

Connecting mathematical logic and computation, it ensures that some aspects of programming are absolute.

BY PHILIP WADLER

Propositions as Types

POWERFUL INSIGHTS ARISE from linking two fields of study previously thought separate. Examples include Descartes's coordinates, which links geometry to algebra, Planck'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 automated proof assistants and programming languages, and continuing to influence the forefronts of computing.

» key insights

- **Propositions as Types observes a deep correspondence between logic and computation: propositions in a logic correspond to types in a programming language; proofs of propositions correspond to programs of the corresponding type; and simplification of proofs corresponds to evaluation of programs.**
- **Propositions as Types is broadly applicable, applying to a wide variety of logics (intuitionistic, second-order, classical, linear) and of language features (lambda calculus parametric polymorphism, continuations, concurrency).**
- **Often the same ideas are discovered independently by logicians and computer scientists, demonstrating some aspects of programming language design are not arbitrary but absolute.**

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 1934 and refined by Howard in 1969. Others draw attention to significant contributions from de Bruijn's Automath and Martin-Löf's Type Theory in the 1970s.

Propositions as Types is a notion with depth. It describes a correspondence between a given logic and a given programming language. 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.

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 simplify a proof there is a corresponding way to evaluate a program—and vice versa. Thus we further have

*simplification 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, simplifications, 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, records, variants, parametric polymorphism, data abstraction, continuations, monads, linear types, and session types. It has inspired automated proof assistants and programming languages, including Agda, Automath, Coq, Epigram, F#, F*, Haskell, LF, ML, NuPRL, Scala, Singularity, and Trellis.

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 30 years later to be essentially identical? And why should it be the case that the same correspondence arises again and again? The logician Hindley and the computer scientist Milner independently developed the same type system, now dubbed Hindley–Milner. The logician Girard and the computer scientist Reynolds independently developed the same calculus, now dubbed Girard–Reynolds. Curry–Howard is a double-barreled name that ensures the existence of other double-barreled names. Those of us who design and use programming languages may often feel they are arbitrary, but Propositions as Types assures us some aspects of programming are absolute. (See the online appendix, which contains a full version of this article, along with additional details and references, plus a historic note provided by William Howard.)

Church and the Theory 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, Whitehead and Russell’s *Principia Mathematica* demonstrated formal logic could express a large part of mathematics. Inspired by this vision, Hilbert and his colleagues at Göttingen became the leading proponents of formal logic, aiming to put it on a firm foundation.

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 these words, given that any basis for Hilbert’s optimism had been undermined the day before, when at the selfsame conference Gödel¹⁸ announced 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 characterization of obscenity, “I know it when I see it.” But 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 formalized only in the 20th century, and then simultaneously received three independent definitions by logicians. Like buses, you wait 2,000 years for a definition of

“effectively calculable,” and then three come along at once. The three were lambda calculus, published in 1936 by Church,⁷ recursive functions, proposed by Gödel at lectures in Princeton in 1934 and published in 1936 by Kleene,²⁴ and Turing machines, published in 1937 by Turing.³⁶

Lambda calculus was introduced by Church at Princeton, and further developed by his students Rosser and Kleene. At that time, Princeton rivaled Göttingen as a center 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 concise notation for functions, including “first-class” functions that may appear as arguments or results of other functions. It is remarkably compact, containing only three constructs: variables, function abstraction, and function application. Church⁶ at first introduced lambda calculus as a way to define 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 realized 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, the number 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.²³ Once this hurdle was overcome, Church and his students soon became convinced 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 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. 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. Kleene took notes and published the definition.²⁴ 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 his own definition was correct! Things stood at an impasse.

Meanwhile, at Cambridge, 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 it to show the Entscheidungsproblem undecidable. Before the paper was published, Newman was dismayed to discover 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 his paper³⁶ appeared in print a year after Church’s, when Turing was 23. Newman arranged for Turing to travel to Princeton, where



Whereas Church merely presented the definition of λ -definability and baldly claimed it corresponded to effective calculability, Turing undertook an analysis of the capabilities of a “computer.”



