Open Peer Review on Qeios

Reification, Curry-Howard Correspondence, and Didactical Consequences

Reinhard Oldenburg[1]

1 Universität Augsburg

Funding: No specific funding was received for this work.

Potential competing interests: No potential competing interests to declare.

Abstract

The roles of languages, processes, and objects in mathematical thinking have led to many theories, yet no consistent big picture has evolved from this. This paper puts forth the hypothesis that the Curry-Howard correspondence from computer science and the theories it is built on provide a unification framework. This correspondence asserts that (formal) proofs and programs (in functional programming languages) do not only have some similarities, but can, at least if formalized in an appropriate way, be mapped to each other by an isomorphism such that proofs are programs and vice versa. Moreover, objects can be realized as function evaluation strategies, and this provides a model of the reification process. The paper explores all this and discusses the didactical relevance; especially, the reification theories are revisited. Computer-based realizations of concepts are used as a tool to show the consistency of ideas and the practicability of concepts.

Keywords: Process-object duality; reification; language; logic; algorithms; proof.

1. Introduction

Mathematics is a great and complex endeavor of human thought. The status and genesis of mathematical objects is still the subject of active philosophical thought, see e.g., Shapiro (2001) for a survey of the last decades regarding the question of whether mathematical objects are discovered or constructed. While for many years the view that they are constructed was dominant (e.g., Ernest, 1991), a new wave of mathematical realism has emerged recently (e.g., Balaguer, 2017). In mathematics education, the existence of objects is often assumed without explicitly analyzing where they come from. For example, it is commonplace to speak about multiple representations of functions, tacitly assuming that the represented objects exist (see Thompson & Sfard, 1994).

Where do these mathematical objects come from? If they are not given in a Platonic fashion, one needs to understand how they are constructed. Reification and procept theories (Sfard, 1991; Tall, 1991) are widely accepted approaches to this problem. A pivotal role in these theories is played by processes that turn into objects. The notion of process is, however, very wide and fuzzy. It includes algorithms that can be described in a very concise way, as well as vague and metaphorical ideas. Moreover, it remains open how the transformation of a process into an object is actually possible. Dubinsky's APOS theory (Arnon et al., 2014) is widely used as well. It emphasizes the view that procedures may act on other procedures, an idea that is essential for the following exposition as well, but the role of data types in APOS theory is not developed in such a depth as is needed for the correspondence described here.

After objects have been constructed, we use mathematical language to discuss their properties. In this view, the meaning of language is given by referring to objects (Kripke, 1980). For example, the common meaning of the symbol π is a certain numeric object referred to by the symbol, and + is a symbol that refers to an operation that combines two numbers (depending on the types of objects it operates on). This referential meaning of language has a long tradition and famous protagonists (Quine, 1974). However, at least since Wittgenstein's philosophical investigations (Wittgenstein, 1953) and the following language turn of philosophy (Rorty, 1967), the role of language has become central. Sfard's commognition theory (Sfard, 2008) emphasizes this by melting communication and thinking into one construct. But if language stands for itself and does not need reference to objects to fix its semantics, then meaning must come from out of the language itself, and Wittgenstein saw the language play as the source of meaning.

Having said all this, it should be clear that one can construct a dichotomy between giving either language or objects a pivotal role. Table 1 emphasizes this dichotomy, and it already contains some pairs that will come from the theory developed later in this paper.

Table 1. Language vs objects

Language is pivotal	**Objects are pivotal**
Objects are constructed from language	Language gets meaning by reference to objects
A single symbol is senseless, only the rules of its use in language play give it meaning	A single symbol can be assigned meaning
Quantification is to be understood by substitution of expressions	Quantification is to be understood by reference to objects
Mathematical questions are at least semi-decidable	There are undecidable questions in mathematics
Countable many objects are sufficient to do mathematics	There are more than countably many objects
All math can be learned from language play, therefore there is no upper limit for large memory models	Math needs object reference and thus cannot be reduced to language
Math can be represented in programming languages based on term rewriting	Math can be represented in programming languages that allow one to reference mathematical objects
Proofs are mathematical objects just like numbers	Proofs are separate from mathematical objects
Lambda calculus	Model theory
Brouwer's intuitionism	Hilbert's formalism

Table 1 emphasizes differences between the two points of view. However, we have already met reification theories, and in some sense, they bridge both sides by explaining how objects are constructed from processes (such as language play or others). This point of view will be strengthened by the theory developed in the present paper. The main goal of this paper is to show that the Curry-Howard-Correspondence provides a clear and sound theoretical framework that ties both sides closely together, thus resolving some apparent differences but also explaining and sharpening real differences.

The research questions that will be addressed are:

- Q1: How can the reification process be modelled in a theoretically and formally precise way?
- Q2: Is language play sufficient to provide all mathematical meaning?
- Q3: Is reference to objects needed to do mathematics?

