Richard Zach

The Significance of the Curry-Howard Isomorphism

Abstract: The Curry-Howard isomorphism is a proof-theoretic result that establishes a connection between derivations in natural deduction and terms in typed lambda calculus. It is an important proof-theoretic result, but also underlies the development of type systems for programming languages. This fact suggests a potential importance of the result for a philosophy of code.

1 Introduction

Many results of mathematical logic are thought to be of philosophical significance. The most prominent examples are, perhaps, Gödel's completeness and incompleteness theorems, and the Löwenheim-Skolem theorem. The completeness theorem is thought to be significant because it establishes, in a mathematically rigorous way, the equivalence between syntactic and semantic definitions of logical consequence. The incompleteness theorem, on the other hand, is significant because it shows the *in*equivalence of syntactic and semantic definitions of mathematical truth. The Löwenheim-Skolem theorem is considered philosophically significant especially because of the use Putnam put it to in his model-theoretic argument (Putnam 1980). I will argue below that the Curry-Howard isomorphism is a logical result that promises to be philosophically significant, even though (among philosophical logicians, at least) it is little known and if it is, is often considered a mere curiosity. That result, in brief, is this: to every derivation in natural deduction there corresponds a term in a lambda calculus, such that transformation of the derivation into a normal form corresponds to evaluating the corresponding lambda term.

Of course, results of mathematical logic, just like mathematical results more generally, are often thought to be significant not just for philosophy, but more generally. The question of what makes a mathematical result significant is a fruitful topic for philosophical investigation, and I will not be able to do it justice here. But a few things can be said. Mathematical results can be significant for a number of different reasons. Significance can arise from the result's theoretical importance. It may, for instance, elucidate a concept by relating it to others. A prime example of this is again the completeness theorem: it elucidates semantic and proof-theoretic

Richard Zach, University of Calgary, Canada, rzach@ucalgary.ca
DOI 10.1515/9783110657883-18

definitions of consequence by showing them to be equivalent (in the case of first-order logic). A result can also be considered significant because it is fruitful in proving other results; it provides methods used in further proofs. The compactness theorem is significant for this reason: it allows us to show that sets of sentences are consistent (and hence satisfiable) by showing that every finite subset is consistent (or satisfiable).

Another dimension along which the significance of results could be measured is the breadth of fields for which it is significant, either by elucidating notions, or by providing proof methods, or by paving the way for "practical" applications. By a "practical" application, I mean a method for finding answers to specific questions or solving particular problems. The line between what counts as a theoretical application and what as a practical application is of course not a clear one. But a paradigm case, I take it, of a practical application is, for instance, determining if a specific inference is valid or finding an interpretation for a specific set of sentences. The specific inferences or sets of sentences here often and importantly may not be problems in mathematical logic itself, but lie in other areas of mathematics or even outside mathematics. Inferences from databases, say, or finite models for formal specifications of circuits can be seen as specific questions that can be solved with methods – proof and model-building methods – from mathematical logic. That such proof methods exist and that they are sound and complete, again, is one reason they are significant. Here, of course, the proof methods are ones that are amenable to implementation in software, such as resolution, and not those commonly used or studied in mathematical logic itself.

Soundness, completeness, and compactness results for various logical systems are easily seen to be significant for these reasons and along these dimensions. Proof-theoretic results such as cut-elimination and consistency proofs more generally are harder to certify as significant. Of course, consistency proofs are one of the more prototypical philosophically significant results of mathematical logic, since they arose directly out of philosophical concerns, viz., Hilbert's program. Normalization of natural deduction plays an important role in the formulation of proof-theoretic semantics.

Consistency proofs are also often practically significant. In the absence of a semantics for a logical formalism, for instance, a consistency proof (e.g., a cut-elimination result) establishes a kind of safety of the formalism. Historically, such results established the safety of various non-classical logics. They also paved the way for the development of formalisms that were amenable to software implementation, i.e., automated theorem proving. Proof search is only feasible if the search space can be sufficiently restricted, and cut-elimination guarantees this. If the cut rule were not eliminable, proof search using analytic calculi such as the sequent

calculus would not be feasible – and cut elimination proofs establish this even in the absence of a proof of cut-free completeness.

