5—out of hand. Proofs and computation have different origins, have different literatures, are developed differently, are used differently, etc. There may be some important overlaps between them, as is suggested by the correspondence, but we should not ignore the many differences that nevertheless distinguish them.

As for identifications like 1, 2, and 3, these seem more plausible, however we must consider the claims carefully still. I believe that the following puzzle arises when we examine them carefully: If proofs and programs are the same object, then *what exactly is that object?* Likewise, if they are two sides of the same field (or coin), then *what is that field? What is that coin?*

Suppose the 'object' are proofs.[20] We would be treating proofs as the more basic, fundamental object that somehow contains programs. But we already saw why we should suspect this exact claim: proofs cannot contain programs in any legitimate sense. The same reasoning developed earlier applies in the opposite direction as well. A program does not *contain* a proof, though they can help us find them or verify them. Moreover, the very fact that we have to design **NJ** and $\lambda^{\rightarrow}$ *precisely just so* problematizes any claim that they are identical to one another or are different sides of the same object. My intuition is that they cannot be the same (or have an identical object in common) if we must push proofs to make them more like programs and pull programs to make them more like proofs. Yet this is precisely what we do when we require **NJ** to carry a label for every assumption or extend $\lambda^{\rightarrow}$ as we did earlier. Now those changes were not arbitrary and unjustifiable, but we

[20]Here and in the following, we use the word 'object'. However, we can also substitute 'field' or 'coin' for 'object' without affecting the argument.

do still have to recognize them for what they are: adjustments undertaken so as to develop a correspondence into a full isomorphism.

So neither proofs nor programs can be the mysterious object in common. Perhaps then the object is a hybrid *proof-program*? That could be a way of *naming* the perspective(s) revealed by the isomorphism, but there is no such proof-program in reality. For one thing, it is not a very well-defined concept, and moreover, it lacks qualities that proofs and programs separately have that assures that they are well-defined and 'exist' in some sense. After all, if there were such an object as a proof-program in the way that a proof or a program is an object, what is its history? Where is its literature? Who has worked on its developments? If in reply, we say *why, of course, that history would be the combined history of proofs and programs, that literature the combined literature, etc.*, we would simply beg the question.

Thus the object is neither a proof nor a program, and it cannot be a hybrid either. Our last resort is to think of the object as something underlying both a proof and a program, but which cannot be described in further terms. This last option leaves as much mystery as leaving the question mostly unanswered. We still do not know what the object is. Is it pure thought itself hiding behind the veil of language? An amorphous object we can't directly access, but whose very existence makes possible the separate existence of proofs and programs, which are accessible to us?

Because 1, 2, and 3 all make direct reference to some kind of object, I would prefer a clearer answer and suppose the claim that there is an amorphous, underlying object false. Evidently, however, this has no real bearing on the truth of the matter. Notice though that if it's true and suppose it so, nothing more can be reasonably said: we would be pushed to help-less silence. If instead we continue and suppose it's false, and it turns out that *it's false that* it's false, then we need only to ignore what was said as a consequence of our assumption. In other words, there is little harm in continuing our current investigation because we cannot expand upon or develop further theory if the claim I assume is false turns out to be true.

In sum, we likewise find the identification between proofs and programs to be false.[21] If we understand such identifications literally and as ordinary assertions, it is strange that something false continues to reappear in serious literature. But again, I contend that all of 1–5 are metaphors, so the apparent falsity is exactly what we would expect should my view be correct. For "it is only when a sentence is taken to be false that we accept it as a metaphor and start to hunt out the hidden implication."[22]

That is, the apparent falsity of the claims is not so important for us who are the recipients of such language. The choices made in language are often determined by speaker's intent. In ordinary language, that intent usually *can* be fulfilled by meaning and context.[23] However in metaphor making, literal meaning often belies speaker intent. Thus metaphor making "concerns not the meaning of words but their use."[24]

But if metaphor is used to deflect our attention from literal meaning, to where is our attention being redirected?[25] I believe that for metaphors like 'a

[21]Or, at the very least, we find them ostensibly false.
[22] Davidson, op. cit., p. 42.
[23]A speaker will use particular language if they judge that its meaning and context *can*—and predictably will—fulfill their intent. Whether or not their choice in language actually fulfills their intent is a separate issue.
[24] Davidson, loc. cit.
[25]I believe the use of 'deflect' here is most apt. The brief, instantaneous meeting with a metaphor's literal meaning propels us elsewhere. Speaking figuratively, our attention is like light and metaphor the carving of a language mirror. Mirrors deflect and divert light, and as we know, light can change an entire appearance.