The methods to answer these questions come from logic and computer science. Thus, the paper will appear very technical in some parts. The benefit is that this approach demonstrates that the theories presented are not just vague ideas but precise notions. It is not, however, meant that humans and computers think the same way. Computers don't think at all; they compute. But if an idea can be turned into an algorithm that can be executed on a computer to give exactly the desired results, then this is a proof that the notion is well-defined. As this approach is not widely used in mathematics education, some details and explanations will be given that go beyond a classical literature review. To compensate for this, well-known mathematical facts will be used without reference to the literature.

It seems that, up to now, there is not a single publication that discusses the Curry-Howard correspondence (short: CHC) from an educational perspective. Of course, there are hundreds of papers in theoretical computer science and in logic, including several books (Girard et al., 1989; Mimram, 2020; Sørensen & Urzyczyn, 2006). Moreover, there are some discussions about the philosophical relevance: Zach (2020) puts CHC in one league with Gödel's incompleteness theorem and the Löwenheim-Skolem theorem. For him, the main reasons to do so are that CHC allows to normalize (simplify) proofs because proofs become mathematical objects themselves, and that it allows strict type inference in programming languages. Quite differently, de Queiroz (2008) emphasizes the role of CHC in the attempt to give a meaning-by-use underpinning of language semantics.

2. Objects as basis of semantics

This short section recalls the standard approach (model theory) to the semantics in mathematical logic (Hilbert & Ackermann, 1967; Rautenberg, 2010; Tourlakis, 2003). Semantics is defined relative to a given domain of objects or universe of discourse. Variables are symbols that gain meaning by assignments that specify an object from the domain for each free variable in a formula. Similarly, function symbols and predicate symbols get interpreted by functions and relations over the domain. Universal quantification, e.g., is meant to mean that a formula is true under all assignments of the quantified variable. This model-theoretic point of view thus requires that there are objects. As Quine has put it, an object exists in this view if it can be the referent of a variable. That does not mean that objects need to exist in a platonistic way, it only means that an "ontological commitment" (Quine, 1948) is necessary, but this may be relative to some discourse or framework. The meaning of all parts of the language is then defined in terms of elements of this domain and relations between them.

Not only does the semantics of logical language need a precise description, but also the semantics of programming languages. Denotational semantics is the “computer science sister” of model theory, as it uses the same idea of symbols that refer to objects by means of an assignment (Scott & Strachey, 1971). Thus, the idea that the meaning of symbols is given by the objects they refer to can be observed in programming languages. For example, in the Scheme programming language (Shinn et al., 2013), a reference can be defined, e.g., by (define a 5), which lets a refer to 5 so that (+ a 1) evaluates to 6 (the notation with the operator written before the operands needs some getting used to but minimized ambiguities). Entering a into the language interpreter simply returns 5. And entering + returns a description of a function, i.e., + refers to an object in the computer memory, namely the procedure that carries out additions. Similarly, = refers to a procedure that determines if two objects are the same. In that way, all language elements (besides syntactical elements such as the parentheses) get their meaning by reference.

3. Language-game based semantics

Wittgenstein (1953) gave birth to the idea that meaning is defined alone by the actual use of words in the language game. This point of view is, of course, very appealing to educators because it naturally fits the way children learn their mother tongue by playing the language game with other speakers of the language.

If this point of view is correct, then large language models from artificial intelligence can indeed understand all aspects of language, as their only access to semantics is the observation of language use in a large corpus of text – but according to language game theory, this is sufficient. I have argued elsewhere against this position and proposed that observation of the language game may not be enough to cover all the logical content of language (Oldenburg, 2023). Even from a non-technical point of view, one can see that this idea is inherently vague, as no one can observe all the uses of words in a language. Nevertheless, this point of view is, of course, attractive from a constructivist perspective.

The semantics of programming languages can also be described in this manner, i.e., focusing on the behavior of the language interpreter. This is done in “operational semantics” (Kahn, 1987). This allows one to describe the semantics of languages without using the notion of reference. There are also languages that minimize the relevance of reference, e.g., Pure (Graef, 2017) or the Wolfram language of Mathematica (Wolfram Research, 2023). In Mathematica, e.g., the equal sign == does not refer to any procedure that checks equalness; it is a pure symbol. Its semantics is defined in a set of rewrite rules (Baader & Nipkow, 1998) that determine how the system transforms the input to the output.

4. Theories of reification

Several theories have investigated the relation of processes and objects in the learning process of mathematics. Sfard (1991; 2000) proposes that students evolve from understanding mathematical concepts initially as processes and then go on to gradually start seeing them as objects. Initially, in the learning process, students see concepts, e.g., like addition, as a procedure to be carried out, but as they get used to this, they can encapsulate the process into a new entity; it is frozen to become a new object, a sum. The same analysis can be carried out for many other mathematical objects that are mentally constructed: square roots, complex numbers, polynomials, etc.