Strengthenings of consistency proofs also have theoretical significance in mathematical logic. Here I have in mind the kinds of results arising out of the work of consistency proofs which show that there are procedures that transform proofs in one system into proofs in another, weaker system (at least for certain classes of theorems). These results provide proof theoretic reductions of one system to another (a theoretically significant result inside mathematical logic), provide the basis for what Feferman 1992 has called "foundational reductions" (a philosophical payoff), but also sometimes provides methods for extracting bounds from proofs of existential theorems in various systems of arithmetic or analysis (a mathematical payoff). They also measure (and hence elucidate) the strength of axiom and proof systems in more fine-grained ways than simple consistency strength, as when the proof-theoretic reduction shows a speed-up of one system over another.

The significance of the Curry-Howard isomorphism has been hard to assess. It started off as a mere curiosity, when Curry 1934 observed a perhaps surprising but possibly merely coincidental similarity between some axioms of intuitionistic and combinatory logic. Following Howard 1980 (originally circulated in 1969) and Reynolds 1974, it became clear that the isomorphism applied in a wider variety of cases, and it took on both theoretical and philosophical significance. Its theoretical significance lies in the fact that it can be used to prove strong normalization of natural deduction, i.e., the result that normal derivations do not just exist but that they are unique. Its philosophical significance arose originally out of the claim that the lambda term assigned to a derivation can be taken to be the "computational content" of the derivation. But in what sense this does provide a "content" in any kind of clear and robust sense was never really elucidated. More work has to be done to justify calling the lambda term assigned to a derivation its "content," and to explain in what sense that content is "computational."

In the past two decades or so, however, it has become clear that the Curry-Howard isomorphism is of very clear practical significance: it now forms the basis of type systems of programming languages and the natural deduction systems on the proof side of the isomorphisms are the basis of automatable and indeed automated type checkers and type inference systems for real-life programming languages. I will argue that this practical significance allows for an overdue philosophical study of programming languages.

2 Natural Deduction

The Curry-Howard isomorphism is a correspondence between proofs in natural deduction systems and terms in lambda calculi. The correspondence is explained much more easily if natural deduction is formulated not as it was originally by Gentzen 1934 and Prawitz 1965 as inferring formulas from formulas, but as proceeding from sequents to sequents. In classical natural deduction, initial formulas in a proof are assumptions, some of which may be discharged by certain inference rules. A proof is thought of as establishing that the end-formula follows from the set of assumptions that remain undischarged or "open." In "sequent style" natural deduction, each sequent $\Gamma \vdash A$ in the proof records the set of assumptions that a formula follows from at every step in the proof. So an assumption by itself is recorded as $A \vdash A$, and generally a proof with end-formula A from open assumptions Γ would be represented, in sequent-style natural deduction, as a tree of sequents with end-sequent $\Gamma \vdash A$. In this setup, the inference rules of natural deduction become:

$$\frac{A, \Gamma \vdash B}{\Gamma \vdash A \to B} \to\text{I} \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B} \to\text{E}$$

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \wedge\text{I} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \wedge\text{E} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \wedge\text{E}$$

Of course, this is just the fragment involving the conditional and conjunction, for simplicity. We're also ignoring, for the time being, the subtleties of keeping track of which assumptions are discharged where, but this will be fixed later on. Here is an example derivation of the theorem $(A \wedge B) \to (B \wedge A)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{A \wedge B \vdash A \wedge B}{A \wedge B \vdash B} \wedge\text{E} \qquad \dfrac{A \wedge B \vdash A \wedge B}{A \wedge B \vdash A} \wedge\text{E}}{A \wedge B \vdash B \wedge A} \wedge\text{I}}{\vdash (A \wedge B) \to (B \wedge A)} \to\text{I}$$

It corresponds to the following derivation in the original formulation of natural deduction:

$$\dfrac{\dfrac{\dfrac{\dfrac{[A \wedge B]^x}{B} \wedge\text{E} \qquad \dfrac{[A \wedge B]^x}{A} \wedge\text{E}}{B \wedge A} \wedge\text{I}}{(A \wedge B) \to (B \wedge A)} \to\text{I}\ x$$

The x labelling the $\to$I inference indicates that the assumption labelled x is discharged at that inference. In the sequent-style derivation, all sequents above the

→I inference depend on the assumption $A \wedge B$ and so $A \wedge B$ appears on the left of the turnstile; in the final sequent, that assumption has been discharged and so $A \wedge B$ no longer appears on the left of the turnstile.

