Propositions as Types *

Philip Wadler
University of Edinburgh
wadler@inf.ed.ac.uk

1. Introduction

Powerful insights arise from linking two fields of study previously thought separate. Examples include Descartes's coordinates, which links geometry to algebra, Plank's Quantum Theory, which links particles to waves, and Shannon's Information Theory, which links thermodynamics to communication. Such a synthesis is offered by the principle of Propositions as Types, which links logic to computation. At first sight it appears to be a simple coincidence—almost a pun—but it turns out to be remarkably robust, inspiring the design of theorem provers and programming languages, and continuing to influence the forefronts of computing.

Propositions as Types is a notion with many names and many origins. It is closely related to the BHK Interpretation, a view of logic developed by the intuitionists Brouwer, Heyting, and Kolmogorov in the 1930s. It is often referred to as the Curry-Howard Isomorphism, referring to a correspondence observed by Curry in 1958 and refined by Howard in 1969 (though not published until 1980, in a Festschrift dedicated to Curry). Others draw attention to significant contributions from de Bruijn's Automath and Martin-Löf's Type Theory in the 1970s. Many variant names are found in the literature, including Formulae as Types, Curry-Howard-de Bruijn Correspondence, Brouwer's Dictum, and others.

Propositions as Types is a notion with depth. It describes a correspondence between a given logic and a given programming language, for instance, between Gentzen's intuitionistic natural deduction and Church's simply-typed lambda calculus. At the surface, it says that for each proposition in the logic there is a corresponding type in the programming language—and vice versa. Thus we have

propositions as *types*.

But it goes deeper, in that for each proof of a given proposition, there is a program of the corresponding type—and vice versa. Thus we also have

proofs as *programs*.

And it goes deeper still, in that for each way to normalise a proof there is a corresponding way to evaluate a program—and vice versa. Thus we further have

normalisation of proofs as *evaluation of programs*.

Hence, we have not merely a shallow bijection between propositions and types, but a true isomorphism preserving the deep structure of proofs and programs, normalisation and evaluation.

Propositions as Types is a notion with breadth. It applies to a range of logics including propositional, predicate, second-order, intuitionistic, classical, modal, and linear. It underpins the foundations of functional programming, explaining features including functions, products, sums, parametric polymorphism, data abstraction, continuations, linear types, and session types. It has inspired theorem provers and programming languages including Agda, Automath, Coq, Epigram, $F^{\#}$, $F^{\star}$, Haskell, LF, ML, NuPRL, Scala, Singularity, and Trellys. Applications include CompCert, a certified compiler for the C programming language verified in Coq, a computer-checked proof of the four-colour theorem also verified in Coq, parts of the Ensemble distributed system verified in NuPRL, and ten thousand lines of browser plug-ins verified in $F^{\star}$.

Propositions as Types is a notion with mystery. Why should it be the case that intuitionistic natural deduction, as developed by Gentzen in the 1930s, and simply-typed lambda calculus, as developed by Church around the same time for an unrelated purpose, should be discovered forty years later to be essentially identical? And why should it be the case that the same correspondence arises again and again? The logician Girard and the computer scientist Reynolds independently developed the same calculus, now dubbed Girard-Reynolds. The logician Hindley and the computer scientist Milner independently developed the same type system, now dubbed Hindley-Milner. Curry-Howard is a double-barrelled name that ensures the existence of other double-barrelled names. Those of us that design and use programming languages may often feel they are arbitrary, but Propositions as Types assures us some aspects of programming are absolute.

2. Church, and the origins of computation

The origins of logic lie with Aristotle and the stoics in classical Greece, Ockham and the scholastics in the middle ages, and Leibniz's vision of a *calculus ratiocinator* at the dawn of the enlightenment. Our interest in the subject lies with formal logic, which emerged from the contributions of Boole, De Morgan, Frege, Peirce, Peano, and others in the 19th century.

As the 20th century dawned, the leading proponents of formal logic were Hilbert and his colleagues at Göttingen. Hilbert's program was to develop a formal logic that could express in symbols any mathematical statement, a vision that was largely realised by Whitehead and Russell's *Principia Mathematica* [42].

One goal of Hilbert's Program was to solve the *Entscheidungsproblem* (decision problem), that is, to develop an "effectively calculable" procedure to determine the truth or falsity of any statement. The problem presupposes completeness: that for any statement, either it or its negation possesses a proof. In his address to the 1930 Mathematical Congress in Königsberg, Hilbert affirmed his belief in this principle, concluding "Wir müssen wissen, wir werden wissen" ("We must know, we will know"), words later engraved on his tombstone. Perhaps a tombstone is an appropriate place for those words, given that any basis for Hilbert's optimism had been undermined the day before, when at the selfsame conference Gödel [19] presented his proof that arithmetic is incomplete.

While the goal was to satisfy Hilbert's program, no precise definition of "effectively calculable" was required. It would be clear whether a given procedure was effective or not, like Justice Stewart's characterisation of obscenity, "I know it when I see it". But

* Submitted for publication. Apologies for the use of numerical references, which are required by the venue; I normally use date-name.

to show the *Entscheidungsproblem* undecidable required a formal definition of "effectively calculable".

One can find allusions to the concept of algorithm in the work of Euclid and, eponymously, al-Khwarizmi, but the concept was only formalised in the 20th century, and then simultaneously received three independent definitions by logicians. Like buses: you wait two thousand years for a definition of "effectively calculable", and then three come along at once. The three were *lambda calculus*, published May 1935 by Alonzo Church [7], *recursive functions*, proposed by Gödel at lectures in Princeton in 1934 and published July 1935 by Stephen Kleene [26], and *Turing machines*, published May 1936 by Alan Turing [37].

Lambda calculus was introduced by Church at Princeton, and further developed by his students Rosser and Kleene. At this time, Princeton rivalled Göttingen as a centre for the study of logic. The Institute for Advanced study was co-located with the mathematics department in Fine Hall. In 1933, Einstein and von Neumann joined the Institute, and Gödel arrived for a visit.