Closely related is the theory of procepts developed by Tall (1991; 2014). A "procept" is a blend of a process and a concept, signifying that a mathematical symbol can represent both a procedure (like an operation or a calculation) and the abstract idea behind that procedure. A single mathematical symbol or expression can thus be understood in two ways: as a process to be carried out and as a concept to be contemplated. Tall assumes that students initially see a mathematical symbol only in terms of the process it represents. As their understanding grows, they start to see it as a concept as well.

Both theories deal with a developmental shift from operational (process-based) to more abstract (conceptual) understanding. One of the differences is that Sfard describes a developmental transition from processes to objects (reifying a process into a concept), while Tall focuses on the simultaneous duality of symbols as both processes and concepts.

5. From lambda calculus to reification

This rather technical section shows that reification is not just a metaphor, but that it can be realized in programming languages. This proves that it is a consistent theory. Some technical theory must be recalled first, however.

The two most fundamental models of computation are Turing machines and the lambda calculus. They have been shown to be equivalent, and it is the content of the widely accepted Church-Turing-Hypothesis (Church, 1936; Turing, 1937; for a modern description: Davis et al., 1994) that they describe everything that can be calculated at all. Lambda calculus is a formal (i.e., purely syntactic) calculus that has a very simple syntax. Its expressions are of the form *e f*, called an application, or $\lambda x{:}e$, where x is a variable and e, f are valid expressions. The idea is that a lambda expression describes an unnamed function, i.e., where in mathematics one writes $x \mapsto 2x$, one would write $\lambda x{:}2x$. The rules of the lambda calculus are (see Barendregt et al., 2013):

- $\lambda x.\, e \ \rightarrow \lambda y.\, e^{'}$, if e is free of y, and e‘ is obtained from e by substituting $x \rightarrow y$
  (α-conversion: variables bound by λ can be renamed)
- $(\lambda x.\, e)y \ \rightarrow e^{'}$, where e‘ is obtained from e by substituting $x \rightarrow y$
  (β-conversion: function application)
- $(\lambda x.\, e)x \ \rightarrow e$ (η-conversion: trivial application)

So, in principle, nothing else but lambda expressions exist. That is similar to building mathematics on top of set theory, where only sets exist, and numbers are defined from sets. In lambda calculus, numbers are defined to be functions: $``0" \equiv \lambda f.\, \lambda x.\, x$ , $``1" \equiv \lambda f.\, \lambda x.\, f\, x$, $``2" \equiv \lambda f.\, \lambda x.\, f\, f\, x$ etc. (church numbers). The addition 3+4 is then carried out by $\lambda f.\, \lambda x.\, ``3"\ f\, (``4"\ f\, x)$ which, when fully reduced by the three rules given above, yields the lambda expression for 7. Sets, tuples, and all other objects of math can be created from functions as well. To make more accessible how this works, we will realize this in a programming language.

Lambda calculus is realized in several programming languages, historically starting with Lisp (McCarthy, 1960). Progress in the correct understanding of the implementation of lambda calculus led to further development, and the language Scheme is a Lisp offspring that puts the theoretical power of the lambda calculus in a very plain and applicable language (Shinn et al., 2013). Scheme provides built-in implementations of numbers and other basic types of objects so that there is no need to realize them with lambda expressions, although this could be done. It also extends functions to allow them to have more than one argument (this is handy, but not necessary, neither for theory (due to a process called "currying") nor for practical programming, as the Haskell programming language shows). The application of a function f to arguments a1… an is written as (f a1 an). A lambda expression has the form (lambda (x1 … xn) e) where x1..xn are the variables of the function and e is the function body. The following example shows an application of a function that doubles its arguments. When entering this in Scheme (following the prompt >), the system responds with the result:

```
> ((lambda (x) (* x 2)) 7)
14
```

A further convenient function is the ability to name expressions. This is not necessary for the theory but very convenient to reduce complexity (and typing). (define a b) says that from now on a shall be a reference to b. Thus, one can write:

```
> (define mean (lambda (x y) (/ (+ x y) 2)))
> (define four 4)
> (mean 2 four)
3
```

Lambda expressions may be returned from functions, of course. The following takes a function f and builds a function that applies it to the absolute value of the input. Note that the minus sign in the name is part of the symbol (function name), not an operator.

```
> (define make-symmetric-function
    (lambda (f) (lambda (x) (if (< x 0) (f (- x)) (f x)))))
> ((make-symmetric-function sqrt) -16)
4
```

Next, we show how to create data objects, namely tuples, from functions. A tuple of two objects (a, b) is an object that allows one to extract its first and its second part. Moreover, there must be a way to construct the tuple objects from their constituents. Here is the realization in Scheme:

```
(define make-pair (lambda (a b)
        (lambda (n)
           (if (= n 1) a b))))
(define get-first (lambda (pair) (pair 1)))
(define get-second (lambda (pair) (pair 2)))
(define p (make-pair 17 23))
```

Then `(get-first p)` yields 17 and `(get-second p)` yields 23.