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 it corresponded to effective calculability, Turing undertook an analysis of the capabilities of a “computer” (at this time, the term referred to 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 9999999999999999 are the same”). Later, Gandy¹⁴ would point out that Turing’s argument amounts to a theorem asserting 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 encoding had to be abandoned because it led to inconsistency. The failure arose for a reason related to Russell’s paradox, namely that the system allowed a predicate to act on itself, and so Church adapted a solution similar to Russell’s, that 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

Whereas self-application in Russell’s logic leads to paradox, self-application in Church’s untyped lambda calculus leads to non-terminating computations. Conversely, Church’s simply typed lambda calculus guarantees every term has a normal form, or corresponds to a computation that halts.

Untyped lambda calculus or typed lambda calculus with a construct for

general recursion (sometimes called a fixpoint operator) permits the definition of any effectively computable function but has a Halting Problem that is unsolvable. Typed lambda calculus without a construct for general recursion has a Halting Problem that is trivial—every program halts!—but cannot define some effectively computable functions. Both kinds of calculus have their uses, depending on the intended application.

Gentzen and the Theory of Proof

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

In 1935, at the age of 25, Gentzen¹⁵ introduced not one but two new formulations of logic—natural deduction and sequent calculus—that became established as the two major systems for formulating a logic and remain so to this day. He showed how to normalize 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 the symbol \exists for the existential quantification introduced by Peano, Gentzen introduced the symbol \forall to denote universal quantification. He wrote implication as $A \supset B$ (if A holds, then B holds), conjunction as $A \& B$ (both A and B hold), and disjunction as $A \vee B$ (at least one of A or B holds).

Gentzen's insight was that proof rules should come in pairs, a feature not 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 noted, “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 normalized 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 normalized 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 Principle. An immediate consequence was consistency. It is a contradiction to prove false, written \perp . The only way to derive a contradiction is to prove, say, both $A \supset \perp$ and A for some formula A . But given such a proof, one could normalize it to one containing only subformulas of its conclusion, \perp . But \perp has no subformulas! It is like the old saw, “What part of no don't you understand?” Logicians became interested in normalization 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 mainly as a technical device for proving the Subformula Principle, though it has independent interest. It is an irony that Gentzen was required to introduce Sequent Calculus in order to prove the Subformula Principle 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 Principle directly, by introducing a way to simplify Natural Deduction proofs; and this set the ground for Howard's work described in the next section.

Propositions as Types

In 1934, Curry observed a curious fact, relating a theory of functions to a theory of implication.¹¹ Every type of a function ($A \rightarrow B$) 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 the corresponding type.

In 1969, Howard circulated a xeroxed manuscript;²² it was not published until 1980, where it appeared in a *Festschrift* dedicate to Curry. Motivated by Curry's

observation, Howard pointed out there is a similar correspondence between natural deduction, on the one hand, and simply typed lambda calculus, on the other, and he made explicit the third and deepest level of the correspondence, as described in the introduction, that simplification of proofs corresponds to evaluation of programs. Howard showed 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.* Conjunction $A \& B$ corresponds to Cartesian product $A \times B$, or 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.* Disjunction $A \vee B$ corresponds to a disjoint sum $A + B$, or 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.* Implication $A \supset B$ corresponds to function space $A \rightarrow B$. 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 $A \rightarrow B$ 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 formalized intuitionistic logic and developed the interpretation in the 1920s and 1930s. Realizability, introduced by Kleene in the 1940s, is based on a similar interpretation.

Given the intuitionistic reading of proofs, it hardly seems surprising that intuitionistic natural deduction and lambda calculus should correspond so closely. But it was not until Howard that the correspondence was laid out clearly, in a way that allowed working logicians and computer scientists to put it to use.