Logicians have long been concerned with the idea of function. Lambda calculus provides a compact notation for defining functions, including "first-class" functions that accept functions as arguments or return functions as results. It is remarkably compact, containing only three constructs: variables, function abstraction, and function application. Church [6] at first introduced lambda calculus as a way to define compact notations for logical formulas (almost like a macro language) in a new presentation of logic. All forms of bound variable could be subsumed to lambda binding. (For instance, instead of $\exists x.\, A[x]$, Church wrote $\Sigma(\lambda x.\, A[x])$.) However, it was later discovered by Kleene and Rosser that Church's system was inconsistent. By this time, Church and his students had realised that the system was of independent interest. Church had foreseen this possibility in his first paper on the subject, where he wrote "There may, indeed, be other applications of the system than its use as a logic."

Church discovered a way of encoding numbers as terms of lambda calculus. The number n is represented by a function that accepts a function f and a value x, and applies the function to the value n times. (For instance, three is $\lambda f.\, \lambda x.\, f(f(f(x)))$.) With this representation, it is easy to encode lambda terms that can add or multiply, but it was not clear how to encode the predecessor function, which finds the number one less than a given number. One day in the dentist's office, Kleene suddenly saw how to define predecessor [25]. When Kleene brought the result to his supervisor, Church confided that he had nearly convinced himself that representing predecessor in lambda calculus was impossible. Once this hurdle was overcome, Church and his students soon became convinced that any "effectively calculable" function of numbers could be represented by a term in the lambda calculus.

Church proposed λ-definability as the definition of "effectively calculable", what we now know as Church's Thesis, and demonstrated that there was a problem whose solution was not λ-definable, that of determining whether a given λ-term has a normal form, what we now know as the Halting Problem [7]. A year later, he demonstrated there was no λ-definable solution to the *Entscheidungsproblem*.

In 1933, Gödel arrived for a visit at Princeton. He was unconvinced by Church's contention that every effectively calculable function was λ-definable. Church responded by offering that if Gödel would propose a different definition, then Church would "undertake to prove it was included in λ-definability". In a series of lectures at Princeton in 1934, based on a suggestion of Herbrand, Gödel proposed what came to be known as "general recursive functions" as his candidate for effective calculability. Church and his students soon determined that the two definitions are equivalent: every general recursive function is λ-definable, and vice-versa. Rather than mollifying Gödel, this result caused him to doubt that his own definition was correct. Things stood at an impasse.

Meanwhile, at Cambridge, Alan Turing, a student of Max Newman, independently formulated his own notion of "effectively calculable" in the form of what we now call a Turing Machine, and used this to show the *Entscheidungsproblem* undecidable. Before the paper was published, Newman was dismayed to discover that Turing had been scooped by Church. However, Turing's approach was sufficiently different from Church's to merit independent publication. Turing hastily added an appendix sketching the equivalence of λ-definability to his machines, and the paper appeared in print a year after Church [37], when Turing was 23. Newman arranged for Turing to travel to Princeton, where he completed a doctorate under Church's supervision.

Turing's most significant difference from Church was not in logic or mathematics but in philosophy. Whereas Church merely presented the definition of λ-definability and baldly claimed that it corresponded to effective calculability, Turing undertook an analysis of the capabilities of a "computer"—at this time, the term referred a human performing a computation assisted by paper and pencil. Turing argued that the number of symbols must be finite (for if infinite, some symbols would be arbitrarily close to each other and undistinguishable), that the number of states of mind must be finite (for the same reason), and that the number of symbols under consideration at one moment must be bounded ("We cannot tell at a glance whether 9999999999999999 and 999999999999999 are the same"). Later, Gandy [14] would point out that Turing's argument amounts to a theorem asserting that any computation a human with paper and pencil can perform can also be performed by a Turing Machine. It was Turing's argument that finally convinced Gödel; since λ-definability, recursive functions, and Turing machines had been proved equivalent, he now accepted that all three defined "effectively calculable".

As mentioned, Church's first use of lambda calculus was to encode formulas of logic, but this had to be abandoned because it led to inconsistency. This was for a reason much the same as Russell's paradox, namely that it allowed a predicate to act on itself, and so he adapted a solution similar to Russell's, of classifying terms according to types. Church's simply-typed lambda calculus ruled out self-application, permitting lambda calculus to support a consistent logical formulation [8].

These two uses of lambda calculus are rather different in nature. If a notion of computation is powerful enough to represent any effectively calculable procedure, then that notion is not powerful enough to solve its own Halting Problem: there is no effectively calculable procedure to determine whether a given effectively calculable procedure terminates. However, to show the consistency of a logic based on simply-typed lambda calculus, it is necessary to guarantee that every simply-typed lambda term has a normal form. So the application of lambda calculus for general computation requires that the halting problem is not solved, while the application of lambda calculus to logic requires every program halt.

Simply-typed lambda calculus has the property that it is strongly normalising; every reduction sequence terminates, and hence, unlike the original untyped lambda calculus, it cannot represent every effectively calculable function. However, it is straightforward to add recursion as an additional construct (sometimes called the fixpoint operator). Both untyped lambda calculus and simply-typed lambda calculus with recursion have the power to define every effectively calculable function, at the cost of losing strong normalisation. All three systems have their uses, depending on what trade-off one prefers.

3. Gentzen, and the origins of logic

A second goal of Hilbert's program was to establish the consistency of various logics. If a logic is inconsistent, then it can derive any formula, rendering it useless.

In 1935, at the age of 25, Gerhard Gentzen published his doctoral thesis. It introduced not one but two new formulations of logic, *natural deduction* and *sequent calculus*, which became established as the two major systems for formulating a logic, and remain so to this day. It showed how to normalise proofs to ensure they were not "roundabout", yielding a new proof of the consistency of Hilbert's system. And, to top it off, to match the use of $\exists$ for the existential quantification introduced by Peano, Gentzen introduced $\forall$ to denote universal quantification.

Gentzen's insight was that proof rules should come in pairs, a feature that had not been present in earlier systems such as Hilbert's. In natural deduction, these are introduction and elimination pairs. An introduction rule specifies under what circumstances one may assert a formula with a logical connective (for instance, to prove $A \supset B$ one may assume A and then must prove B), while the corresponding elimination rule shows how to use that logical connective (for instance, from a proof of $A \supset B$ and a proof of A one may deduce B, a property dubbed *modus ponens* in the middle ages). As Gentzen notes, "The introductions represent, as it were, the 'definitions' of the symbols concerned, and the eliminations are no more, in the final analysis, than the consequences of these definitions."