This is a first example of reification realized in technology: The process of selecting and giving a or giving b is encapsulated into a function, and this gives the tuple-object. When the object is used, this encapsulated object is turned into a process again. Next, the reification process is applied to construct the procept of square roots and real numbers.

The following calculates an approximation of the square root of x by bisection of the interval from low to up, with a precision of eps.

```
(define sqrt-eps
  (lambda (x low up eps)
    (let ((m (/ (+ up low) 2)))
      (if (< (- up low) eps)
        m
        (if (> (expt m 2) x)
          (sqrt-eps x low m eps)
          (sqrt-eps x m up eps)
          )))))
```

Then `(sqrt-eps 5 0 5 0.001)` yields `2+3867/16384`, i.e., it calculates $\sqrt{5}$ from the interval $[0, 5]$ up to precision 0.001. Note that the result is a rational number. By specifying a smaller eps, one can calculate better approximations, but, of course, none of these approximations is the real number $\sqrt{5}$. But encapsulating this calculation process, one gets an object that really is the real number in the sense of computational real numbers (Weihrauch, 2000).

```
(define Nsqrt
  (lambda (x)
    (lambda (eps)
      (sqrt-eps x 0 (+ 1 x) eps))))
```

```
(define sqrt2 (Nsqrt 2))
(define sqrt3 (Nsqrt 3))
> (sqrt2 0.0001)
1+27143/65536
```

The point here is that one really can say that sqrt3 is $\sqrt{3}$ as an object created from the process. To show that it is an object, one needs to exhibit how operations with these objects are performed. Here is summation:

```
(define Nadd
  (lambda (x y)
    (lambda (eps)(+ (x (/ eps 2)) (y (/ eps 2))))))
```

```
(define sqrt2-plus-sqrt3 (Nadd sqrt2 sqrt3))
```

Then `(sqrt2-plus-sqrt3 0.0001)` yields `3+19167/131072`

The result of `Nadd` is again an object of the same kind as `sqrt2` or `sqrt3`, which when called with a value for `eps` gives a rational approximation of the sum. Thus, this example shows a full reification process, including encapsulating a process and evoking the process again on demand. The only thing missing from this is the symbolic level, but this may be added if the functions are defined in a way that gives symbolic descriptions when the required precision is e.g., negative. This requires new definitions, now called `Ssqrt` and `Sadd`.

```
(define Ssqrt
    (lambda (x)
        (lambda (eps)
            (if (< eps 0)
                (list 'Ssqrt x)
                (sqrt-eps x 0 (+ 1 x) eps)))))
```

```
(define Sadd
    (lambda (x y)
        (lambda (eps)
            (if (< eps 0)
                (list 'Sadd (x -1) (y -1))
                (+ (x (/ eps 2)) (y (/ eps 2)))))))
(define ssum25 (Sadd (Ssqrt 2) (Ssqrt 5)))
> (list (ssum25 -1) '= (ssum25 0.001))
'((Sadd (Ssqrt 2) (Ssqrt 5)) = 3+5325/8192)
```

Summing up, the examples given in this section show that computational processes can indeed be frozen to form objects and that the processes can later on be evoked when needed. The crucial ingredient in this is functional abstractions, which leaves an expression unevaluated. This is like students that shall not carry out 2*13*5 immediately (starting from the left) but view it as an unevaluated expression that can be rearranged to get 2*5*13 which is easier to calculate – if a result is needed.

6. Curry-Howard correspondence

The lambda calculus is too powerful in a sense (as there are now restrictions on what applications can be written down) and allows formulating paradoxical things. Since all objects are functions, it is hard to tell functions that shall represent numbers apart from functions that represent sets etc. The solution is (not accidentally) the same that cures naïve set theory, which was plagued by Russel's antinomy, namely, to give types. An extensive description of typed lambda calculus is given by Barendregt et al. (2013).

The usual notation is $e{:}T$ to say that e is an expression of type T. Types may be thought of as somewhat similar to sets, and $e{:}T$ to be like $e \in T$. However, expressions are elements of a formal language, so there are only countably many expressions of a given type, while sets may be much bigger. The type $A \to B$ of a function means that the function maps elements of type A to elements of type B. Already this example shows that the set of types must be richer than just basic types like integers, floats, character strings, and arrays available in many programming languages. Especially, one needs the following type constructors: If A, B are types, then $A \times B$ is the type of tuples (a, b) where $a{:}A, b{:}B$ and $A + B$ is the type of elements that are either of type A or of type B.