proof is a program' we are directed to seeing a proof *as* a program (and vice versa when the isomorphism is complete). In other words, having demonstrated the isomorphism, we do not *see that* we have a proof-program, rather the result is that we see a proof *as* a program. Drawing from Wittgenstein, Davidson writes "Seeing as is not seeing that. Metaphor makes us see one thing as another by making some literal statement that inspires or prompts the insight."[26] Here the insight is that a proof can be seen as a program, which is the exact same insight that the isomorphism develops out of. Differently put, the isomorphism is a discovery of a new *perspective* on the familiar proof and the familiar program. Metaphors like 'proofs and programs are two sides of the same coin' precisely capture this shift in perspective.

Like Wittgenstein's seeing-as, the change to seeing a proof as a program "is not part of perception," which makes the change both like and unlike seeing.[27] It is like seeing in that the change causes us to see or interpret proofs anew, but it is unlike seeing in that we do not actually see a proof-program. Moreover, attending to this shift in perspective is "half visual experience, half thought."[28] Similarly, the perspective revealed by the correspondence can be illustrated, but that is only half of it. To truly *see* the new perspective, we must ourselves be able to attend to the change.

This topic brings me to the slogan 'proofs-as programs' and other such slogans that are simile-like. I believe such slogans are short form for 'proofs as kinds of programs', 'proofs taken as programs', or something of that

[26] Davidson, op. cit., p. 47.

[27] Ludwig Wittgenstein. *Philosophical investigations*. Ed. by Joachim Schulte and P. M. S. Hacker. Trans. by G. E. M. Anscombe, P. M. S. Hacker, and Joachim Schulte. Rev. 4th ed. Chichester, West Sussex: Wiley-Blackwell, 2009, sec. 137.

[28] Ibid., p. 140.

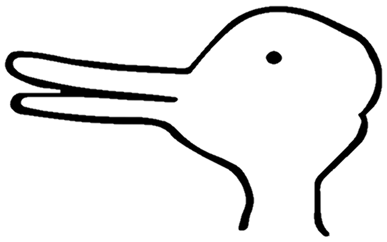

Figure 4.1: Duck-Rabbit

nature.[29] I think when written long form, it becomes quite evident that we are seeing-as, rather than seeing that. The expression ‘proofs taken as programs’ seems to me both a call and invitation to adopt a different perspective. Much like how the label ‘duck-rabbit’ guides us to adopt a new perspective and attend to the change in Figure 4.1, the long form expression ‘proofs taken as programs’ also guides us in this way. So by using ‘proofs-as-programs’, we are made aware that between a program and a proof, there is a startling likeness—a congruence—even in their difference.

In many ways, ‘proofs-as-programs’ is similar to the duck-rabbit figure.[30] For one thing, both duck-rabbit and ‘proofs-as-programs’ invite us to see-as. Furthermore, both *require* seeing-as to understand the full picture: if you only see-that, one risks missing the rabbit and, likewise, the programs. And if one fails to see how a program seems to appear in a proof, then one likely misses the entire correspondence.

When we take an arbitrary program such as the one above, we are also met with something that structurally mimics a similar effect to that of duck-rabbit. In duck-rabbit, the two perspectives are contained by the exact same

[29]And we can act as if proofs were kinds of programs because proofs, in certain respects, *are* like programs. So the structure of the simile follows the structure of the objects at hand.

[30]Henceforth, I will use just ‘duck-rabbit’ to refer to the figure.

$$x\,y\;\dfrac{\begin{matrix}\vdots\,\pi_0\\ \Gamma\vdash\ t:A+B\end{matrix}\qquad\begin{matrix}\Gamma,x:A\vdash\ x:A\\ \vdots\,\pi_1\\ \Gamma,x:A\vdash\ s:C\end{matrix}\qquad\dfrac{\begin{matrix}\Gamma,y:B\vdash\ y:B\\ \vdots\,\pi_3\\ \Gamma,y:B\vdash\ u:C\times D\end{matrix}}{\Gamma,y:B\vdash\ \pi_1 u:C}\times\mathrm{E}_1}{\Gamma\vdash(\text{case }t\text{ of}(x)s;(y)\pi_1 u):C}+\mathrm{E}_4$$