A consequence of this insight was that any proof could be normalised to one that is not "roundabout", where "no concepts enter into the proof other than those contained in the final result". For example, in a normalised proof of the formula $A \,\&\, B$, the only formulas that may appear are itself and its subformulas, A and B, and the subformulas of A and B themselves. No other formula, such as $(B \,\&\, A) \supset (A \,\&\, B)$ or $A \vee B$, may appear; this is called the Subformula Property. An immediate consequence was consistency. The only way to derive a contradiction (that is, to prove false, written $\lightning$), is to prove, say, both $A \supset \lightning$ and A for some formula A. But given such a proof, one could normalise it to one containing only subformulas of its conclusion, $\lightning$. But $\lightning$ has no subformulas! It is like the old saw, "What part of *no* don't you understand?" Logicians became interested in normalisation of proofs because of its role in establishing consistency.

Gentzen preferred the system of Natural Deduction because it was, in his view, more natural. He introduced Sequent Calculus as a technical device for proving the Subformula Property. Sequent Calculus had two key properties. First, every proof in Natural Deduction can be converted to a proof in Sequent Calculus, and conversely, so the two systems are equivalent. Second, unlike Natural Deduction, every rule save one has the property that its hypotheses only involve subformulas of those that appear in its conclusion. The one exception, the Cut rule, can always be removed by a process called Cut Elimination. Hence every proof had an equivalent normal form satisfying the Subformula Property. Gentzen's main interest in Sequent Calculus was to prove the Subformula Property, although Sequent Calculus has features of independent interest, such as providing a more symmetric presentation of classical logic.

It is an irony that Gentzen was required to introduce Sequent Calculus in order to prove the Subformula Property for Natural Deduction. He needed a roundabout proof to show the absence of roundabout proofs! Later, in 1965, Prawitz showed how to prove the Subformula Property directly, by introducing a way to simplify Natural Deduction proofs. We will return to this point later, also.

4. Propositions as Types

In their textbook, Curry and Feys [11] observed a curious fact, relating a theory of functions to a theory of implication. Every type of a function ($A \to B$) could be read as a proposition ($A \supset B$), and under this reading the type of any given function would always correspond to a provable proposition. Conversely, for every provable proposition there was a function with a corresponding type.

In 1969, Howard circulated a mimeographed manuscript. It was not published until 1980, where it appeared in a Festschrift dedicated to Curry [24]. Motivated by Curry's observation, Howard points out that there is a similar correspondence between natural deduction, on the one hand, and simply-typed lambda calculus on the other, and (unlike Curry) he makes explicit the third and deepest level of the correspondence as described in the introduction, that normalisation of proofs corresponds to evaluation of programs. Howard shows the correspondence extends to the other logical connectives, conjunction and disjunction, by extending his lambda calculus with constructs that represent pairs and disjoint sums. Just as proof rules come in introduction and elimination pairs, so do typing rules: introduction rules correspond to ways to define or construct a value of the given type, and elimination rules correspond to ways to use or deconstruct values of the given type.

We can describe Howard's observation as follows:

- Conjunction $A \,\&\, B$ corresponds to Cartesian product $A \times B$, that is, a record with two fields, also known as a pair. A proof of the proposition $A \,\&\, B$ consists of a proof of A and a proof of B. Similarly, a value of type $A \times B$ consists of a value of type A and a value of type B.
- Disjunction $A \vee B$ corresponds to a disjoint sum $A + B$, that is, a variant with two alternatives. A proof of the proposition $A \vee B$ consists of either a proof of A or a proof of B, including an indication of which of the two has been proved. Similarly, a value of type $A + B$ consists of either a value of type A or a value of type B, including an indication of whether this is a left or right summand.
- Implication $A \supset B$ corresponds to function space $A \to B$. A proof of the proposition $A \supset B$ consists of a procedure that given a proof of A yields a proof of B. Similarly, a value of type $A \to B$ consists of a function that when applied to a value of type A returns a value of type B.

This reading of proofs goes back to the intuitionists, and is often called the BHK interpretation, named for Brouwer, Heyting, and Kolmogorov. Brouwer founded intuitionism, and Heyting and Kolmogorov formalised intuitionistic logic, and developed the interpretation above, in the 1920s and 1930s.

Given the intuitionistic reading of proofs, it hardly seems surprising that intuitionistic natural deduction and lambda calculus should correspond so closely. But recall that Gentzen invented Sequent Calculus because he could not find a direct proof of the Subformula Property for Natural Deduction, and the direct proof was not published until three decades later, by Prawitz. As noted by Howard, Prawitz's technique for normalising a proof corresponds exactly to reduction of lambda terms. Gentzen, Church, and Prawitz never drew these parallels. Certainly Howard was proud of the connection he drew, citing it as one of the two great achievements of his career [35]. While the connection may be obvious in retrospect, it was far from obvious in prospect!

Howard's paper divides into two halves. The first half explains a correspondence between two well-understood concepts, the propositional connectives $\&$, $\vee$, $\supset$ on the one hand and the computational types $\times$, $+$, $\to$ on the other hand. The second half extends this analogy, and for well-understood concepts from logic proposes new

concepts for types that correspond to them. In particular, Howard proposes that the predicate quantifiers $\forall$ and $\exists$ corresponds to new types that we now call *dependent types*.

With the introduction of dependent types, every proof in predicate logic can be represented by a term of a suitable typed lambda calculus. Mathematicians and computer scientists proposed numerous systems based on this concept, including de Bruijn's Automath [13], Martin-Löf's type theory [28], Bates and Constable's PRL and nuPRL, [2], and Coquand and Huet's Calculus of Constructions [9]. The last of these developed into the Coq proof assistant, which was later used by Gonthier to validate the proof of the four-colour theorem [20], and by Leroy to verify the correctness of a C compiler [27].

5. Intuitionistic logic

In Gilbert and Sullivan's *The Gondoliers*, Casilda is told that as an infant she was married to the heir of the King of Batavia, but that due to a mix-up no one knows which of two individuals, Marco or Giuseppe, is the heir. Alarmed, she wails "Then do you mean to say that I am married to one of two gondoliers, but it is impossible to say which?" To which the response is "Without any doubt of any kind whatever."

Logic comes in many varieties, and one distinction is between *classical* and *intuitionistic*. Intuitionists, concerned by cavalier assumptions made by some logicians about the nature of infinity, insist upon a constructionist notion of truth. In particular, they insist that a proof of $A \vee B$ must show *which* of A or B holds, and hence they would reject the claim that Casilda is married to Marco or Giuseppe until one of the two was identified as her husband. Perhaps Gilbert and Sullivan anticipated intuitionism, for their story's outcome is that the heir turns out to be a third individual, Luiz, with whom Casilda is, conveniently, already in love.