Assume $f{:}A \to B$ and $x{:}A$, then $f(x){:}B$. Taking only the “type part” of this statement, one has: Assume$A \to B$ and A, then B. That is, types can be seen as propositions and that programs (lambda expressions) of that type exist can be seen as proofs of these propositions. This is the famous Curry-Howard Correspondence (CHC), see Curry & Feys (1958), Girard et al. (1989), Mimram (2020), and Sørensen & Urzyczyn (2006). The CHC is given in table 2.

Table 2. The Curry-Howard-Correspondence

Lambda calculus	Logic
Expression	Proof
Type	Proposition
Function type $A \to B$	Implication $A \to B$
Product type $A \times B$	Conjunction $A \wedge B$
Sum type $A + B$	Disjunction $A \vee B$
Empty Type, Unit type	False, true
Universal quantification	Dependent type
Program reduction by calculus rules	Proof simplification/normalization
Application of program to specific input	Apply/specialize proposition to an example
Continuation	Negation

The original CHC regarded only intuitionistic propositional logic. The extension to dependent types allowed the inclusion of the quantifiers of intuitionistic first-order predicate logic. Intuitionistic logic is constructive logic (Brouwer, 1981), which means that only those objects exist that can be constructed by an algorithm (note that this has nothing to do with epistemic constructivism as it extends objective truth even beyond humans, namely to machines, rather than restricting it). CHC brings out the constructive nature of intuitionistic logic clearly; one can say that it exhibits the computational content of a logical proposition. Consider $a{:}A,\ a^{'}{:}A$ and $b{:}B$. Hence, $a, a^{'}$ prove the same proposition. The existence of a can be seen as a witness of the truth of A and there may be more than one witness: All expressions of the same type prove the same proposition. Applying the rules of lambda calculus gives a simplified proof (normalization). A wrong proposition cannot have a proof; thus, falsity is the empty type. To prove $A \wedge B$, one needs a proof of \(A\\)and a proof of B, thus one has to have an element of type $A \times B$. A function $f{:}A \to B$ takes a proof of A as input and transforms it to a proof of$B$, thus it is an implication.

The main drawback of intuitionistic logic is that the law of excluded middle is not true: $A \vee \neg A$ may not be true because neither A nor $\neg A$ may be provable (recall Gödel's results), and hence the method of proof by contradiction is not considered valid.

The extension to non-intuitionistic logic was discovered by Parigot (1992). He discovered that the non-local control-flow operations, called continuations, that have been investigated in computer science for decades (and are part of the Scheme programming language), extend the correspondence to classical logic. The technical details are very difficult, but for our purposes, we only need an intuitive understanding. Consider that you are investigating a proposition A of which you don't know if it is true. When you go on hypothetically assuming it is true and drawing conclusions and finally discover that you get a contradiction, then you must go back to the point where you set the hypothesis that A is true and then go on knowing that $\neg A$ is true. Thus, one has to jump non-locally in the proof text – and this is what continuations do (they abstract over the future of a computation).

To sum up, proofs and programs (expressions in the typed lambda calculus) are in a 1:1 correspondence. Every proof can be read as a program, and every program can be seen as a proof. Thus, the intellectual activities to find proofs and to find programs should correspond, although the individual cognitive processes in setting up proofs and programs may differ. Yet it is a remarkable insight that computer science and mathematics developed independently with exactly the same structures. The CHC was not constructed but discovered!

7. A possible resolution of the language-objects duality

By now, a possible resolution of the language-object duality made in table 1 can be given by simply appending table 1 and table 2 to form a consistent theory: CHC bridges the gap by showing that differently looking things are indeed the same.

As explained in section 2, the semantics of mathematical logic, and thus of standard proof theory, rests on the existence of object domains, and meaning comes from reference to these objects. Consider especially universal quantification $\forall x : A(x)$. This is understood to mean that $A(x)$ holds, whatever element of the domain x refers to. There has been a second interpretation advocated by Ruth Marcus (for a discussion, see Hand, 2007), namely that the sentence $A(x)$ holds, whatever expression describing an element is inserted in the placeholder. At first sight, these two things seem to be different because there are domains with uncountably many objects, but there are only countably many expressions in any language. However, it was already settled by the Skolem-Löwenheim theorem within model theory that this difference may not be relevant, at least for first-order predicate logic. The CHC framework shows both interpretations to be equivalent (restricted to the logics covered) in the sense that they yield the same logical results. It does not mean, of course, that argumentations on both sides are equally easy to understand for humans and hence, one or the other may have specific advantages.