Howard's paper²² 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 , $+$, \rightarrow 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 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,²⁶ Bates and Constable's PRL and nuPRL,² and Coquand and Huet's Calculus of Constructions,⁹ which developed into the Coq proof assistant.

Applications include CompCert, a certified compiler for the C programming language verified in Coq; a computer-checked proof of the four-color theorem also verified in Coq; parts of the Ensemble distributed system verified in NuPRL; and 20,000 lines of browser plug-ins verified in F*.

de Bruijn's work was independent of Howard's, but Howard directly inspired Martin-Löf and all the other work listed earlier. Howard was (justly!) proud of his paper, citing it as one of the two great achievements of his career.³⁴

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 formalized 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.

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.

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. For this reason,

the second-order lambda calculus was discovered twice, once by the logician Girard¹⁶ and once by the computer scientist Reynolds.³³ And for the same reason, a similar system that supports principle type inference was also discovered twice, once by the logician Hindley²⁰ and once by the computer scientist Milner.²⁷ Building on the correspondence, Mitchell and Plotkin²⁸ observed existential quantification in second-order logic corresponds precisely to data abstraction, an idea that now underpins much research in the semantics of programming languages. The design of generic types in Java and C# draws directly upon Girard-Reynolds, while the type systems of functional languages, including ML and Haskell, are based on Hindley-Milner. 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 two decades later was the correspondence extended to also apply to classical logic, when Griffin¹⁹ observed that Peirce's Law in classical logic provides a type for the `call/cc` operator of Scheme. Murthy³¹ 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 by both semanticists and implementers of lambda calculus. Parigot,³² Curien and Herbelin,¹⁰ and Wadler³⁹ introduced various computational calculi motivated by correspondences to classical logic.

Modal logic permits propositions to be labeled as "necessarily true" or "possibly true." Clarence Lewis introduced modal logic in 1910, and his 1938 textbook²⁵ describes five variants, S1–S5. Some claim each of these variants has an interpretation as a form of computation via Propositions as Types, and a down payment on this claim is given by an interpretation of S4 as staged computation due to Davies and Pfenning,¹² and of S5 as spatially distributed computation due to Murphy et al.³⁰

Moggi²⁹ introduced monads as a

Figure 1. Gerhard Gentzen (1935)—Natural Deduction.

$$\begin{array}{c}
 \frac{A \quad B}{A \& B} \&-I \quad \frac{A \& B}{A} \&-E_1 \quad \frac{A \& B}{B} \&-E_2 \\
 \\
 \frac{[A]^x \quad \vdots \quad B}{A \supset B} \supset-I^x \quad \frac{A \supset B \quad A}{B} \supset-E
 \end{array}$$

Figure 2. A proof.

$$\frac{\frac{[B \& A]^z}{A} \&-E_2 \quad \frac{[B \& A]^z}{B} \&-E_1}{A \& B} \&-I \quad \frac{A \& B}{(B \& A) \supset (A \& B)} \supset-I^z$$

Figure 3. Simplifying proofs.

$$\begin{array}{c}
 \frac{\frac{A \quad B}{A \& B} \&-I}{A} \&-E_1 \Rightarrow A \\
 \\
 \frac{[A]^x \quad \vdots \quad B}{A \supset B} \supset-I^x \quad \frac{A \supset B \quad A}{B} \supset-E \Rightarrow B
 \end{array}$$

Figure 4. Simplifying a proof.

$$\begin{array}{c}
 \frac{[B \& A]^z}{A} \&-E_2 \quad \frac{[B \& A]^z}{B} \&-E_1}{A \& B} \&-I \quad \frac{B \quad A}{B \& A} \&-I \\
 \frac{A \& B}{(B \& A) \supset (A \& B)} \supset-I^z \quad \frac{B \& A}{B} \&-E_2}{A \& B} \supset-E \\
 \\
 \Downarrow \\
 \frac{\frac{B \quad A}{B \& A} \&-I}{A} \&-E_2 \quad \frac{\frac{B \quad A}{B \& A} \&-I}{B} \&-E_1}{A \& B} \&-I \\
 \\
 \Downarrow \\
 \frac{A \quad B}{A \& B} \&-I
 \end{array}$$

technique to explain the semantics of important features of programming languages such as state, exceptions, and input-output. Monads became widely adopted in the functional language Haskell and later migrated into other languages, including Clojure, Scala, F#, and C#. Benton et al.³ observed that monads correspond to yet another modal logic, differing from all of S1–S5.