Intuitionists also reject the law of the excluded middle, which asserts $A \vee \neg A$ for every A, since the law gives no clue as to *which* of A or $\neg A$ holds. Heyting formalised a variant of Hilbert's classical logic that captures the intuitionistic notion of provability. In particular, the law of the excluded middle is provable in Hilbert's logic, but not in Heyting's. Further, if the law of the excluded middle is added as an axiom to Heyting's logic, then it becomes equivalent to Hilbert's. Kolmogorov showed the two logics were closely related: he gave a double-negation translation, such that a formula is provable in classical logic if and only if its translation is provable in intuitionistic logic.

Propositions as Types was first formulated for intuitionistic logic. It is a perfect fit, because in the intuitionist interpretation the formula $A \vee B$ is provable exactly when one exhibits either a proof of A or a proof of B, so the type corresponding to disjunction is a disjoint sum.

6. Other logics, other computation

The principle of Propositions as Types would be remarkable even if it applied only to one variant of logic and one variant of computation. How much more remarkable, then, that it applies to a wide variety of logics and of computation.

Quantification over propositional variables in second-order logic corresponds to type abstraction in second-order lambda calculus. A consequence of this correspondence is that a principal type inference algorithm was discovered twice, once by the logician Roger Hindley [22] and once by the computer scientist Robin Milner [29]. For the same reason, a related but more powerful system, the second-order lambda calculus, was also discovered twice, once by the logician Jean-Yves Girard [18] and once by the computer scientist John Reynolds [34]. The type systems of functional languages including ML and Haskell is based upon the Hindley-Milner system, while the design of generic types in Java and C# draws directly upon the Girard-Reynolds system. Subsequently, Mitchell and Plotkin [30] observed that existential quantification in second-order logic corresponds precisely to data abstraction, an idea that now underpins much research in the semantics of programming languages. Philosophers might argue as to whether mathematical systems are 'discovered' or 'devised', but the same system arising in two different contexts argues that here the correct word is 'discovered'.

Two major variants of logic are intuitionistic and classical. Howard's original paper observed a correspondence with intuitionistic logic. Not until a decade later was the correspondence extended to also apply to classical logic, when Tim Griffin [21] observed that Peirce's Law in classical logic provides a type for the call/cc operator of Scheme. Chet Murthy [32] went on to note that Kolmogorov and Gödel's double-negation translation, widely used to relate intuitionistic and classical logic, corresponds to the continuation-passing style transformation widely used both by semanticists and implementers of lambda calculus. Parigot [33], Curien and Herbelin [10], and Wadler [40] introduced various computational calculi motivated by correspondences to classical logic.

Another variant of logic is modal logic, where propositions can be labelled as 'necessarily true' or 'possibly true'. Eugenio Moggi [31] introduced monads as a technique to explain the semantics of important features of programming languages such as state, exceptions, and input-output, and monads became widely adopted in the functional language Haskell, later migrating into other languages including Clojure, Scala, F#, and C#. Benton, Bierman, and de Paiva [3] observed one form of 'possibility' in modal logic corresponds precisely to monads.

In classical, intuitionistic, and modal logic, any hypothesis can be used an arbitrary number of times—zero, once, or many. Linear logic, introduced in 1987 by Girard [17], requires that each hypothesis is used exactly once. Linear logic is 'resource conscious' in that facts may be used up and superseded by other facts, suiting it for reasoning about a world where situations change. From its inception, linear logic was suspected to apply to problems of importance to computer scientists, and its first publication was not in *Annals of Mathematics* but in *Theoretical Computer Science*. Computational aspects of linear logic are discussed by Abramsky [1] and Wadler [39], among many others, and applications to quantum computing are surveyed by Gay [15]. Most recently, Caires and Pfenning [4] have applied Propositions as Types to explain Session Types, a way of describing communication protocols introduced by Honda [23], inspiring a new view of Propositions as Sessions [41].

7. Natural deduction

We now turn to a more formal development, presenting a fragment of natural deduction and a fragment of typed lambda calculus in a style that makes clear the connection between the two.

We begin with the details of natural deduction as defined by Gentzen [16]. The proof rules are shown in Figure 1. To simplify our discussion, we consider just two of the connectives of natural deduction. We write A and B as placeholders standing for arbitrary formulas. Conjunction is written $A \,\&\, B$ and implication is written $A \supset B$.

We represent proofs by trees, where each node of the tree is an instance of a proof rule. Each proof rule consists of zero or more formulas written above a line, called the *premises*, and a single formula written below the line, called the *conclusion*. The interpretation of a rule is that when all the premises hold, then the conclusion follows.

The proof rules come in pairs, with rules to introduce and to eliminate each connective, labelled -I and -E respectively. As we read the rules from top to bottom, introduction and elimination

rules do what they say on the tin: the first *introduces* a formula for the connective, which appears in the conclusion but not in the premises; the second *eliminates* a formula for the connective, which appears in a premise but not in the conclusion. An introduction rule describes under what conditions we say the connective holds—how to *define* the connective. An elimination rule describes what we may conclude when the connective holds—how to *use* the connective.

The introduction rule for conjunction, &-I, states that if formula A holds and formula B holds, then the formula $A \,\&\, B$ must hold as well. There are two elimination rules for conjunction. The first, &-E_0, states that if the formula $A \,\&\, B$ holds, then the formula A must hold as well. The second, &-E_1 concludes B rather than A.

The introduction rule for implication, $\supset$-I, states that if from the *assumption* that formula A holds we may derive the formula B, then we may conclude that the formula $A \supset B$ holds and *discharge* the assumption. To indicate that A is used as an assumption zero, once, or many times in the proof of B, we write A in brackets and tether it to B via a chain of ellipses. A proof is complete only when every assumption in it has been discharged by a corresponding use of $\supset$-I, which is indicated by writing the same name (here x) as a superscript on each instance of the discharged assumption and on the discharging rule. The elimination rule for implication, $\supset$-E, states that if formula $A \supset B$ holds and if formula A holds, then we may conclude formula B holds as well. In the middle ages, logicians dubbed this last rule *modus ponens*.