Derivations in natural deduction *normalize*, i.e., there is always a sequence of reduction steps which transforms a derivation into a derivation in normal form. A derivation is in normal form if no introduction rule for a connective is immediately followed by an elimination rule for the same connective. This result plays a similar role for natural deduction as the cut-elimination theorem plays for the sequent calculus. In particular, it establishes the subformula property for natural deduction derivations. As noted above, the subformula property is essential for the practical implementation of proof search algorithms, since it limits the search space and so eliminates one source of infeasibility in actually carrying out the search for a derivation.

Normalization is carried out by removing "detours" – applications of introduction rules immediately followed by elimination rules – one by one. In the sequent-style natural deduction formalism sketched above, such a replacement would be, for instance, the following:

$$\dfrac{\dfrac{\begin{matrix}\pi\\ A, \Gamma \vdash B\end{matrix}}{\Gamma \vdash A \to B}\,{\to}\mathrm{I} \qquad \begin{matrix}\pi'\\ \Gamma \vdash A\end{matrix}}{\Gamma \vdash B}\,{\to}\mathrm{E} \qquad \mapsto \qquad \begin{matrix}\pi[\pi'/A]\\ \Gamma \vdash B\end{matrix}$$

Here, $\pi[\pi'/A]$ is the result of replacing, in the subderivation π, every initial sequent of the form $A \vdash A$ by the subderivation π' ending in $\Gamma \vdash A$. Note that no natural deduction rules apply to the left side of sequents (the so-called context). Hence, any occurrence of A in the contexts of sequents in π is replaced by Γ, and the A's in the contexts in π disappear. Thus, if the derivation π has end-sequent $A, \Gamma \vdash B$, then $\pi[\pi'/A]$ has end-sequent $\Gamma \vdash B$.

In the case of a ∧E rule following a ∧I rule, the reduction step is even simpler. Here, nothing has to be done to the original (sub-)derivations.

$$\dfrac{\dfrac{\begin{matrix}\pi\\ \Gamma \vdash A\end{matrix} \qquad \begin{matrix}\pi'\\ \Gamma \vdash B\end{matrix}}{\Gamma \vdash A \wedge B}\,\wedge\mathrm{I}}{\Gamma \vdash A}\,\wedge\mathrm{E} \qquad \mapsto \qquad \begin{matrix}\pi\\ \Gamma \vdash A\end{matrix}$$

Of course, if the conclusion of ∧E had been $\Gamma \vdash B$, the derivation would instead be reduced to just the derivation π' of the sequent $\Gamma \vdash B$.

3 The Typed Lambda Calculus

The Curry-Howard isomorphism is a correspondence between derivations in natural deduction and terms in a typed lambda calculus. But let's begin with the untyped lambda calculus. It is a term calculus; terms are built up from variables. If t is a term, so is $\lambda x.t$, called a lambda abstract. Intuitively, it represents a function (a program) that takes x as argument and returns a value specified by the term t (in which x occurs free). In $\lambda x.t$, x is bound. So terms can represent functions; applying a function to an argument is simply represented by a term (ts), where s is another term representing the argument to the function represented by t. Our toy calculus also contains operations working on pairs: If t and s are terms, then $\langle t, s\rangle$ is the pair consisting of t and s. If t is a term representing a pair, then $\pi_1 t$ represents its first component and $\pi_2 t$ its second.

The lambda calculus is a very simple programming language, in which terms are programs. *Execution* of a program for an input is the conversion of terms to one that cannot be further evaluated. Such terms are said to be in normal form; they represent outcomes of computations. The conversion of terms proceeds according to the following *reduction rules*:

$$\begin{aligned} (\lambda x.t)s &\mapsto t[s/x] \\ \pi_1\langle t, s\rangle &\mapsto t \\ \pi_2\langle t, s\rangle &\mapsto s \end{aligned}$$

The notation $t[s/x]$ means: replace every free occurrence of x in t by s. Terms of the form given on the left are called *redexes*; they are the kinds of (sub)terms to which reduction can be applied.