Figure 4.2: A Program

lines, which is to say that they are congruent. If we compare Figure 4.2 with Figure 4.3, we can witness a similar effect where by focusing on some aspects and ignoring others, we begin to see a proof instead.[31]

$$x\,y\;\dfrac{\begin{matrix}\vdots\,\pi_0\\ \Gamma\vdash\ t:\ A+B\end{matrix}\qquad\begin{matrix}\Gamma,x:A\vdash\ x:A\\ \vdots\,\pi_1\\ \Gamma,x:A\vdash\ s:C\end{matrix}\qquad\dfrac{\begin{matrix}\Gamma,y:B\vdash\ y:B\\ \vdots\,\pi_3\\ \Gamma,y:B\vdash\ u:\ C\times D\end{matrix}}{\Gamma,y:B\vdash\ \pi_1 u:\ C}\times\mathrm{E}_1}{\Gamma\vdash(\text{case }t\text{ of}(x)s;(y)\pi_1 u):\ C}+\mathrm{E}_4$$

Figure 4.3: Seeing "Proof-Program"

The rough congruence illustrated above is likely a factor in the origin of the expressions that identify proofs and programs.[32] Again, those expressions of identification spur us to see-as, and this seeing-as captures the core insight of the correspondence. Now the incorrect lesson to draw from the congruence is that there *actually is* a proof-program or that proofs *actually are* programs. After all, having seen duck-rabbit, this would be as mistaken as thinking that there *actually is* a duck-rabbit or that ducks *actually are* rabbits.[33]

[31]Remember to replace $+$ with $\vee$ and $\times$ with $\wedge$.

[32]Of course, the rough congruence could have also developed from the opposite direction (i.e., start from a proof and add a few terms to make a program).

[33]Other similarities between duck-rabbit and 'proofs-as-programs' is the contrived,

Some kind of congruence seems to be a prior requisite to seeing-as. That is, lacking any kind of overlap—whether visual, structural, conceptual, emotive, etc.—the possibilities of seeing-as are vanishingly small. It is this observation that points to another feature of metaphor, which is that metaphors involve objects that are embedded in systems of 'isomorphic structure'.[34] And so it is with proofs and programs: $\wedge$ becomes $\times$, $\rightarrow$ I becomes abs, assumption labels become typing judgments on the left, propositions become types, and so on.

Thus I have the following to say in conclusion: the literature surrounding the Curry-Howard isomorphism is full of metaphors and figurative language in general. We cannot properly explain the presence of such language if we take them as ordinary assertions. However, the moment we recognize the metaphors, we are somewhat freed from the question of their meaning and are pushed to consider their use. It is here we strike upon sudden harmony for such metaphors guide our seeing-as, and this seeing-as makes us attend to the rough pre-existing likeness in the structure of proofs and programs. In a phrase, the metaphors of this literature *encapsulate* the correspondence. In another phrase, the structure of metaphor is a mirror to the structure of correspondence. The last point explains why the *isomorphism's* literature is replete with them and illuminates the question of their use in this particular context.

artificial character of the congruence as well as the non-simultaneous appearance of impressions. To expand on the first, we have to draw the rabbit and duck *precisely just so* to create the congruence. Likewise, for the full isomorphism to hold, we also have to develop **NJ** and $\lambda^{\rightarrow}$ *precisely just so*. Moreover, just as "an impression is not simultaneously of a picture-duck and a picture-rabbit," we never simultaneously have a proof and a program. See Wittgenstein, op. cit., sec. 157. Though these other similarities are interesting, they are not incredibly important for our present purposes.

[34] Max Black. "More about Metaphor". In: *Dialectica* 31.3/4 (1977), pp. 431–457. JSTOR: 42969757, pp. 444–445.

To expediently express or intimate the correspondence between proofs and programs without directly saying so is something that can be done in metaphor, but cannot be done in ordinary, plain language. The very fact that they can encapsulate isomorphic structure with such expedience is itself something of wonder just as the happenstance correspondence between proofs and programs is of wonder. We see then that metaphors become the most apt means of describing the correspondence, and such an observation aligns with their wide usage. Here then, lie the cognitive use, necessity, and ineliminability of metaphors with respect to the Curry-Howard correspondence.

## 4.2 Type-Safety and Proof Assistants

Having said all that, there are of course differences between duck-rabbit and the correspondence referred to by the abbreviation 'proofs-as-programs'. One significant difference is that the correspondence is good for *a great deal* more than duck-rabbit, whose only claims to fame are its illusion and its appearance in Wittgenstein's *Philosophical Investigations*. In contrast, the Curry-Howard correspondence underlies a number of innovations in computer science and (formal) mathematics. So the correspondence is not merely a recognition of striking likeness. Nor is it only theoretically significant for the reason that it allows us to prove strong normalization.[35] It turns out that the correspondence also has powerful practical applications, which must be acknowledged if we are to properly understand its significance.