Critical readers will observe that we use similar language to describe rules ('when-then') and formulas ('implies'). The same idea applies at two levels, the meta level (rules) and the object level (formulas), and in two notations, using a line with premises above and conclusion below for implication at the meta level, and the symbol $\supset$ with premise to the left and conclusion to the right at the object level. It is almost as if to understand implication one must first understand implication! This Zeno's paradox of logic was wryly observed by Lewis Carroll [5]. We need not let it disturb us; everyone possesses a good informal understanding of implication, which may act as a foundation for its formal description.

A proof of the formula

$$(B \,\&\, A) \supset (A \,\&\, B).$$

is shown in Figure 2. In other words, if B and A hold then A and B hold. This may seem so obvious as to be hardly deserving of proof! However, the formulas $B \supset A$ and $A \supset B$ have meanings that differ, and we need some formal way to conclude that the formulas $B \,\&\, A$ and $A \,\&\, B$ have meanings that are the same. This is what our proof shows, and it is reassuring that it can be constructed from the rules we posit.

The proof reads as follows. From $B \,\&\, A$ we conclude A, by &-E_1, and from $B \,\&\, A$ we also conclude B, by &-E_0. From A and B we conclude $A \,\&\, B$, by &-I. That is, from the assumption $B \,\&\, A$ (used twice) we conclude $A \,\&\, B$. We discharge the assumption and conclude $(B \,\&\, A) \supset (A \,\&\, B)$ by $\supset$-I, linking the discharged assumptions to the discharging rule by writing z as a superscript on each.

Now consider a larger proof built from this smaller proof, as shown at the top of Figure 4. This proof makes two assumptions, $[B]$ and $[A]$, and concludes with the formula $A \,\&\, B$. However, rather than concluding it directly we derive the result in a roundabout way, in order to illustrate an instance of $\supset$-E, *modus ponens*. The proof reads as follows. On the left is the proof given previously, concluding in $(B \,\&\, A) \supset (A \,\&\, B)$. On the right, from B and A we conclude $B \,\&\, A$ by &-I. Combining these yields $A \,\&\, B$ by $\supset$-I.

This proof is unnecessarily roundabout. We may simplify it by applying the rewrite rules of Figure 3. These rules specify how to simplify a proof when an introduction rule is immediately followed

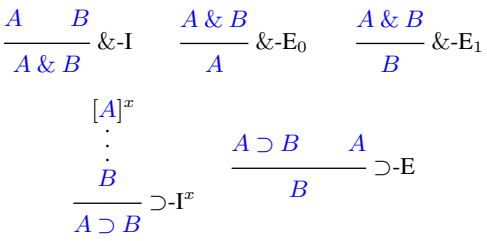

Figure 1. Gerhard Gentzen (1935) — Natural Deduction

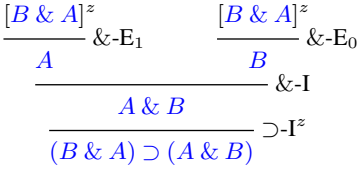

Figure 2. A proof

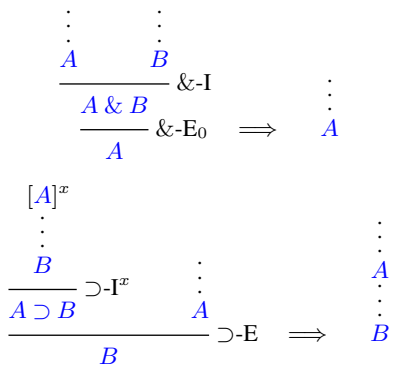

Figure 3. Simplifying proofs

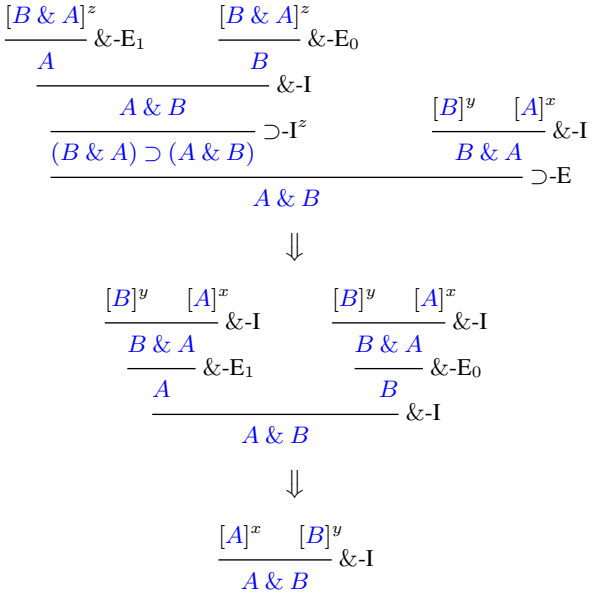

Figure 4. Simplifying a proof

by the corresponding elimination rule. Each rule shows two proofs connected by an arrow, indicating that the *redex* (the proof on the left) may be rewritten, or simplified, to yield the *reduct* (the proof on the right). Rewrites always take a valid proof to another valid proof.

For &, the redex consists of a proof of A and a proof of B, which combine to yield $A \,\&\, B$ by &-I, which in turn yields A by &-E$_0$. The reduct consists simply of the proof of A, discarding the unneeded proof of B. There is a similar rule, not shown, to simplify an occurrence of &-I followed by &-E$_1$.

For $\supset$, the redex consists of a proof of B from assumption A, which yields $A \supset B$ by $\supset$-I, and a proof of A, which combine to yield B by $\supset$-E. The reduct consists of the same proof of B, but now with every occurrence of the assumption A replaced by the given proof of A. The assumption A may be used zero, once, or many times in the proof of B in the redex, so the proof of A may be copied zero, once, or many times in the proof of B in the reduct. For this reason, the reduct may be larger than the redex, but it will be simpler in the sense that it has removed an unnecessary detour via the subproof of $A \supset B$.

We can think of the assumption of A in $\supset$-I as a debt which is discharged by the proof of A provided in $\supset$-E. The proof in the redex accumulates debt and pays it off later; while the proof in the reduct pays directly each time the assumption is used. Proof debt differs from monetary debt in that there is no interest, and the same proof may be duplicated freely as many times as needed to pay off an assumption, the very property which money, by being hard to counterfeit, is designed to avoid!