The reduction rules apply not just to entire terms, but also to subterms. For instance, here is a very simple program that inverts the order of the elements of a pair:

$$\lambda x.\langle \pi_2 x, \pi_1 x\rangle$$

If we apply this term to a pair $\langle u, v\rangle$, conversion will produce $\langle v, u\rangle$:

$$\begin{aligned} (\lambda x.\langle \pi_2 x, \pi_1 x\rangle)\langle u, v\rangle &\mapsto \langle \pi_2\langle u, v\rangle, \pi_1\langle u, v\rangle\rangle \\ &\mapsto \langle v, \pi_1\langle u, v\rangle\rangle \\ &\mapsto \langle v, u\rangle \end{aligned}$$

In the untyped lambda calculus, it is allowed to apply terms to terms to which they intuitively shouldn't be applied. In the simple example above, the program on the left expects the argument x to be a pair, but the formation rules don't prohibit

applying it, e.g., to another function $\lambda y.t'$. In the *typed* lambda calculus, the syntax prohibits this. Now, every variable comes with a type, denoted by an uppercase letter: x^A means that the variable x only takes values of type A. One can think of types as objects of a certain sort (e.g., truth values or natural numbers) but there are also function types (e.g., functions from numbers to truth values). The type $A \to B$ are the functions with arguments of type A and values of type B. There may be other types as well. In our example language, for instance, we have product types: if A and B are types, then $A \wedge B$ is the type consisting of pairs where the first element is of type A and the second of type B. So our example program now becomes $\lambda x^{A\wedge B}.\langle \pi_2 x, \pi_1 x\rangle$: the variable x is restricted to objects of type $A \wedge B$. It is then easy to see that the term represents a function from pairs to pairs, but while arguments are of type $A \wedge B$, values are of type $B \wedge A$. So our term is of type $(A \wedge B) \to (B \wedge A)$. This is represented in a *type judgment*

$$\lambda x^{A\wedge B}.\langle \pi_2 x, \pi_1 x\rangle : (A \wedge B) \to (B \wedge A)$$

Now the point of the typed lambda calculus is to disallow terms in which the types don't match up. For instance, we should not allow the formation of the term

$$(\lambda x^{A\wedge B}.\langle \pi_2 x, \pi_1 x\rangle)(\lambda y^A.y)$$

because the term on the left is a function of type $(A \wedge B) \to (B \wedge A)$ and the term $\lambda y^A.y$ on the right is a term of type $A \to A$, i.e., certainly not a pair. To do this, we give formation rules for terms that take the types of variables and subterms into account. They operate on sequents $\Gamma \vdash t : A$ where Γ is now a set of type judgments for the free variables in the term t. For instance,

$$x : A, y : B \vdash \langle x, y\rangle : A \wedge B$$

means that if x is of type A and y of type B, then the term $\langle x, y\rangle$ is of type $A \wedge B$. So the rules for forming terms now become:

$$\frac{x : A, \Gamma \vdash t : B}{\Gamma \vdash \lambda x^A.t : A \to B} \to\text{I} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma' \vdash s : A}{\Gamma, \Gamma' \vdash (ts) : B} \to\text{E}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash s : B}{\Gamma \vdash \langle t, s\rangle : A \wedge B} \wedge\text{I} \qquad \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash \pi_1 t : A} \wedge\text{E}$$

$$\frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash \pi_2 t : B} \wedge\text{E}$$

Of course, $x : A \vdash x : A$ is always true, and so sequents of this form serve as axioms.

4 The Curry-Howard Isomorphism

The first part of the Curry-Howard isomorphism consists in the observation that the right-hand sides of the type judgments in the rules of term formation for the typed lambda calculus given in the previous section are exactly the introduction and elimination rules of natural deduction. When considered as term formation and type inference rules, the →E rule, for instance, tells us that if s is a term of type $A \to B$ and t is a term of type A, then (st) is a term of type B. However, these same rules also allow us to systematically assign terms to sequents in a natural deduction derivation (once variables are assigned to the assumptions). If we focus not on the term side of the type judgments but on the type (formula) side, we can read the inference rules another way. For instance, in the case of →E, they say that if s has been assigned to the premise $A \to B$, and t to the premise A, then we should assign (st) to the conclusion B. In this way, we get that for every derivation in natural deduction we can assign terms from the typed lambda calculus to each sequent (or rather, the formula on the right of the sequent). For instance, here is a derivation of $(A \wedge B) \to (B \wedge A)$ with the corresponding terms assigned to each formula on the right; the assignment is determined once we pick variables to assign to the assumptions (in this case, we assign x to the assumption $A \wedge B$):

$$\dfrac{\dfrac{\dfrac{\dfrac{x : A \wedge B \vdash x : A \wedge B}{x : A \wedge B \vdash \pi_2 x : B}\wedge\mathrm{E} \quad \dfrac{x : A \wedge B \vdash x : A \wedge B}{x : A \wedge B \vdash \pi_1 x : A}\wedge\mathrm{E}}{x : A \wedge B \vdash \langle \pi_2 x, \pi_1 x\rangle : B \wedge A}\wedge\mathrm{I}}{\vdash \lambda x^{A\wedge B}.\langle \pi_2 x, \pi_1 x\rangle : (A \wedge B) \to (B \wedge A)}\to\mathrm{I}$$