Within the overlaps between computer science and programming, the

[35] Zach, op. cit., pp. 315, 321.

correspondence provides the ability to type-check a program.[36] Essentially, the correspondence simplifies the problem of determining a program's type, because all a computer has to do is find a corresponding derivation of that type in normal form.[37] The properties of normal form derivations further ensure that this search can effectively be done.[38] The ability to type-check is particularly important because it allows a computer to check for type-errors, often during compile time before the program is even run.[39]

Relatedly, then, the correspondence can also provide a kind of programming safety. In particular, the correspondence can be used to guarantee *type safety*. Type safety is a property of programming languages, and a type-safe language is one that is secure from type errors.[40] One important feature of a type-safe language is that programs will always execute until the result is obtained, and this is a feature that comes out of the correspondence.[41] By limiting (or otherwise preventing) the possibility of writing a program that commits type errors, it is said that such languages are "easier to read, easier to maintain, easier to debug, and they are, demonstrably, more secure."[42] These are obvious gains for any programming language, and here, I emphasize again that such gains depend upon the correspondence. Given the importance of type safety, as well as its associated advantages, it remains and will remain a vital property for programming languages now and in the

[36] Ibid., p. 322.
[37] Ibid., p. 322.
[38] Ibid., p. 322.
[39] A type-error is a error that arises when there is a mismatch of data types. For instance, if I have defined a program that will evaluate the sum of two integers, a type error would occur if I later use that program to work on sets of strings. After all, an integer is not a string (and is not a set of strings either), so we should not even be allowed to use the program in that way.
[40] Zach, loc. cit.
[41] In programming, they say that such languages can never *hang*. See ibid., p. 322.
[42] Ibid., p. 324.

foreseeable future.

Within the overlaps between computer science and mathematics, Curry-Howard correspondences established with more powerful systems—like Coquand and Huet's Calculus of Constructions or Martin-Löf's intuitionistic type theory—are the bases for proof assistants, which are software that allow computers to manage and work with formal proofs. Specifically, proof assistants are used to find formal proofs either automatically (i.e., automated theorem proving) or through interactions with human mathematicians who guide the computer's proof search by applying tactics to resolve a series of goals (i.e., interactive theorem proving).[43]

In fact, that last application overlaps with *formal verification*, another significant use of proof assistants.[44] And this is because interactive theorem proving works by entering "enough information for the system to confirm that there is a formal axiomatic proof of the theorem that the user has asserted."[45] Formal verification, however, is not exhausted by verifying the correctness of our mathematical proofs: it is also used in the verification of computer hardware and software (i.e., verifying meeting specifications), and this particular application is used for tasks as crucial as verifying the control systems of planes or cars.[46] And how exactly can proof assistants do all of this? Similar to what we found in programming, these various uses of proof assistants also depend upon type checking.[47] And as was already

[43]One can alternatively think of 'proof search' as proof construction.

[44] Jeremy Avigad. "Understanding, Formal Verification, and the Philosophy of Mathematics". In: *Journal of the Indian Council of Philosophical Research* 27 (2010), pp. 161–197, p. 6.

[45] Ibid., p. 6.

[46] Jeremy Avigad. *Mathematics and Language*. 2015. arXiv: `1505.07238 [math]`. URL: `http://arxiv.org/abs/1505.07238`, p. 13.

[47] Geuvers, "Proof Assistants", p. 9.

mentioned before, the Curry-Howard correspondence is largely responsible for a computer's ability to type check.

In closing, the promise of proof assistants is wide and multifarious. From the dream of formalizing all of mathematics, to making proof verification simple for academic peer review, to managing vast digital libraries of mathematical knowledge, to verifying proofs that are too complex or long for human mathematicians, to creating interactive mathematical exercises for students, and to verifying proofs on exam papers, proof assistants are billed to do all this and more.[48] It is again a wonderful, curious thing that the Curry-Howard correspondence has a fundamental role to play now and in such things that are still yet to pass.

[48] Ibid., pp. 14–21.

# Bibliography

Avigad, Jeremy. *Mathematics and Language.* 2015. arXiv: 1505.07238 [math]. URL: http://arxiv.org/abs/1505.07238.

— "Understanding, Formal Verification, and the Philosophy of Mathematics". In: *Journal of the Indian Council of Philosophical Research* 27 (2010), pp. 161–197.