On the language side, equality of a, b means that a can be substituted for b (and vice versa) everywhere without changing the value of expressions (this is exactly how Leibniz defined identity). On the object side, $a = b$ means that both refer to the same object. Again, by benefit of CHC, both views are isomorphic, and hence one can choose the view by convenience depending on the situation. At this point, the importance of type theory comes in again. Consider the simple true formula $4(x+1) = 4x+4$. Hence, one can substitute in the equation $\sqrt{4x+4} = 12$ to get the equivalent $\sqrt{4(x+1)} = 12$, which can be further transformed to $\sqrt{x+1} = 6$. But what about the sentence "$4(x+1)$ is factored"? After substitution, one gets "$4x+4$ is factored". Type theory clarifies the situation: When expressions are considered to have the type of real numbers, then $4(x+1) : \mathrm{R} = 4x+4 : \mathrm{R}$, but over the type of expressions E, one has $4(x+1) : \mathrm{E} \neq 4x+4 : \mathrm{E}$.

Taken together, CHC bridges "meaning by use" and "meaning by reference". Regarding proof theory, similar ideas have been formulated by de Queiroz (2008), but the use in general mathematics and in mathematics education is new.

Both sides of the two tables fit together according to CHC. However, one may see reification as another band that ties both sides together: From the process of language use, a domain of objects is reified that enables a model-theoretic semantics that is equivalent to the process semantics that the reification process started from. And, conversely, one may start with already (mentally) constructed objects and try to understand the computational content. On the language side, substitution and its properties are essential for meaning and especially identity (recall Leibniz's definition of identity), and the same holds true on the object side because the primary feature needed to qualify as an object is that questions of identity are clarified ("No entity without identity," Quine, 1960).

8. Didactical relevance

An explicit teaching of the CHC is only possible at the university level, where it can be justified as an intellectual masterpiece. For mathematicians, CHC can have direct importance because proof assistant systems (such as Lean) and automatic provers are increasingly used in mathematics research, and most of them are based on CHC. For high schools and for universities, the theory may, however, have indirect consequences.

First of all, a scientific discipline should be clear about the nature of the things it investigates. Mathematics is often characterized by its deductive method of proofs, but CHC tells us that – with the same right – it could be characterized by its constructive method of algorithms. Proving and programming lead to the same results and are thus related, and they pose the same intellectual challenges. It is often said that an important goal of teaching mathematics at all levels is to show an authentic picture of what mathematics really *is*. As the CHC has the power to change the way professional mathematicians work, it obviously changes what mathematics is and what the role of logic and programming is within mathematics. Therefore, e.g., teacher students should have an opportunity to get an idea about this.

CHC implies that if somebody feels that proving is more difficult than programming (or vice versa), this must be due to different acquaintance with the notation (or with the theory of the domain, as proving in algebra is different from proving in calculus), but not with the intrinsic complexity of the problem.

The theory clearly exhibits the power of giving types to expressions. Type judgements are propositions and carry essential information. Declaring types is stating propositions. The theory therefore suggests that typing should be done with care and that students should be introduced to using typing as a tool to clarify thinking. It seems reasonable that even in elementary mathematics, clear definitions of types should be helpful. For example, Weigand & Oldenburg (2023) found that many students mix up operations (such as $\bar{z}$ for the complex conjugate of a number) and attributes (e.g., the attribute of being a vector $\vec{v}$). It may help them when types are declared by standard notation, not by rules for attributes that are domain specific. It seems also reasonable that the famous professor-student type task (Rosnick & Clement, 1980) may lead to fewer reversal errors when given as "At a university there are P:N professors and S:N students…." (assuming, of course, that this is not the first time students see type declarations). Moreover, in geometry, there is a distinction between a side of an object and the length of this side. This can concisely be expressed by type declarations – and the types determine what operations on the objects are possible. One might object that the same can be achieved by set-theoretic declarations, e.g. $a \in$ Q, However, this is cognitively more demanding than a:Q because it requires the set of all rational numbers to be constructed and because a refers to an element of this huge set (it is essential to understand this reference relation because not the variable a is an element in the set of rational numbers but the object that it refers to by means of some assignment). Let's shift to another domain to make the argument clearer: Assume your neighbor has an animal called Snoopy and you know what a dog is, i.e., you recognize a dog when you see one. Then you can certainly assert the type judgement Snoopy:Dog. Compare this to Snoopy $\in$ Dogs. This set-theoretic form requires that you form the set of all dogs, which is a cognitive monster: Does it include only living dogs or all dogs, even those that will be born in the future? Does it include toy dogs or paintings of dogs? In contrast, the type declarator ":" is a formalization of "is a" in everyday language, and its semantics is therefore much clearer.

The distinction between procedural and conceptual knowledge is used widely in mathematics education (Hiebert & Lefevre 1986). This distinction, while often useful, is however sometimes difficult to draw (Oldenburg, 2023b). Further questions arise when viewing proofs as the bearers of conceptual knowledge in mathematics and programs as bearers of conceptual knowledge. Then, of course, the distinction is no longer possible.