The second part of the Curry-Howard isomorphism extends this observation. Applying the normalization procedure to derivations in natural deduction corresponds to evaluation (reduction) of the corresponding terms. In the case where we remove a →I/→E detour, the term assigned to the conclusion is of the form $(\lambda x.t)s$ and the term assigned to the conclusion of the derivation after applying the normalization step is $t[s/x]$, i.e., the result of a lambda calculus reduction step:

$$\dfrac{\dfrac{\overset{\pi}{x : A, \Gamma \vdash t : B}}{\Gamma \vdash \lambda x^A.t : A \to B}\to\mathrm{I} \quad \overset{\pi'}{\Gamma \vdash s : A}}{\Gamma \vdash (\lambda x^A.t)s : B}\to\mathrm{E} \qquad \mapsto \qquad \begin{array}{c}\pi[\pi'/A]\\ \Gamma \vdash t[s/x] : B\end{array}$$

The same happens when we remove a ∧I/∧E detour:

$$\dfrac{\dfrac{\overset{\pi}{\Gamma \vdash t : A} \quad \overset{\pi'}{\Gamma \vdash s : B}}{\Gamma \vdash \langle t, s\rangle : A \wedge B}\wedge\mathrm{I}}{\Gamma \vdash \pi_1\langle t, s\rangle : A}\wedge\mathrm{E} \qquad \mapsto \qquad \begin{array}{c}\pi\\ \Gamma \vdash t : A\end{array}$$

Note in particular that a derivation in natural deduction is normal (contains no detours) if and only if the term assigned to it is in normal form: any →I/→E detour would correspond to a redex of the form $(\lambda x^A.t)s$, and any ∧I/∧E detour would correspond to a redex of the form $\pi_i\langle s, t\rangle$. (A redex, again, is a subterm to which a reduction can be applied; a term is normal if it contains no redexes.)

More detailed expositions of the Curry-Howard isomorphism for more comprehensive systems can be found in Girard et al. 1989 and Sørensen/Urzyczyn 2006.

5 The Significance of the Curry-Howard Isomorphism

In its logical form, the Curry-Howard isomorphism consists of the following facts:

1. Natural deduction proofs have associated proof terms.
2. Normalization corresponds to reduction of the corresponding proof terms.

We have only seen it at work in the toy case of the →/∧-fragment of minimal logic, but similar proof term assignments can and have been given for a wide variety of natural deduction systems and logics. In this version, the Curry-Howard isomorphism is theoretically important for one main reason: it allows us to prove what's called strong normalization. It is relatively easy to prove that there always is a sequence of normalization steps that results in a derivation in normal form. Strong normalization is the claim that *any* sequence of reduction steps (a) terminates and (b) results in the same normal form. Via the Curry-Howard correspondence, this statement about derivations and normalization is a consequence of a corresponding result about lambda calculus and evaluation of typed lambda terms: any sequence of evaluation steps (a) terminates in a term in normal form (a value) and (b) all possible sequences of evaluation steps result in the same value.

The Curry-Howard isomorphism, in its computational version, is the basis for a very important body of work developed over the last 30 years, in the area of theorem proving and programming language theory. The computational version results from considering the term (program) side of the isomorphism as primary:

1. Well-formed lambda terms have associated type derivations.
2. Evaluation of terms corresponds to normalization of the corresponding type derivations.

Its importance for the theory of programming languages lies in the fact that it provides a model for performing type checking for programs in various programming languages. In our toy example, the typed lambda calculus plays the role of the programming language. A term of the form $\lambda x^A.t$ is a program which takes inputs of type A. Its outputs are of type B just in case the term itself is of type $A \to B$. To type check the program means to ensure that the term has this type. It obviously corresponds exactly to the proof-theoretic question of whether there is a derivation of $\vdash \lambda x^A.t : A \to B$, i.e., whether there is a derivation of $A \to B$ which has the type $\lambda x^A.t$ assigned to it as a proof term. The Curry-Howard isomorphism establishes that well-formed terms can always be type checked by a derivation in normal form. Since derivations in normal form have the subformula property, searching for a derivation that type checks a given program can (often) be done effectively.