Figure 4 demonstrates use of these rules to simplify a proof. The first proof contains an instance of $\supset$-I followed by $\supset$-E, and is simplified by replacing each of the two assumptions of $B \,\&\, A$ on the left by a copy of the proof of $B \,\&\, A$ on the right. The result is the second proof, which as a result of the replacement now contains an instance of &-I followed by &-E$_1$, and another instance of &-I followed by &-E$_0$. Simplifying each of these yields the third proof, which derives $A \,\&\, B$ directly from the assumptions A and B, and can be simplified no further.

It is not hard to see that proofs in normal form satisfy the Subformula Property: every formula of such a proof must be a subformula of one of its undischarged assumptions or of its conclusion. The proof in Figure 2 and the final proof of Figure 4 both satisfy this property, while the first proof of Figure 4 does not, since $(B \,\&\, A) \supset (A \,\&\, B)$ is not a subformula of $A \,\&\, B$.

8. Lambda calculus

We now turn our attention to the simply-typed lambda calculus of Church [6, 8]. The type rules are shown in Figure 5. To simplify our discussion, we take both products and functions as primitive types; Church's original calculus contained only function types, with products as a derived construction. We now write A and B as placeholders for arbitrary types, and L, M, N as placeholder for arbitrary terms. Products are written $A \times B$ and functions are written $A \to B$. Now instead of formulas, our premises and conclusions are judgments of the form

$$M : A$$

indicating that term M has type A.

We represent type derivations by trees, where each node of the tree is an instance of a type rule. Each type rule consists of zero or more formulas written above a line, called the *premises*, and a single formula written below the line, called the *conclusion*. The interpretation of a rule is that when all the premises hold, then the conclusion follows.

Type rules, like proof rules, come in pairs. An introduction rule describes how to *define* or *construct* a term of the given type, while

$$\frac{M : A \qquad N : B}{\langle M, N\rangle : A \times B}\,\times\text{-I} \qquad \frac{L : A \times B}{L_0 : A}\,\times\text{-E}_0 \qquad \frac{L : A \times B}{L_1 : B}\,\times\text{-E}_1$$

$$\frac{\begin{array}{c}[x : A]^x \\ \vdots \\ N : B\end{array}}{\lambda x.\, N : A \to B}\,\to\text{-I}^x \qquad \frac{L : A \to B \qquad M : A}{L\, M : B}\,\to\text{-E}$$

Figure 5. Alonzo Church (1935) — Lambda Calculus

$$\dfrac{\dfrac{\dfrac{[z : B \times A]^z}{z_1 : A}\,\times\text{-E}_1 \qquad \dfrac{[z : B \times A]^z}{z_0 : B}\,\times\text{-E}_0}{\langle z_1, z_0\rangle : A \times B}\,\times\text{-I}}{\lambda z.\, \langle z_1, z_0\rangle : (B \times A) \to (A \times B)}\,\to\text{-I}^z$$

Figure 6. A program

$$\dfrac{\dfrac{\begin{array}{c}\vdots \\ M : A\end{array} \qquad \begin{array}{c}\vdots \\ N : B\end{array}}{\langle M, N\rangle : A \times B}\,\times\text{-I}}{\langle M, N\rangle_0 : A}\,\times\text{-E}_0 \quad\Longrightarrow\quad \begin{array}{c}\vdots \\ M : A\end{array}$$

$$\dfrac{\dfrac{\begin{array}{c}[x : A]^x \\ \vdots \\ N : B\end{array}}{\lambda x.\, N : A \to B}\,\to\text{-I}^x \qquad \begin{array}{c}\vdots \\ M : A\end{array}}{(\lambda x.\, N)\, M : B}\,\to\text{-E} \quad\Longrightarrow\quad \begin{array}{c}\vdots \\ M : A \\ \vdots \\ N[M/x] : B\end{array}$$

Figure 7. Simplifying programs

$$\dfrac{\dfrac{\dfrac{\dfrac{[z : B \times A]^z}{z_1 : A}\,\times\text{-E}_1 \qquad \dfrac{[z : B \times A]^z}{z_0 : B}\,\times\text{-E}_0}{\langle z_1, z_0\rangle : A \times B}\,\times\text{-I}}{\lambda z.\, \langle z_1, z_0\rangle : (B \times A) \to (A \times B)}\,\to\text{-I}^z \qquad \dfrac{[y : B]^y [x : A]^x}{\langle y, x\rangle : B \times A}\,\times\text{-I}}{(\lambda z.\, \langle z_1, z_0\rangle)\, \langle y, x\rangle : A \times B}\,\to\text{-E}$$

$$\Downarrow$$

$$\dfrac{\dfrac{\dfrac{[y : B]^y [x : A]^x}{\langle y, x\rangle : B \times A}\,\times\text{-I}}{\langle y, x\rangle_1 : A}\,\times\text{-E}_1 \qquad \dfrac{\dfrac{[y : B]^y [x : A]^x}{\langle y, x\rangle : B \times A}\,\times\text{-I}}{\langle y, x\rangle_0 : B}\,\times\text{-E}_0}{\langle \langle y, x\rangle_1, \langle y, x\rangle_0\rangle : A \times B}\,\times\text{-I}$$

$$\Downarrow$$

$$\dfrac{[x : A]^x \qquad [y : B]^y}{\langle x, y\rangle : A \times B}\,\times\text{-I}$$

Figure 8. Simplifying a program

an elimination rule describes how to *use* or *deconstruct* a term of the given type.

The introduction rule for products, $\times$-I, states that if term M has type A and term N has type B, then we may form the pair term $\langle M, N \rangle$ of product type $A \times B$. There are two elimination rules for products. The first, $\times$-E$_0$, states that if term L has type $A \times B$, then we may form the term L_0 of type A, which selects the first component of the pair. The second, $\times$-E$_1$ is similar, save that it forms the term L_1 of type B.

The introduction rule for functions, $\rightarrow$-I, states that if given a variable x of type A we may form a term N of type B, then we may form the lambda term $\lambda x.\, N$ of function type $A \rightarrow B$. The variable x appears *free* in N and *bound* in $\lambda x.\, N$. Undischarged assumptions correspond to free variables, while discharged assumptions correspond to bound variables. To indicate that the variable x may appear zero, once, or many times in the term N, we write $x : A$ in brackets and tether it to $N : B$ via a chain of ellipses. A term is closed only when every variable in it is bound by a corresponding λ term. The elimination rule for functions, $\rightarrow$-E, states that given term L of type $A \rightarrow B$ and term M of type A we may form the application term $L\, M$ of type B.