The relevance of CHC for a deep understanding of reification has already been pointed out. CHC strengthens the general approach, and the concrete realizations in Scheme above show that the "freezing" of processes is the central step. The programming technique applied in section 5 is also known as "lazy evaluation," which suggests that reification of algebraic constructs is connected to avoiding work. How this insight can be used in teaching is yet to be explored. However, in computer science education, there is a related and wide-spread teaching technique: E.g., if students are to learn loops, they are first instructed to draw a picture with repeated elements by textual programming. Hence, they have to repeat a lot of code and therefore appreciate when they are taught that writing loops greatly shortens the task. Or put the other way round: Getting tired of repetition can pave the way for abstraction. For mathematics education, the benefits of this principle are yet to be explored.

Coming back to the research questions posed in section 1, one can conclude that Q1 was answered in section 5. CHC also shows that the answer to Q2 is yes, but with the restriction that full classical logic requires non-local control flow, which extends the usual understanding of what language play is. In this sense, the answer to Q3 is that reference is not needed, but it makes a lot of things easier, especially when intuitionistic logic is not sufficient (e.g., as it avoids the complexity of non-local control on the language side).

These answers to Q2 and Q3 fit well with the developmental perspective on mathematical thinking: The language perspective is best for beginners, as there is no need to have constructed a domain of mathematical objects. By reification, these objects can be constructed from the processes in the language and can then form the basis of a referential understanding that makes advanced reasoning easier.

9. Conclusion

It has been argued that the Curry-Howard Correspondence, which states that programs and proofs are the same, is connected deeply with reification theory and has further didactical consequences. Most of them are not yet fully understood, but it seems that there is potential for better understanding of learning processes both in mathematics and in computer science. Maybe future research can use this as a basis to further cooperation between both subjects and research on their teaching and learning.

Literature

- Arnon, I., et al. (2014). *APOS Theory*. Springer. https://doi.org/10.1007/978-1-4614-7966-6
- Baader, F., & Nipkow, T. (1998). *Term Rewriting and All That*. Cambridge University Press. https://doi.org/10.1017/CBO9781139172752
- Balaguer, M. (2017). Mathematical Pluralism and Platonism. *Journal of Indian Council of Philosophical Research* 34, Nr. 2, S. 63–84. https://doi.org/10.1007/s40961-016-0084-4
- Barendregt, H., Dekkers, W., & Statman, R. (2013). *Lambda Calculus with Types*. Cambridge. https://doi.org/10.1017/CBO9781139032636
- Brouwer, L. E. J. (1981*). Brouwer's Cambridge Lectures on Intuitionism*. Cambridge University Press. (Original work published 1951)
- Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2), 345-363.
- Curry, H. B., & Feys, R. (1958). *Combinatory Logic* (Vol. 1). North-Holland Publishing Company.
- Davis, M., Sigal, R., & Weyuker, E. J. (1994). *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science* (2nd ed.). Academic Press.
- Ernest, P. (1991). *The philosophy of mathematics education.* Routledge.
- Girard, J.-Y., Lafont, Y., & Taylor, P. (1989). *Proofs and Types*. Cambridge University Press.
- Graef, A. (2017). *Pure*. https://agraef.github.io/pure-lang/
- Hand, M. (2007). Objectual and Substitutional Interpretations of the Quantifiers. In D. Jacquette (Ed.) *Handbook of the Philosophy of Science, Philosophy of Logic*, North-Holland. 10.1016/B978-044451541-4/50020-8