A programming language is called type safe if it prevents programmers from writing programs that result in type errors. A type error occurs if a program of type $A \to B$, say, is applied to an argument which is not of type A, or when its evaluation for an argument of type A results in something that is not of type B. The Curry-Howard isomorphism, and properties like it for actual programming languages, guarantees type safety. This has tremendous theoretical and practical importance: programs in type-safe languages cannot hang: there is always a way to continue the execution until a value is arrived at. The Curry-Howard isomorphism has this consequence because it implies two things: If a term is not already a value (i.e., it is not in normal form), then evaluation can continue because it contains a redex that can be reduced. This property is called "progress." But more importantly – and here the Curry-Howard isomorphism comes in – when a term is reduced, its type stays the same (and there is a corresponding natural deduction derivation which verifies this). This property is called "type preservation." Together, these two properties establish that if t is a term (program) of type $A \to B$, and s is a term (argument) of type A, (ts) will always evaluate to a normal term (value) of type B. The strong normalization property for the typed lambda calculus establishes that evaluation always terminates and that any order of evaluation steps results in the same value. The Curry-Howard isomorphism, via its consequence of type preservation, establishes that each intermediate step and the final result, will be a term of type B.

The ability to type check programs, and the property of type safety, are kinds of completeness and soundness properties for programs. Like consistency in the foundations of mathematics, type safety provides a minimal safety guarantee for programs in type-safe languages. Type safety does not, of course, guarantee that any program will output the specific desired result – it is still possible that there are programming errors – but it does guarantee that the program will not produce

a result of the wrong type. It won't hang, and it won't produce, say, a number when the program is of type, say, natural numbers to Boolean values.[1]

6 Toward a Philosophy of Code

For a long time, debates in the philosophy of mathematics were dominated by questions of little or no significance or even relation to mathematical practice, e.g., the realism/anti-realism debate. Concurrently, philosophy of science, and philosophies of the special sciences, concentrated on questions that were of quite central and immediate relevance to their respective areas. At the same time, computer science developed as an independent discipline. Philosophers of mathematics, however, have focused on just a very narrow set of questions related to computability – mainly those related to the limits of computability, the status of the Church-Turing thesis, philosophical analyses of the notion of mechanical computation, and the relation between formal and physical models of computability. Philosophy of programs and higher level programming languages is just in its infancy.

By analogy with the philosophy of mathematics, logical results like the Curry-Howard isomorphism promise to play a similar role in such a nascent philosophy of code as earlier results like Gödel's incompleteness theorems, the Löwenheim-Skolem theorem, and consistency proofs and proof-theoretic reductions play in the philosophy of mathematics. It can help frame and solve philosophical questions that naturally arise about programming languages. One such question, for instance, is this: Although several foundational frameworks are available for mathematics, one of them, Zermelo-Fraenkel set theory, has dominated both in mathematics and in the philosophy of mathematics. By contrast, many more frameworks for specifying algorithms have been and are being proposed by computer scientists. What explains this? What are the fundamental differences between programming languages and according to which criteria should we compare them? There are obvious candidates, such as efficiency or suitability for specific applications. But to differentiate programming languages and programming paradigms it will be important to consider the conceptual differences between languages, such as the presence or absence, and the power and structure of their type systems.

Recent work in the philosophy of mathematics has made headway on some aspects of mathematics that were bracketed by traditional mid-century philosophical engagement with mathematics, and which are more closely related to the

1 For more detail on the importance of the Curry-Howard isomorphism and type systems for programming languages, see Cardelli 2004, Pierce 2002, Wadler 2015.

