On the use of conceptual knowledge from computer science in mathematics education

Reinhard Oldenburg

Angaben zur Veröffentlichung / Publication details:

Oldenburg, Reinhard. 2025. "On the use of conceptual knowledge from computer science in mathematics education." In *Proceedings of the Fourteenth Congress of the European Society for Research in Mathematics Education (CERME14), February 3–7, 2025, Bozen-Bolzano, Italy*, edited by Marianna Bosch Casabò, Susana Carreira, Giorgio Bolondi, Michael Gaidoschick, and Camilla Spagnolo, 1846–53. Bozen-Bolzano: Free University of Bozen-Bolzano and ERME. https://hal.science/hal-05230093v1.

On the use of Conceptual Knowledge from Computer Science in Mathematics Education

Reinhard Oldenburg[1]

[1]Augsburg University, Mathematics Institute, Augsburg, Germany;
reinhard.oldenburg@math.uni-augsburg.de

In its need to describe formal system so precisely that they can be implemented on machines, computer science has developed many ideas that can also help to describe and understand mathematics. In fact, some of these ideas just refine and clarify older mathematical notions. Computer realizations of concepts prove (similar to the role of models in model theory) that concepts are free of self-contradictions. The clarification of objects and their relations has, moreover, the potential to improve students' learning. The paper will investigate formal calculi, type theory, lambda calculus and the Curry-Howard correspondence and explain their contribution to math education.

Keywords: Algebra education, computer science, lambda calculus, reification, type theory.

Introduction

The relationship between mathematics and computer science (CS) is multi-faceted, multidimensional and time-dependent (Modeste, 2015). As an offspring of math and engineering it inherits methods and knowledge from mathematics, and it gives tools to mathematics as is highlighted in the movement for computational thinking (Wing, 2006; Broley et al., 2023). A great deal of current research on this investigates algorithms and procedural tasks that can be carried out by machines, i.e. how actual computation (ranging in scope from using dynamic geometry to programs written by the learner) is useful in learning mathematics. The present paper discusses a relationship that is less prominent: Computer science can provide conceptual knowledge that explains and supports certain aspects of the learning of mathematics. Only very few publications have addressed this explicitly (e.g., Kortenkamp, 2006). There is a partial overlap with a prior more foundational paper (Oldenburg, 2024) but here the aim is to collect ideas that prepare the ground for actual teaching interventions.

Among the tasks of mathematics education is the clarification of the ontological status of mathematical objects and the understanding of how symbols acquire meaning (e.g., Queiroz, 2008). Computer science faces some similar problems, although a bit more difficult due to the fact that computer science needs to put its theories to test by implementing them on computers.

Methods

In its quest to define consistent and meaningful formal systems, CS has developed a bunch of concepts. If they can be successfully realized in soft- and hardware, then one may conclude that they are self-consistent (in analogy to the role of models in logic, see e.g., Mendelson, 2015). Extending this line of argument, one has the following methods on how to transfer conceptual knowledge from CS to math and math education my means of analogical reasoning (Bartha, 2010):

1) If a method of CS allows unambiguous definition of concepts, and if it allows interpretation by humans, then one expects that the method can clarify concepts for humans, too.

2) If a distinction between realizations of mathematical concepts is necessary in CS, one has reason to expect that this distinction is useful for understanding mathematics.

It is important to note that these are heuristic rules, they provide evidence, but no proof. Nevertheless, the research question of this paper is: Do these principles yield plausible results in a number of application cases? This question will be investigated in five topics. The choice of these topics was guided by the wish to present ideas that are hardly discussed in the didactical literature, and to span a range from more concrete to rather abstract and philosophical bridges between the domains.

Topic 1: Make properties of calculi explicit

Mathematics develops and justifies calculi (e.g., Baader & Nipkow, 1998). They are, in a sense, even more fundamental than algorithms: One can view functional algorithms as functional (lambda) calculus together with an evaluation strategy (Barendregt, 1984). Calculi are used in two different ways: They may either be used for semi-automatic reasoning in a given pre-existing theory, then one seeks for calculi that are correct and, if possible, complete (i.e. give the full theory), or they may be used in a creative sense to create a theory, e.g. when a group is defined from generators and relations.