Avigad, Jeremy and Richard Zach. *The Epsilon Calculus.* The Stanford Encyclopedia of Philosophy. Aut. 2020. URL: https://plato.stanford.edu/archives/fall2020/entries/epsilon-calculus/.

Avron, Arnon. "Whither Relevance Logic?" In: *Journal of Philosophical Logic* 21.3 (1992), pp. 243–281. DOI: 10.1007/BF00260930.

Barendregt, Henk and Erik Barendsen. *Introduction to the Lambda Calculus.* 1998, pp. 1–55. URL: https://www.researchgate.net/publication/215458960_Introduction_to_lambda_calculus.

Black, Max. "More about Metaphor". In: *Dialectica* 31.3/4 (1977), pp. 431–457. JSTOR: 42969757.

Cardone, Felice and J. Roger Hindley. *History of Lambda-Calculus and Combinatory Logic.* MRRS-05-06. Swansea University Mathematics Department, 2006, pp. ii+1–93. URL: https://www.researchgate.net/

publication/228386842_History_of_lambda-calculus_and_combinatory_logic.

Constable, Robert L. "Programs as Proofs: A Synopsis". In: *Information Processing Letters* 16.3 (1983), pp. 105–112. DOI: 10.1016/0020-0190(83)90060-1.

Curry, Haskell B. "Functionality in Combinatory Logic". In: *Proceedings of the National Academy of Sciences* 20.11 (1934), pp. 584–590. DOI: 10.1073/pnas.20.11.584.

— "Some Properties of Equality and Implication in Combinatory Logic". In: *Annals of Mathematics* 35.4 (1934), pp. 849–860. DOI: 10.2307/1968498. JSTOR: 1968498.

Curry, Haskell B. and Robert Feys. *Combinatory Logic*. Studies in Logic and the Foundations of Mathematics. Amsterdam: North-Holland, 1958. URL: https://catalog.hathitrust.org/Record/001918752.

Davidson, Donald. "What Metaphors Mean". In: *Critical Inquiry* 5.1 (1978), pp. 31–47. JSTOR: 1342976.

De la Tour, Thierry Boy and Christoph Kreitz. "Building Proofs by Analogy via the Curry-Howard Isomorphism". In: *Logic Programming and Automated Reasoning*. Ed. by Andrei Voronkov. Lecture Notes in Computer Science 624. Berlin, Heidelberg: Springer, 1992, pp. 202–213. DOI: 10.1007/BFb0013062.

Geuvers, Herman. "Proof Assistants: History, Ideas and Future". In: *Sadhana* 34.1 (2009), pp. 3–25. DOI: 10.1007/s12046-009-0001-5.

Girard, Jean-Yves, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge: Cambridge University Press, 1989.

Goad, Christopher Alan. “Computational Uses of the Manipulation of Formal Proofs”. PhD dissertation. Stanford University, 1980.

— “Proofs as Descriptions of Computation”. In: *5th Conference on Automated Deduction Les Arcs, France, July 8–11, 1980* 87 (1980), pp. 39–52. DOI: 10.1007/3-540-10009-1_4.

Hartnett, Kevin. *Building the Mathematical Library of the Future*. Quanta Magazine. URL: https://www.quantamagazine.org/building-the-mathematical-library-of-the-future-20201001/.

Hills, David. *Metaphor*. The Stanford Encyclopedia of Philosophy. Aut. 2017. URL: https://plato.stanford.edu/archives/fall2017/entries/metaphor/.

Hindley, J. Roger. *Lambda-Calculus and Combinators: An Introduction*. In collab. with Jonathan P. Seldin. Cambridge: Cambridge University Press, 2008.

Howard, William A. *The Formulae-as-Types Notion of Construction*. Chicago: Department of Mathematics, University of Illinois, 1969, pp. 1–12.

Howard, William Alvin. “The Formulae-as-Types Notion of Construction”. In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Ed. by Jonathan P. Seldin and J. Roger Hindley. London and New York: Academic Press, 1980, pp. 480–490.

Jaśkowski, Stanisław. “On the Rules of Suppositions in Formal Logic”. In: McCall, Storrs. *Polish Logic 1920–1939*. Oxford: Oxford University Press, 1967, pp. 232–258.

Longo, Giuseppe. “Proofs and Programs”. In: *Synthese* 134.1/2 (2003), pp. 85–117. DOI: 10.1023/A:1022135614184.