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 the world where situations change. Computational aspects of linear logic are discussed by Abramsky¹ and Wadler,³⁸ among many others. Most recently, Session Types, a way of describing communication protocols introduced by Honda,²¹ have been related to intuitionistic linear logic by Caires and Pfenning,⁴ and to classical linear logic by Wadler.⁴⁰

Propositions as Types remains a topic of active research.

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 Gentzen¹⁵; 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, labeled $\&-I$ 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, $\&-E_1$, states that if the formula $A \& B$ holds, then the formula A must hold as well. The second, $\&-E_2$, 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 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 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; as mentioned earlier, this rule also goes by the name modus ponens.

Critical readers will observe 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 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; that is, 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 it can be constructed from the rules we posit.

The proof reads as follows. From $B \& A$ we conclude A , by $\&$ -E2, and from $B \& A$ we also conclude B , by $\&$ -E1. 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.

Some proofs are unnecessarily roundabout. Rules for simplifying proofs appear in Figure 3, and an example appears in Figure 4. Let us focus on the example first.

The top of Figure 4 shows a larger proof built from the proof in Figure 2. The larger proof assumes as premises two formulas, B and A , and concludes with the formula $A \& B$. However, rather

Figure 5. Alonzo Church (1935)—Lambda Calculus.

$$\begin{array}{c} \frac{M:A \quad N:B}{\langle M, N \rangle : A \times B} \times\text{-I} \quad \frac{L:A \times B}{\pi_1 L:A} \times\text{-E}_1 \quad \frac{L:A \times B}{\pi_2 L:B} \times\text{-E}_2 \\ \\ \frac{[x:A]^x \quad \vdots \quad N:B}{\lambda x. N : A \rightarrow B} \rightarrow\text{-I}^x \quad \frac{L:A \rightarrow B \quad M:A}{LM:B} \rightarrow\text{-E} \end{array}$$

Figure 6. A program.

$$\frac{\frac{[z:B \times A]^z}{\pi_2 z:A} \times\text{-E}_2 \quad \frac{[z:B \times A]^z}{\pi_1 z:B} \times\text{-E}_1}{\langle \pi_2 z, \pi_1 z \rangle : A \times B} \times\text{-I} \quad \frac{\lambda z. \langle \pi_2 z, \pi_1 z \rangle : (B \times A) \rightarrow (A \times B)}{\lambda z. \langle \pi_2 z, \pi_1 z \rangle : (B \times A) \rightarrow (A \times B)} \rightarrow\text{-I}^z$$

Figure 7. Evaluating programs.

$$\begin{array}{c} \frac{\frac{M:A \quad N:B}{\langle M, N \rangle : A \times B} \times\text{-I}}{\pi_1 \langle M, N \rangle : A} \times\text{-E}_1 \quad \Rightarrow \quad M:A \\ \\ \frac{[x:A]^x \quad \vdots \quad N:B}{\lambda x. N : A \rightarrow B} \rightarrow\text{-I}^x \quad \frac{\vdots \quad M:A}{M:A} \rightarrow\text{-E} \quad \Rightarrow \quad N[M/x]:B \end{array}$$

Figure 8. Evaluating a program.