In school, students learn a lot of calculi, e.g. the calculus of equivalent transformations of algebraic expressions, the calculus of transforming equations, forming the derivative or the anti-derivative. It might be beneficial for students to develop some conceptual knowledge about calculi and their properties. For example, equivalence of expressions and equations can be defined exactly the same way (as yielding the same (truth) value under any assignment of values to the variables), and rules of the equivalence calculus are correct but not complete. The derivative calculus is correct (by the usual proofs) and complete on the class of elementary school functions, while the school calculus of anti-derivates is obviously not complete. A complete calculus exists, but is beyond scope in school. An explicit treatment of calculi could also trigger epistemological and ontological considerations: What are the objects that are differentiated? Functions (e.g. as subsets of $\mathbb{R} \times \mathbb{R}$ with some properties) or expressions (i.e. strings of symbols formulated according to algebraic grammar)? Experience shows that the vast majority of teacher students in their third year at university have no idea about this and tend to get confused when asked about it. Most of them cannot even name and explain a difference between functions and expressions. But after explaining that the symbolic text of an expression can be "compiled" to a set-theoretic function, they usually appreciate the diagram in Figure 1.

Confluence is another property of calculi that might be useful to find orientation in the world of symbolic algebra. Moreover, specifying calculus rules often requires concepts that are rarely dealt explicit with in math lessons, e.g. the order of variables and monomials.

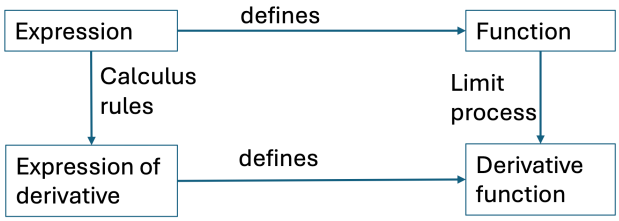

Figure 1: Relation of calculus and objects exemplified for the derivative

Topic 2: The distinction between program-time and run-time

Perhaps every algebra teacher in school knows situation in which students are confused when the values of variables may be changed. This problem even went into a cartoon where a pupil objects the solution $x = 5$ of an equation by arguing that the day before the teacher had said that $x = 3$. Understanding variable conventions correctly seems, however, to be a necessary condition for mastering other parts of elementary algebra (Oldenburg & Stacey, 2024). It is my suspect that the didactical concepts of variables as changing quantities or variables representing many numbers simultaneously are not helpful in this respect (Oldenburg, 2023), but that the distinction between program-time (or design-time) and run-time that students experience from day 1 when starting to program, may help. In fact, this distinction is even more wide-spread: When composing a PowerPoint presentation one can clearly distinguish design-time and presentation-time. This concept of different logical times explains the mentioned algebraic problem easily: One may distinguish between calculus-time where on sets up expressions, transforms them by calculus rules, and chooses (and changes) assignments for variables on one hand, and, on the other hand, calculation-time when an assignment of numbers for every variable is fixed and expressions are evaluated with respect to this assignment. At calculation-time each variable stands for one and only one number, but at calculus-time no value needs to be fixed at all. Changing the value of variables is the same as running a program several times with different inputs.

Topic 3: Type theory and algebra: Enabling the substitution principle

In programming, the data type of objects determine what operations are possible with the objects. Type declarations can ease the understanding of code. In mathematics, type theory has evolved since its incarnation in Russell's approach to avoid paradoxes to a complex system of theories (The Univalent Foundations Program, 2013) that is believed by some mathematicians to be a replacement for set theory and first order logic as foundation of mathematics (Altenkirch, 2023). For the working mathematician and for educational purposes, the less ambiguous simple type theory (Farmer, 2023) provides many benefits, but a discussion of this is far beyond the scope of this paper. Here, I'd like to draw attention to the fact that type declarations can clarify the semantics and support learning. It has been shown (e.g., Hanenberg et al., 2013) that type annotations in programming languages can improve program comprehension and detection of errors. It is plausible that analogous results hold at least in higher mathematics. Consider e.g. the action functional in Lagrangian mechanics which is often abbreviated as $S(q, \dot{q})$ although a correct definition would be $S(q) \coloneqq \int_a^b L(q(t), \dot{q}(t), t)dt$, i.e. the argument of S is a function (i.e. of type $\mathbb{R} \to \mathbb{R}$, rather than type $\mathbb{R}$), so that the type of S is $(\mathbb{R} \to \mathbb{R}) \to \mathbb{R}$. Giving correct type information might help in understanding the theory, although it seems that this has not been investigated. Moreover, one may conjecture that type declarations may support learning of more elementary mathematics as well.