Mancosu, Paolo. *The Adventure of Reason: Interplay between Philosophy of Mathematics and Mathematical Logic, 1900-1940*. Oxford: Oxford University Press, 2014.

Mancosu, Paolo, Richard Zach, and Sergio Galvan. *An Introduction to Proof Theory: Normalization, Cut-Elimination, and Consistency Proofs*. Oxford, New York: Oxford University Press, 2021.

Martin-Löf, Per. "Constructive Mathematics and Computer Programming". In: (1982), pp. 153–175.

Nordström, Bengt. "Terminating General Recursion". In: *BIT* 28.3 (1988), pp. 605–619. DOI: 10.1007/BF01941137.

Nordström, Bengt and Jan Smith. "Propositions and Specifications of Programs in Martin-Löf's Type Theory". In: *BIT* 24.3 (1984), pp. 288–301. DOI: 10.1007/BF02136027.

Parigot, Michel. "Classical Proofs as Programs". In: *Computational Logic and Proof Theory*. KGC 1993. Ed. by Georg Gottlob, Alexander Leitsch, and Daniele Mundici. Vol. 713. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer-Verlag, 1993, pp. 263–276. DOI: 10.1007/BFb0022575.

Pelletier, Francis Jeffry. "A Brief History of Natural Deduction". In: *History and Philosophy of Logic* 20.1 (1999), pp. 1–31. DOI: 10.1080/014453499298165.

Pierce, Benjamin C. *Types and Programming Languages*. Cambridge, Mass: MIT Press, 2002.

Prawitz, Dag. "Ideas and Results in Proof Theory". In: *Studies in Logic and the Foundations of Mathematics*. Vol. 63. Elsevier, 1971, pp. 235–307. DOI: 10.1016/S0049-237X(08)70849-8.

Rathjen, Michael and Wilfried Sieg. *Proof Theory*. The Stanford Encyclopedia of Philosophy. Aut. 2020. URL: https://plato.stanford.edu/archives/fall2020/entries/proof-theory/.

Rojas, Raul. *A Tutorial Introduction to the Lambda Calculus*. 2015. arXiv: 1503.09060 [cs]. URL: http://arxiv.org/abs/1503.09060.

Runke, Jennifer. "Towards an Adequate Theory of Scientific Metaphor". PhD dissertation. Calgary: University of Calgary, 2008. URL: http://www.proquest.com/pqdtlocal1006267/docview/304693593/abstract/A318288088C4917PQ/30.

Seldin, Jonathan P. *The Logic of Curry and Church*. Department of Mathematics and Statistics, University of Lethbridge, 2008, pp. 1–77. URL: https://people.uleth.ca/~jonathan.seldin/CCL.pdf.

Selinger, Peter. *Lecture Notes on the Lambda Calculus*. Halifax: Department of Mathematics, Dalhousie University, pp. 1–120. URL: https://arxiv.org/abs/0804.3434.

Shapiro, Stewart and Peter King. "The History of Logic". In: *The Oxford Companion to Philosophy*. Ed. by Ted Honderich. Oxford University Press, 1995, pp. 496–500.

Sørensen, Morten Heine and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. 1st ed. Amsterdam: Elsevier, 2006.

Wadler, Philip. *Proofs Are Programs: 19th Century Logic and 21st Century Computing*. Avaya Labs, 2000, pp. 1–15. URL: https://homepages.inf.ed.ac.uk/wadler/papers/frege/frege.pdf.

— "Propositions as Types". In: *Communications of the ACM* 58.12 (2015), pp. 75–84. DOI: 10.1145/2699407.

Wittgenstein, Ludwig. *Philosophical investigations*. Ed. by Joachim Schulte and P. M. S. Hacker. Trans. by G. E. M. Anscombe, P. M. S. Hacker, and Joachim Schulte. Rev. 4th ed. Chichester, West Sussex: Wiley-Blackwell, 2009.

Zach, Richard. “The Significance of the Curry-Howard Isomorphism”. In: *Philosophy of Logic and Mathematics. Proceedings of the 41st International Ludwig Wittgenstein Symposium*. Ed. by Gabriele M. Mras, Paul Weingartner, and Bernhard Ritter. Publications of the Austrian Ludwig Wittgenstein Society, New Series 26. Berlin: De Gruyter, 2019, pp. 313–325. DOI: 10.1515/9783110657883-018.

Zalta, Edward N. *Gottlob Frege*. The Stanford Encyclopedia of Philosophy. Aut. 2020. URL: https://plato.stanford.edu/archives/fall2020/entries/frege/.