practice of mathematics and to methodological considerations made by mathematicians themselves. For instance, philosophers have considered the questions of what makes a proof explanatory (Mancosu 2018), the notions of elegance and beauty as employed by mathematicians (Montano 2014), differences in mathematical style (Mancosu 2017), and the question of when proof methods count as "pure" (e.g. Detlefsen/Arana 2011). By analogy, methodological considerations made by computer scientists in proposing, designing, revising, criticizing, and choosing between programming languages provide an area suitable for philosophical analysis and reflection – which may well be of interest to computer scientists the way, say, some work in the philosophy of biology has been of interest in biology itself. Computer scientists also talk about programs and programming languages in aesthetic (elegance), cognitive (readability), or practical terms (efficiency and speed, ease of maintenance, security) which play a role in the above-mentioned comparison between and design of programming languages. Type-safe languages, for instance, have been claimed to be superior to non-type-safe and untyped languages along these lines. They have been claimed to be easier to read, easier to maintain, easier to debug, and they are, demonstrably, more secure. It is also claimed that typed languages are more abstract, i.e., that they enable programming at a higher level of abstraction. This notion of abstraction itself could be subjected to philosophical analysis. In all of this, a logical result – the Curry-Howard isomorphism – plays a fundamental role, and it would be important to understand this role better, and to make use of it for philosophical discussions of code and programming languages.

Bibliography

Cardelli, Luca (2004): "Type Systems." In: *Computer Science and Engineering Handbook*. Tucker, Allen B. (ed.), 2nd edition, chapter 97. Boca Raton, FL: CRC Press

Curry, Haskell B. (1934): "Functionality in Combinatory Logic." In: *Proceedings of the National Academy of Sciences of the United States of America*. Vol. 20, No. 11, 584 – 590.

Detlefsen, Michael and Arana, Andrew (2011): "Purity of Methods." In: *Philosopher's Imprint*. Vol. 11, No. 2, 20.

Feferman, Solomon (1992): "What Rests on What? The Proof-Theoretic Analysis of Mathematics." In: *Akten des 15. Internationalen Wittgenstein-Symposiums: 16. Bis 23. August 1992, Kirchberg Am Wechsel*. Vol. 1, 147 – 171. Vienna: Hölder-Pichler-Tempsky. Reprinted in: Feferman 1998, chapter 10, 187 – 208.

Feferman, Solomon (1998): *In the Light of Logic*. New York and Oxford: Oxford University Press.

Gentzen, Gerhard (1934): "Untersuchungen über das logische Schließen I–II." In: *Mathematische Zeitschrift*. Vol. 39, 176 – 210, 405 – 431. English translation in Gentzen 1969, 68 – 131.

Gentzen, Gerhard (1969): *The Collected Papers of Gerhard Gentzen*. Szabo, Manfred E. (ed.). Amsterdam: North-Holland.

Girard, Jean-Yves, Taylor, Paul, and Lafont, Yves (1989): *Proofs and Types*. New York: Cambridge University Press.

Howard, William A. (1980): "The Formulae-As-Types Notion of Construction." In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Seldin, Jonathan P.; Hindley, J.R. (eds.). London and New York: Academic Press, 480 – 490.

Mancosu, Paolo (2017): "Mathematical Style." In: *The Stanford Encyclopedia of Philosophy*. Zalta, Edward N. (ed.). Metaphysics Research Lab, Stanford University, fall 2017 ed. https://plato.stanford.edu/archives/fall2017/entries/mathematical-style/.

Mancosu, Paolo (2018): "Explanation in Mathematics." In: *The Stanford Encyclopedia of Philosoph*. Zalta, Edward N. (ed.). Metaphysics Research Lab, Stanford University, summer 2018 ed. https://plato.stanford.edu/archives/sum2018/entries/mathematics-explanation/.

Montano, Ulianov (2014): *Explaining Beauty in Mathematics: An Aesthetic Theory of Mathematics*. Berlin: Springer.

Pierce, Benjamin C. (2002): *Types and Programming Languages*. Cambridge (MA): MIT Press.

Prawitz, Dag (1965): *Natural Deduction*. Stockholm Studies in Philosophy 3. Stockholm: Almqvist & Wiksell.

Putnam, Hilary (1980): "Models and Reality." In: *Journal of Symbolic Logic*. Vol. 45, No. 3, 464 – 482.

Reynolds, John C. (1974): "Towards a Theory of Type Structure." In: *Programming Symposium*. Robinet, B. (ed.), Lecture Notes in Computer Science 19. Berlin and Heidelberg: Springer, 408 – 425.

Sørensen, Morten Heine and Urzyczyn, Pawel (2006): *Lectures on the Curry-Howard Isomorphism*. In: *Studies in Logic and the Foundations of Mathematics*. Vol. 149. New York: Elsevier.

Wadler, Philip (2015): "Propositions as Types." In: *Communications of the ACM*. Vol. 58, No. 12, 75 – 84.