From an ontological point of view, types require less commitment than set theoretic notions: Understanding the full meaning of $x \in \mathbb{R}$ requires to mentally construct the set of all real numbers and saying that x is one of its uncountably many elements. In contrast, the type declaration $x\colon \mathbb{R}$ just says that x is a real number, and to understand this one just needs an idea what a single real number

is, not what the set of all real numbers is. To emphasize this point even more, compare the statements "Snoopy is a dog" and "Snoopy is in the set of all dogs". The latter requires thinking about a complex construct, the set of all dogs (only living or also past and future dogs? Are wolves included?), while the former one only needs the ability to identify Snoopy as a dog. Of course, one has to admit that often the explicit use of sets and classes can be eliminated from propositions ("virtual classes" in the parlance of Quine (1969)), but this eases difficulties that may not be present if set theory would be replaced by type theory from the start.

There is another issue that may pose difficulties for learners and that may be clarified by giving explicit declarations. Leibniz defined identity in terms of substitution: Two expressions have identical meaning if one can be substituted for the other everywhere without changing the truth value of statements (salve veritate) (see, e.g. Mates, 1986). Quine (1960) has noted that there are opaque contexts in language that may violate this principle, e.g. although "Paris" and "the capital of France" refer to the same thing and are thus identical, it may be true that "Peter believes that London is the capital of France" but the substituted version "Peter believes that London is Paris" may be false. In fact, similar contexts exist in algebra: Although $2(x+1) = 2x+2$ holds, one may not substitute one for the other in the proposition "$2(x+1)$ is factorized" because this yields the wrong statement "$2x+2$ is factorized". The problem can be resolved by recognizing, that the type of $2x+2$ is not declared. If one declares $2x+2 : \mathbb{R}$ and $2(x+1) : \mathbb{R}$, then they are equal and can be substituted for each other. However, "factorized" is not a predicate for objects of type $\mathbb{R}$. On the other hand, if the expressions are declared to be of type 'expression', then they differ and one has the property of being factorized, the other not. Hence, type-respecting substitutions heal this problem.

Topic 4: Reification can be formally understood as functional abstraction

The mental construction of mental objects is considered in a large variety of approaches, e.g. reification theory (Sfard, 1991), APOS theory (Dubinsky & Mcdonald, 2001), or procept theory (Tall, 2014). These approaches share a lot but differ in detail. All of them are, however, only described informally and their consistency is implicitly deduced from their plausibility. However, since Church introduced the realizations of natural numbers in pure lambda calculus (Church, 1936), it has been known that objects can be constructed from functions, i.e. from processes. Other examples are the constructions of tuples from procedures (Abelson et al., 1996). Although these examples are loaded with the burden of some technical details that are absent in a natural language explanation, they have the benefit that they can be tested by realizations on computer and that they make the object creation step explicit: Functional abstraction prevents an expression to be evaluated and captures it in a "frozen" form. The same mechanism also enables to deal with infinite sequences, e.g. the sequence of natural numbers or of prime numbers. At first sight, there is little didactical benefit from this insight because it is well known that students need to overcome the urge to carry out calculations and to accept "open expression" (lack of closure obstacle, Thompson & Tall, 1991) – i.e. they need to learn to freeze a calculation to an object. What is, however, interesting is the conclusion that delaying the evaluation and capturing the calculation in unevaluated form is at least at this level not only necessary but sufficient to build new objects. This supports e.g. the sensibility of exercises that present a numerical expression and ask students to explain the structure without calculating it.