$$\begin{array}{c} \frac{\frac{[z:B \times A]^z}{\pi_2 z:A} \times\text{-E}_2 \quad \frac{[z:B \times A]^z}{\pi_1 z:B} \times\text{-E}_1}{\langle \pi_2 z, \pi_1 z \rangle : A \times B} \times\text{-I} \quad \frac{y:B \quad x:A}{\langle y, x \rangle : B \times A} \times\text{-I} \\ \frac{\lambda z. \langle \pi_2 z, \pi_1 z \rangle : (B \times A) \rightarrow (A \times B)}{\lambda z. \langle \pi_2 z, \pi_1 z \rangle : (B \times A) \rightarrow (A \times B)} \rightarrow\text{-I}^z \quad \frac{\langle y, x \rangle : B \times A}{\langle y, x \rangle : B \times A} \times\text{-E} \\ \frac{\lambda z. \langle \pi_2 z, \pi_1 z \rangle : (B \times A) \rightarrow (A \times B) \quad \langle y, x \rangle : B \times A}{(\lambda z. \langle \pi_2 z, \pi_1 z \rangle) \langle y, x \rangle : A \times B} \rightarrow\text{-E} \\ \\ \Downarrow \\ \frac{\frac{y:B \quad x:A}{\langle y, x \rangle : B \times A} \times\text{-I}}{\pi_2 \langle y, x \rangle : A} \times\text{-E}_2 \quad \frac{\frac{y:B \quad x:A}{\langle y, x \rangle : B \times A} \times\text{-I}}{\pi_1 \langle y, x \rangle : B} \times\text{-E}_1 \\ \frac{\pi_2 \langle y, x \rangle : A \quad \pi_1 \langle y, x \rangle : B}{\langle \pi_2 \langle y, x \rangle, \pi_1 \langle y, x \rangle \rangle : A \times B} \times\text{-I} \\ \\ \Downarrow \\ \frac{x:A \quad y:B}{\langle x, y \rangle : A \times B} \times\text{-I} \end{array}$$

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 -E.

We may simplify the proof by applying the rewrite rules of Figure 3. These rules specify how to simplify a proof when an introduction rule is immediately followed 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 that combine to yield $A \& B$ by $\&$ -I, which in turn yields A by $\&$ -E₁. 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 $\&$ -E₂.

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 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 that 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 that money, by being difficult 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 $\&$ -E₂, and another instance of $\&$ -I followed by $\&$ -E₁. 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 difficult to see that proofs in normal form satisfy the Subformula Principle: 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 $A \& B$.

Lambda Calculus

We now turn our attention to the simply typed lambda calculus of Church⁸; the type rules are 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 placeholders for arbitrary terms. Product types are written $A \times B$, and function types are written $A \rightarrow B$. Now, instead of formulas, our premises and conclusions are judgments of the form

$$M : A$$

indicating term M has type A .

Like proofs, 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 judgments written above a line, called the “premises,” and a single judgment written below the line, called the “conclusion.” The interpretation of a rule is that when all the premises hold, then the conclusion follows.

Like proof rules, type rules come in pairs. An introduction rule describes how to *define* or *construct* a term of the given type, while 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 -E₁, states that if term L has type $A \times B$, then we may form the term $\pi_1 L$ of type A , which selects the first component of the pair. The second, \times -E₂ is similar, save that it forms the term $\pi_2 L$ of type B .

The introduction rule for functions, \rightarrow -I, states that if given a variable x of type A we have formed 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 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 LM of type B .

For natural deduction, we noted earlier there might be confusion between implication at the meta level and at 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, then 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 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 coloring of the rules is chosen to highlight the similarity.

A program of type

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

is shown in Figure 6. Whereas the difference between $B \& A$ and $A \& B$ appears a mere formality, the difference between

$B \times A$ and $A \times B$ 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 $\pi_2 z$ of type A by $\times\text{-}E_2$ and also term $\pi_1 z$ of type B by $\times\text{-}E_1$. From these two terms we form the pair $\langle \pi_2 z, \pi_1 z \rangle$ of type $A \times B$ by $\times\text{-}I$. Finally, we bind the free variable z to form the lambda term $\lambda z. \langle \pi_2 z, \pi_1 z \rangle$ of type $(B \times A) \rightarrow (A \times B)$ by $\rightarrow\text{-}I$, 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.