For natural deduction, we noted that there might be confusion between implication at the meta level and the object level. For lambda calculus the distinction is clearer, as we have implication at the meta level (if terms above the line are well typed so are terms below) but functions at the object level (a function has type $A \rightarrow B$ because if it is passed a value of type A then it returns a value of type B). What previously had been discharge of assumptions (perhaps a slightly diffuse concept) becomes binding of variables (a concept understood by most computer scientists).

The reader will by now have observed a striking similarity between Gentzen's rules from the preceding section, and Church's rules from this section: ignoring the terms in Church's rules then they are identical, if one replaces & by $\times$ and $\supset$ by $\rightarrow$. The colouring of the rules is chosen to highlight the similarity.

A program of type

$$(A \times B) \rightarrow (B \times A)$$

is shown in Figure 6. Whereas the difference between $A \,\&\, B$ and $B \,\&\, A$ appears a mere formality, the the difference between $A \times B$ and $B \times A$ is easier to appreciate: converting the latter to the former requires swapping the elements of the pair, which is precisely the task performed by the program corresponding to our former proof.

The program reads as follows. From variable z of type $B \times A$ we form term z_1 of type A by $\times$-E$_1$ and also term z_0 of type B by $\times$-E$_0$. From these two we form the pair $\langle z_1, z_0 \rangle$ of type $A \times B$ by $\times$-I. Finally, we bind the free variable z to form the lambda term $\lambda z.\, \langle z_1, z_0 \rangle$ of type $(A \times B) \rightarrow (B \times A)$ by $\rightarrow$-I, connecting the bound typings to the binding rule by writing z as a superscript on each. The function accepts a pair and swaps its elements, exactly as described by its type.

Now consider a larger program built from this smaller program, as shown at the top of Figure 8. This program has two free variables, y of type B and x of type A, and constructs a value of type $A \times B$. However, rather than constructing it directly we reach the result in a roundabout way, in order to illustrate an instance of $\rightarrow$-E, function application. The program reads as follows. On the left is the program given previously, forming a function of type $(B \times A) \rightarrow (A \times B)$. On the right, from B and A we form the pair $\langle y, x \rangle$ of type $B \times A$ by $\times$-I. Applying the function to the pair forms a term of type $A \times B$ by $\rightarrow$-I.

This program is not in normal form. We may simplify it by applying the rewrite rules of Figure 7. These rules specify how to simplify a term when an introduction rule is immediately followed by the corresponding elimination rule. Each rule shows two derivations connected by an arrow, indicating that the *redex* (the term on the left) may be rewritten, or simplified, to yield the *reduct* (the term on the right). Rewrites always take a valid type derivation to another valid type derivation, ensuring that rewrites preserve types, a property known as *subject reduction* or *type soundness*.

For $\times$, the redex consists of term M of type A and term N of type B, which combine to yield term $\langle M, N \rangle$ of type $A \times B$ by $\times$-I, which in turn yields term $\langle M, N \rangle_0$ of type A by $\times$-E$_0$. The reduct consists simply of term M of type A, discarding the unneeded term N of type B. There is a similar rule, not shown, to simplify an occurrence of $\times$-I followed by $\times$-E$_1$.

For $\rightarrow$, the redex consists of a derivation of term N of type B from variable x of type A, which yields the lambda term $\lambda x.\, N$ of type $A \rightarrow B$ by $\rightarrow$-I, and a derivation of term M of type A, which combine to yield the application $(\lambda x.\, N)\, M$ of type B by $\rightarrow$-E. The reduct consists of the term $N[M/x]$ that replaces each free occurrence of the variable x in term N by term M. Further, replacing each assumption that x has type A in the derivation that N has type B by the derivation that M has type A gives a derivation showing that $N[M/x]$ has type B. Since the variable x may appear zero, once, or many times in the term N, the term M may be copied zero, once, or many times in the reduct $N[M/x]$. For this reason, the reduct may be larger than the redex, but it will be simpler in the sense that is has removed a subterm of type $A \rightarrow B$. Thus, discharge of assumptions corresponds to applying a function to its argument.

Figure 8 demonstrates use of these rules to simplify a program. The first program contains an instance of $\rightarrow$-I followed by $\rightarrow$-E, and is simplified by replacing each of the two occurrences of z of type $B \times A$ on the left by a copy of the term $\langle y, x \rangle$ of type $B \times A$ on the right. The result is the second program, which as a result of the replacement now contains an instance of $\times$-I followed by $\times$-E$_1$, and another instance of $\times$-I followed by $\times$-E$_0$. Simplifying each of these yields the third proof, which derives the term $\langle x, y \rangle$ of type $A \times B$, and can be simplified no further. Hence, simplification of proofs corresponds exactly to evaluation of programs, in this instance demonstrating that applying the function to the pair indeed swaps its elements.

9. Conclusion

Proposition as Types informs our view of the universality of certain programming languages.

The Pioneer spaceship contains a plaque designed to communicate with aliens, if any should ever intercept it (see Figure 9). They may find some parts of it easier to interpret than others. A radial diagram shows the distance of fourteen pulsars and the centre of the galaxy from Sol. Aliens are likely to determine that the length of each line is proportional to the distances to each body. Another diagram shows humans in front of a silhouette of Pioneer. If Star Trek gives an accurate conception of alien species, they may respond "They look just like us, except they lack pubic hair." However, if the aliens's perceptual system differs greatly from our own, they may be unable to decipher these squiggles.

What would happen if we tried to communicate with aliens by transmitting a computer program? In the movie Independence Day, the heroes destroy the invading alien mother ship by infecting it with a computer virus. Close inspection of the transmitted program shows it contains curly braces—it is written in a dialect of C! It is unlikely that alien species would program in C, and doubtful that aliens could decipher a program written in C if presented with one.