Topic 5: The Curry-Howard correspondence bridges "meaning by reference" and "meaning by use"

To understand where the meaning of words and sentences comes from is a major goal of philosophy and several theories have been developed. For the understanding of mathematics two theories (or families of theories) may be considered most influential:

Referential theories: Words acquire meaning because they refer to something. The standard interpretation of predicate logic following the line of Hilbert (Hilbert & Ackermann, 1967), Tarski (1941) and Quine (1981) falls in this class, because interpretations of symbols must be fixed, that assign e.g. a function to each function symbol, and this fixes the symbol's meaning.

Language play theories: The idea that the meaning of words is given by the way they are used in the language game is due to Wittgenstein (1958), but has spread further (e.g., Kripke, 1982).

It is clear that referential theories are more naturally combined with realistic or relativistic convictions on the status of mathematical objects, while language play theory is more in line with constructivist theories. Of course, this dichotomy over-simplifies the philosophy of meaning, but it is quite useful in its application to the learning in general (and of math in particular) and the semantics of programming languages: Assume I shall explain what the German word "Baum" means. One way is to ostentatiously exhibit objects the word "Baum" refers to, i.e., I shall point to trees, or I give examples of sentences where the word is used. Another example bridges math and computer science: What does + mean? The referential explanation is that it refers to the operation of adding numbers. This can be made concrete e.g. in the programming language Scheme, were + refers to a function object that can be applied to numbers, but it can also be stored in data structures or passed to functions. The meaning of + can also be fixed by language play. Then, + does not refer to anything, it is just a symbol that can be used according to some rules, e.g. the rule that $x + 0$ can always be changed to x or that $x + y$ can always be changed to $y + x$. In computer science, this is realized in term rewriting systems (Baader & Nipkow, 1998) such as Mathematica (Wolfram Research, 2024). Lambda calculus can be seen as a term rewriting programming language.

Both theories are in a sense in opposition: For referential theories, objects are fundamental, and language plays the secondary role of describing objects and their relations. In contrast, in language play theories, of course, language is fundamental, and objects emerge from talking as is they were there. This carries over directly to logic: The meaning of $\forall x: p(x)$ in referential theory means that $p(x)$ is true, whatever object x refers to. In language play theory it means that $p(x)$ is true, whatever expression is inserted for in place of x in the formula $p(x)$ (in philosophy, the terms used are usually objectual and substitutional quantification, see Hand, 2007). These two interpretations of quantification may be different: There are only countable many expressions, but maybe more than countably many objects. The mainstream in mathematical logic adopts the referential interpretation and justifies the rules of proof systems from this (Mendelson, 2015). The "world" of referential theories thus includes objects, model theory, denotational semantics-based proof theory as well as referential programming languages like Scheme or Julia. Besides obvious strengths, a clear drawback, especially as seen from an educational perspective is that these theories assume (at least as hypothesis in the sense of model theory or relativistic ontology) that the objects that symbols refer to "exists" in

some form. Language play theory in contrast does need such assumptions. One may first fix conventions of the language game and then objects may emerge from this when one observes that it is a good interpretation of words to assume that they refer to some object. Consider the intersection of sets: from a referential point of view, one may think of A, B as existing sets of existing objects and form $\{x: x \in A \wedge x \in B\}$. From a language play theory, one may know how to speak about having property A or property B and establish rules how to speak about things that have both properties. From this use of language, it may emerge that it is sensible to assume that there are sets of these things. So, language play theory includes processes, term rewriting, much of type theory, and programming languages such as lambda calculus, Pure or Wolfram Mathematica. I should be said that this simple dichotomic picture is adequate only when restricting to the main-stream approaches. However, there is the concept of proof-theoretic semantics which puts semantics in the language world, and models for type theories which puts type theories into the referential world.

With reification theory, we have already met a bridge between these worlds, showing that processes can be turned into objects. This bridge can be extended to span all of the two worlds: The famous Curry-Howard correspondence (Curry & Feys, 1958; Girard et al. 1989; Mimram, 2020) gives an isomorphism between formalized proofs and programs in typed lambda calculus, i.e. programs are proofs, and their types are the propositions proved by them. This surprising result seems not to be recognized much in educational sciences (Oldenburg, 2024). The whole theory is quite involved, but a key idea is that logical modus ponens is the same deriving $f(a): B$ from $a: A, f: A \to B$.