A program may be evaluated by rewriting. Rules for evaluating programs appear in Figure 7, and an example appears in Figure 8. Let us focus on the example first.

The top of Figure 8 shows a larger program built from the program in Figure 6. The larger 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 $\rightarrow\text{-}E$, 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 $\times\text{-}I$. Applying the function to the pair forms a term of type $A \times B$ by $\rightarrow\text{-}E$.

We may evaluate this program by applying the rewrite rules of Figure 7. These rules specify how to rewrite a term when an introduction rule is immediately followed by the corresponding elimination rule. Each rule shows two derivations connected by an arrow, indicating the “redex” (the term on the left) may be rewritten, or evaluated, to yield the “reduct” (the term on the right). Rewrites always take a valid type derivation to another valid type derivation, ensuring 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 that combine to yield term $\langle M, N \rangle$ of type $A \times B$ by $\times\text{-}I$, which in turn yields term $\pi_1 \langle M, N \rangle$ of type A by $\times\text{-}E_1$. 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 rewrite an occurrence of $\times\text{-}I$ followed by $\times\text{-}E_2$.

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 $\rightarrow\text{-}I$, and a derivation of term M of type A , which combine to yield the application $(\lambda x. N) M$ of type B by $\rightarrow\text{-}E$. The reduct consists of the term $N[M/x]$, which replaces each free occurrence of the variable x in term N by term M . Further, if in the derivation that N has type B we replace each assumption that x has type A by the derivation that M has type A , we get a derivation showing $N[M/x]$ 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 it has removed a subterm of type $A \rightarrow B$. Discharge of assumptions thus corresponds to applying a function to its argument.

Figure 8 demonstrates use of these rules to evaluate a program. The first program contains an instance of $\rightarrow\text{-}I$ followed by $\rightarrow\text{-}E$, and is rewritten 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 $\times\text{-}I$ followed by $\times\text{-}E_2$, and another instance of $\times\text{-}I$ followed by $\times\text{-}E_1$. Rewriting each of these yields the third program, which derives the term $\langle x, y \rangle$ of type $A \times B$, and can be evaluated 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.

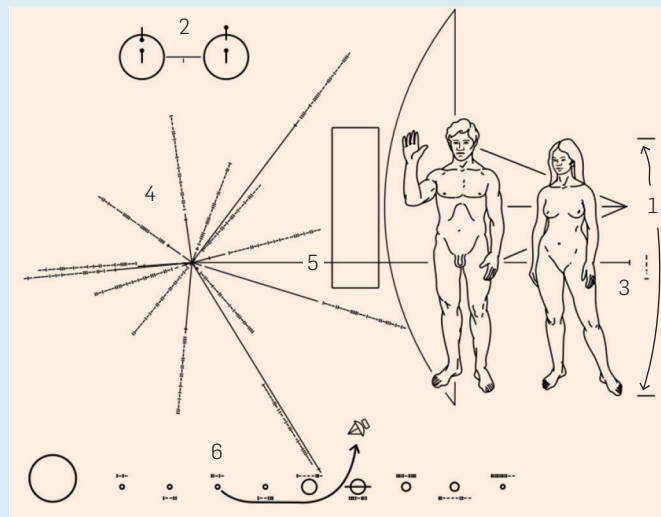
Conclusion

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

The *Pioneer* spaceship includes 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 14 pulsars and the center of our galaxy from Sol. Aliens are likely to determine 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’ 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 mothership by infecting

Figure 9. Plaque on *Pioneer* spaceship.




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 unclear that aliens could decipher a program written in C if presented with one.

What about lambda calculus? Propositions as Types tell us lambda calculus is isomorphic to natural deduction. It seems difficult to conceive of alien beings who 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 understanding the radial diagram of pulsars than of understanding the image of a man and a woman on the *Pioneer* plaque.