What about lambda calculus? Propositions as Types tell us that lambda calculus is isomorphic to natural deduction. It seems difficult to conceive of alien beings that do not know the fundamentals of logic, and we might expect the problem of deciphering a program written in lambda calculus to be closer to the problem of un-

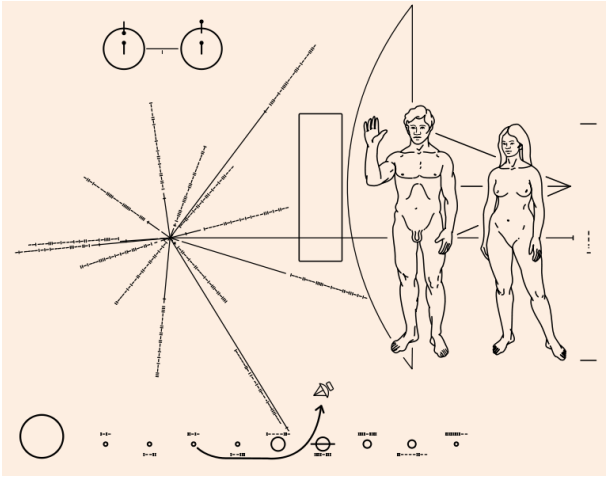

Figure 9. Plaque on Pioneer Spaceship

derstanding the radial diagram of pulsars than that of understanding the image of a man and a woman on the Pioneer plaque.

We might be tempted to conclude that lambda calculus is universal, but first let's ponder the suitability of the word 'universal'. These days the multiple worlds interpretation of quantum physics is widely accepted. Scientists imagine that in different universes one might encounter different fundamental constants, such as the strength of gravity or the Planck constant. Such constants appear finely-tuned to values conducive to the formation of stars, and hence life. Some explain this by saying we must be in a universe where the constants are so tuned, else there could be no living things to observe the result [12]. But easy as it may be to imagine a universe where gravity differs, it is difficult to conceive of a universe where fundamental rules of logic fail to apply. Natural deduction, and hence lambda calculus, should not only be known by aliens throughout our universe, but also throughout others. So we may conclude it would be a mistake to characterise lambda calculus as a universal language, because calling it universal is *too limiting*.

Acknowledgements. Thanks to John Hughes, Simon Peyton-Jones, and Benjamin Pierce for comments on an earlier draft, to Moshe Vardi for kibitzing, and to Martin Erwig, Mikhail Glushenkov, Gabor Greif, Sylvain Henry, Daniel Marsden, Lee Pike, Andrés Sicard-Ramírez, and Dann Toliver for spotting typos.

References

[1] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1&2):3–57, 1993.

[2] J. L. Bates and R. L. Constable. Proofs as programs. *Transactions on Programming Languages and Systems*, 7(1):113–136, Jan. 1985.

[3] P. N. Benton, G. M. Bierman, and V. de Paiva. Computational types from a logical perspective. *Journal of Functional Programming*, 8(2):177–193, 1998.

[4] L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, pages 222–236, 2010.

[5] L. Carroll. What the Tortoise said to Achilles. *Mind*, 4(14):278–280, April 1895.

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

[7] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1938. Presented to the American Mathematical Society, 19 April 1935; abstract in *Bulletin of the American Mathematical Society*, **41**, May 1935.

[8] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[9] T. Coquand and G. P. Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[10] P.-L. Curien and H. Herbelin. The duality of computation. In *International Conference on Functional Programming (ICFP)*, pages 233–243, 2000.

[11] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, 1958.

[12] P. Davies. *The Goldilocks Engima: Why Is the Universe Just Right for Life?* Houghton Mifflin Harcourt, 2008.

[13] N. G. de Bruijn. The mathematical language Automath, its usage, and some of its extensions. In *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Computer Science*, pages 29–61. Springer-Verlag, 1968.

[14] R. Gandy. The confluence of ideas in 1936. In R. Herken, editor, *The Universal Turing Machine: a Half-Century Survey*, pages 51–102. Springer, 1995.

[15] S. Gay. Quantum programming languages: survey and bibliography. *Mathematical Structures in Computer Science*, 16(4):581–600, 2006.

[16] G. Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39(2–3):176–210, 405–431, 1935. Reprinted in [36].

[17] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[18] J.-Y. Girard. *Interprétation functionelle et élimination des coupures dans l'arithmétique d'ordre supérieure*. PhD thesis, Université Paris VII, 1989.

[19] K. Gödel. Über formal unterscheidbare Sätze der *Principia Mathematica* und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. Reprinted in [38].

[20] G. Gonthier. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[21] T. Griffin. A formulae-as-types notion of control. In *Principles of Programming Languages (POPL)*, pages 47–58. ACM, Jan. 1990.

[22] R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, Dec. 1969.

[23] K. Honda. Types for dyadic interaction. In *CONCUR*, pages 509–523, 1993.

[24] W. A. Howard. The formulae-as-types notion of construction. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–491. 1980. The original version was circulated privately in 1969.

[25] S. Kleene. Origins of recursive function theory. *Annals of the History of Computing*, 3(1):52–67, 1981.

[26] S. C. Kleene. General recursive functions of natural numbers (abstract). *Bulletin of the American Mathematical Society*, July 1935.

[27] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[28] P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis Naples, Italy, 1984.

[29] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[30] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[31] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[32] C. Murthy. An evaluation semantics for classical proofs. In *Logic in Computer Science (LICS)*, pages 96–107, 1991.

[33] M. Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Logic programming and automated reasoning*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer-Verlag, 1992.

[34] J. C. Reynolds. Towards a theory of type structure. In *Symposium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423, 1974.

[35] A. E. Shell-Gellasch. Reflections of my advisor: Stories of mathematics and mathematicians. *The Mathematical Intelligencer*, 25(1):35–41, 2003.

[36] M. E. Szabo, editor. *The collected papers of Gerhard Gentzen*. North Holland, 1969.

[37] A. M. Turing. On computable numbers, with an application to the *Entscheidungsproblem*. *Proceedings of the London Mathematical Society*, May 1936.

[38] J. van Heijenoort. *From Frege to Gödel: a sourcebook in mathematical logic, 1879–1931*. Harvard University Press, 1967.

[39] P. Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science (MFCS)*, volume 711 of *LNCS*, pages 185–210. Springer-Verlag, 1993.

[40] P. Wadler. Call-by-value is dual to call-by-name. In *International Conference on Functional Programming (ICFP)*, pages 189–201. ACM, 2003.

[41] P. Wadler. Propositions as sessions. In *International Conference on Functional Programming (ICFP)*, pages 273–286. ACM, 2012.

[42] A. N. Whitehead and B. Russell. *Principia mathematica*. University Press, 1912.