- Hiebert, J., & Lefevre, P. (1986). Conceptual and procedural knowledge in mathematics: An introductory analysis. In J. Hiebert (Ed.) *Conceptual and procedural knowledge: The case of mathematics* (pp. 1-27). Lawrence Erlbaum Associates.
- Hilbert, H., & Ackermann, W. (1967). *Grundzüge der theoretischen Logik*. 5th edition. Springer.
- Kahn, G. (1987). Natural semantics. In F. J. Brandenburg, G. Vidal-Naquet, M. Wirsing (Eds)*STACS 87*. Lecture Notes in Computer Science, vol 247. Springer, Berlin, Heidelberg. https://doi.org/10.1007/BFb0039592
- Kripke, S. (1980). *Naming and Necessity*. Harvard University Press.
- McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM, 3*(4), 184–195.
- Mimram, S. (2020). *Program=Proof*. Independently published. http://www.lix.polytechnique.fr/Labo/Samuel.Mimram/teaching/INF551/course.pdf
- Oldenburg, R. (2023). Limitations of and Lessons from the Learning of Large Language Models.*Qeios*. https://doi.org/10.32388/9FH6AD
- Oldenburg, R. (2023b). The procedural-conceptual dichotomy is not invariant under transposition to applied fields. in: Dreyfus et al. (Eds.) *The Learning and Teaching of Calculus Across Disciplines*– Proceedings of the Second Calculus Conference, June 5-9, 2023, Bergen, Norway
- Oldenburg, R., & Weigand, H.G. (2023). On the connection between basic mental models and the understanding of equations. In: M. Ayalon et al. (Eds) *Proceeding of the 46th Conference of the International Group for the Psychology of Mathematics Education*, July 16 – 21, 2023,University of Haifa.
- Parigot, M. (1992). Lambda-mu-calculus: An algorithmic interpretation of classical natural deduction. In*Proceedings of the International Conference on Logic Programming and Automated Reasoning*, LPAR '92, 190–201, Springer.
- Queiroz, R. J. G. B. de (2008). On Reduction Rules, Meaning-as-use, and Proof-theoretic Semantics.*Studia Logica*, 90: 211–247 DOI: 10.1007/s11225-008-9150-5
- Quine, W. V. O. (1948). On What There Is. *The Review of Metaphysics*, 2(5), 21–38.
- Quine, W. V. O. (1960). *Word and Object*. MIT Press.
- Quine, W. V. O. (1974). *The roots of reference*. Open Court Publishing Company.
- Rautenberg, W. (2010). *A Concise Introduction to Mathematical Logic*. Springer
- Rorty, R. (Ed.). (1967). *The Linguistic Turn: Essays in Philosophical Method.* University of Chicago Press.
- Rosnick, P., & Clement, J. (1980). Learning without understanding: The effect of tutoring strategies on algebra misconceptions. *The Journal of Mathematical Behavior*, 3, 3–27.
- Scott, D. S., & Strachey, C. (1971). Towards a mathematical semantics for computer languages. In Proceedings of the Symposium on Computers and Automata (Vol. 21, pp. 19-46). Microwave Research Institute Symposia Series. Polytechnic Press of the Polytechnic Institute of Brooklyn.
- Sfard, A. (1991) On the dual nature of mathematical conceptions: Reflections on processes and objects as different sides of the same coin. *Educ Stud Math* 22, 1–36 (1991). https://doi.org/10.1007/BF00302715.
- Sfard, A. (2000). Symbolizing mathematical reality into being: How mathematical discourse and mathematical objects create each other. In P. Cobb,K. E. Yackel, & K. McClain (Eds), *Symbolizing and communicating: perspectives on*

Mathematical Discourse, Tools, and Instructional Design (pp. 37-98). Mahwah, NJ: Erlbaum.

- Sfard, A. (2008). *Thinking as Communicating: Human Development, the Growth of Discourses, and Mathematizing.* Cambridge University Press.
- Shapiro, S. (2001). *Thinking About Mathematics: The Philosophy of Mathematics*. Cambridge.
- Shinn, A., Cowan, J., & Gleckler, A. (Eds) (2013). *Revised7 Report on the Algorithmic Language Scheme.* https://small.r7rs.org/attachment/r7rs.pdf
- Sørensen, M., & Urzyczyn, P. (2006). *Lectures on the Curry–Howard isomorphism*. Studies in Logic and the Foundations of Mathematics, vol. 149. Elsevier.
- Tall, D. (1991). The transition to advanced mathematical thinking: Functions, limits, infinity, and proof. In D. A. Grouws (Ed.), *Handbook of Research on Mathematics Teaching and Learning* (pp. 495-511). Macmillan Publishing Co.
- Tall, D. (2014). *How Humans Learn to Think Mathematically*. Cambridge.
- Thompson, P. W., & Sfard, A. (1994). Problems of reification: Representations and mathematical objects. In D. Kirshner (Ed.) *Proceedings of the Annual Meeting of the International Group for the Psychology of Mathematics Education — North America*, Plenary Sessions Vol. 1 (pp. 1-32). Baton Rouge, LA: Lousiana State University.
- Tourlakis, G. (2003). *Lectures in Logic and Set Theory Vol. 1: Mathematical Logic* Cambridge University Press.
- Turing, A. M. (1937). On computable numbers, with an application to the Entscheidungsproblem.*Proceedings of the London Mathematical Society*, Series 2, 42(1), 230-265.
- Weihrauch, K. (2000). *Computable Analysis: An Introduction*. Springer.
- Wittgenstein, L. (1953). *Philosophical investigations*. Basil Blackwell.
- Wolfram Research, Inc. (2023). *Mathematica* (Version 13.0) [Software]. Wolfram Research. https://www.wolfram.com/mathematica/
- Zach, R (2020). The Significance of the Curry-Howard Isomorphism. In G. M. Mras , P. Weingartner & B. Ritter (Eds.) *Philosophy of Logic and Mathematics*, de Gruyter. https://doi.org/10.1515/9783110657883-018