We might be tempted to conclude lambda calculus is universal, but first 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). 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 characterize lambda calculus as a universal language, because calling it universal would be *too limiting*.

Acknowledgments

Thank you to Gershon Bazerman, Pete Bevin, Guy Blelloch, Rintcius Blok, Ezra Cooper, Ben Darwin, Benjamin Denckla, Peter Dybjer, Johannes Emerich, Martin Erwig, Yitz Gale, Mikhail Glushenkov, Gabor Greif, Vinod Grover, Sylvain Henry, Philip Hölzenspies, William Howard, John Hughes, Colin Lupton, Daniel Marsden, Craig McLaughlin, Tom Moertel, Simon Peyton-Jones, Benjamin Pierce, Lee Pike, Andrés Sicard-Ramírez, Scott Rostrup, Dann Toliver, Moshe Vardi, Jeremy Yallop, Richard Zach, Leo Zovik, and the referees. This work was funded under Engineering and Physical Sciences Research Council grant EP/K034413/1. 

References

- Abramsky, S. Computational interpretations of linear logic. *Theoretical Computer Science* 111, 1–2 (1993), 3–57.
- Bates, J.L., Constable, R.L. Proofs as programs. *Transactions on Programming Languages and Systems* 7, 1 (Jan. 1985), 113–136.
- Benton, P.N., Bierman, G.M., de Paiva, V. Computational types from a logical perspective. *Journal of Functional Programming* 8, 2 (1998), 177–193.
- Caires, L., Pfenning, F. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory* (Paris, France, Aug. 31–Sept. 3, 2010), 222–236.
- Carroll, L. What the Tortoise said to Achilles. *Mind* 4, 14 (Apr. 1895), 278–280.
- Church, A. A set of postulates for the foundation of logic. *Ann. Math.* 33, 2 (1932), 346–366.
- Church, A. An unsolvable problem of elementary number theory. *American Journal of Mathematics* 58, 2 (Apr. 1936), 345–363; presented to the American Mathematical Society, Apr. 19, 1935; abstract in *Bulletin of the American Mathematical Society* 41 (May 1935).
- Church, A. A formulation of the simple theory of types. *Journal of Symbolic Logic* 5, 2 (June 1940), 56–68.
- Coquand, T. and Huet, G.P. The calculus of constructions. *Information and Computation* 76, 2/3 (1988), 95–120.
- Curien, P.-L., Herbelin, H. The duality of computation. In *Proceedings of the International Conference on Functional Programming* (Montreal, Canada, Sept. 18–20). ACM Press, New York, 2000, 233–243.
- Curry, H.B. Functionality in combinatory logic. *Proceedings of the National Academy of Science* 20 (1934), 584–590.
- Davies, R., Pfenning, F. A modal analysis of staged computation. In *Principles of Programming Languages* (St. Petersburg Beach, FL, 1996), 258–270.
- de Bruijn, N.G. The mathematical language Automath, its usage, and some of its extensions. In *Proceedings of the Symposium on Automatic Demonstration, Volume 125 of Lecture Notes in Computer Science* (Versailles, France, Dec.). Springer-Verlag, 1968, 29–61.
- Gandy, R. The confluence of ideas in 1936. In *The Universal Turing Machine: A Half-century Survey*, R. Herken, Ed. Springer, 1995, 51–102.
- Gentzen, G. Untersuchungen über das logische Schließen. *Math. Z.* 39, 2–3 (1935), 176–210, 405–431; reprinted in Szabo.³⁵
- Girard, J.Y. Interprétation fonctionnelle et élimination des coupures dans l’arithmétique d’ordre supérieure. These D’Etat, Université Paris VII, 1972.
- Girard, J.-Y. Linear logic. *Theoretical Computer Science* 50 (1987), 1–102.
- Gödel, K. Über formal unterscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik* 38 (1931), 173–198; reprinted in Heijenoort.³⁷
- Griffin, T. A formulae-as-types notion of control. In *Proceedings of the 40th Annual Symposium on Principles of Programming Languages* (Rome, Italy, Jan. 23–25). ACM Press, New York, 1990, 47–58.
- Hindley, R. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society* 146 (Dec. 1969), 29–60.
- Honda, K. Types for dyadic interaction. In *Proceedings of the Fourth International Conference on Concurrency Theory* (Hildesheim, Germany, Aug. 23–26, 1993), 509–523.
- Howard, W.A. The formulae-as-types notion of construction. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980, 479–491; original version was circulated privately in 1969.
- Kleene, S. Origins of recursive function theory. *Annals of the History of Computing* 3, 1 (1981), 52–67.
- Kleene, S.C. General recursive functions of natural numbers. *Mathematical Annalen* 112, 1 (Dec. 1936); abstract in *Bulletin of the AMS* (July 1935).
- Lewis, C. and Langford, C. *Symbolic Logic*, 1938; reprinted by Dover, 1959.
- Martin-Löf, P. *Intuitionistic Type Theory*. Bibliopolis, Naples, Italy, 1984.
- Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences* 17, 3 (1978), 348–375.
- Mitchell, J.C., Plotkin, G.D. Abstract types have existential type. *Transactions on Programming Languages and Systems* 10, 3 (July 1988), 470–502.
- Moggi, E. Notions of computation and monads. *Information and Computation* 93, 1 (1991), 55–92.
- Murphy VII, T., Cray, K., Harper, R., Pfenning, F. A symmetric modal lambda calculus for distributed computing. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science* (Turku, Finland, July 13–17). IEEE Press, 2004, 286–295.
- Murthy, C. An evaluation semantics for classical proofs. In *Proceedings of Sixth Annual IEEE Symposium on Logic in Computer Science* (Amsterdam, the Netherlands, July 15–18). IEEE Press, 1991, 96–107.
- Parigot, M. $\lambda\mu$ -calculus: An algorithmic interpretation of classical natural deduction. In *Logic Programming and Automated Reasoning, Volume 624 of Lecture Notes in Computer Science*. Springer-Verlag, 1992, 190–201.
- Reynolds, J.C. Towards a theory of type structure. In *Proceedings of the Symposium on Programming, Volume 19 of Lecture Notes in Computer Science* (1974), 408–423.
- Shell-Gellasch, A.E. Reflections of my advisor: Stories of mathematics and mathematicians. *The Mathematical Intelligencer* 25, 1 (2003), 35–41.
- Szabo, M.E., Ed. *The Collected Papers of Gerhard Gentzen*. North Holland Publishing Co., Amsterdam, the Netherlands, 1969.
- Turing, A.M. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* s2–42, 1 (1937); received May 28, 1936, read Nov. 12, 1936.
- van Heijenoort, J. *From Frege to Gödel: A Sourcebook in Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge, MA, 1967.
- Wadler, P. A taste of linear logic. In *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science Volume 711 of Lecture Notes on Computer Science* (Gdańsk, Poland, Aug. 30–Sept. 3). Springer-Verlag, 1993, 185–210.
- Wadler, P. Call-by-value is dual to call-by-name. In *Proceedings of the International Conference on Functional Programming* (Uppsala, Sweden, Aug. 25–29). ACM Press, New York, 2003, 189–201.
- Wadler, P. Propositions as sessions. In *Proceedings of the International Conference on Functional Programming* (Copenhagen, Denmark, Sept. 10–12). ACM Press, New York, 2012, 273–286.

Philip Wadler (wadler@inf.ed.ac.uk) is Professor of Theoretical Computer Science in the Laboratory for Foundations of Computer Science in the School of Informatics at the University of Edinburgh, Scotland.

Copyright held by authors.
Publication rights licensed to ACM. \$15.00.