There are several insights gained from this: First of all, the two seemingly distinct worlds described above are tightly linked and therefore consistent. The important role of types that was already pointed out above is further emphasized. Computational thinking (e.g., Wing, 2006) is brought even closer to mathematical thinking because, in the language of ATD (Chevallard, 2019), the connections between the logos of math and computer science extend to the whole praxeologies of both domains. The traditional emphasis of math education on functional thinking is justified. Moreover, it seems plausible that the learning process may start out with language play and referential interpretations may represents a more developed form that should be supported by teaching.

Conclusion

This paper began by questioning whether the principles of transferring conceptual knowledge from computer science to mathematics education can provide valuable insights. The conclusion, as evidenced by the examples provided, is affirmative. The exact way this transfer is accomplished is somewhat different between the five topics, but this is not uncommon with heuristic methods. A key didactical implication is the need to place greater emphasis on precisely defining types, as clarity about types parallels clarity about propositions. It should be noted that the list of five topics discussed is not exhaustive. Additional topics worthy of consideration in this context include generative grammars, the parsing of expressions, and the concept of effective computability—a property that, for instance, the supremum of a bounded subset of $\mathbb{R}$ generally lacks. Clearly, there is an abundance of ideas in this area that remain to be explored.

References

Abelson, H., Sussman, G. J., & Sussman, J. (1996). *Structure and interpretation of computer programs* (2nd ed.). MIT Press.

Altenkirch, T. (2023). Should Type Theory Replace Set Theory as the Foundation of Mathematics? *Global Philosophy* 33, 21. https://doi.org/10.1007/s10516-023-09676-0

Baader, F., & Nipkow, T. (1998). *Term Rewriting and All That*. Cambridge University Press. https://doi.org/10.1017/CBO9781139172752

Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics* (vol. 103). North-Holland.

Bartha, P. F. A. (2010). *By Parallel Reasoning: The Construction and Evaluation of Analogical Arguments*. Oxford University Press.

Broley, L., Buteau, C., Modeste, S., Rafalska, M., & Stephens, M. (2023). Computational Thinking and Mathematics. In B. Pepin, G. Gueudet, J. Choppin (Eds.), *Handbook of Digital Resources in Mathematics Education*, Springer, pp.1-38, 2023, Springer, 978-3-030-95060-6. ff10.1007/978-3-030-95060-6_12-1ff. ffhal04536404f

Chevallard, Y. (2019). Introducing the Anthropological Theory of the Didactic: An attempt at a principled approach. *Hiroshima Journal of Mathematics Education*, 12, 71–114.

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.

Dubinsky, E., & Mcdonald, M. A. (2001). APOS: A Constructivist Theory of Learning in Undergraduate Mathematics Education. In D. Holton et al. (Eds.) *The Teaching and Learning of Mathematics at University Level* (pp. 275-282). Springer.

Farmer, W. M. (2023). *Simple Type Theory*. Birkhäuser.

Girard, J.-Y., Lafont, Y., & Taylor, P. (1989). *Proofs and Types*. Cambridge University Press.

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. https://doi.org/10.1016/B978-044451541-4/50020-8

Hanenberg, S., Kleinschmager, S., Robbes, R., & others. (2014). An empirical study on the impact of static typing on software maintainability. *Empirical Software Engineering*, 19(6), 1335–1382. https://doi.org/10.1007/s10664-013-9289-1

Hilbert, H., & Ackermann, W. (1967). *Grundzüge der theoretischen Logik* [*Principles of Theoretical Logic*] (5th ed.). Springer.

Kortenkamp, U. (2006). Terme erklimmen. Klammergebirge als Strukturierungshilfe. *Mathematik lehren* 136.

Kripke, S. A. (1982). *Wittgenstein on Rules and Private Language*. Harvard University Press.

Mates, B. (1986). *The Philosophy of Leibniz: Metaphysics and Language*. Oxford University Press.

Mendelson, E. (2015). *Introduction to mathematical logic* (6th ed.). CRC Press.

Mimram, S. (2020). *Program=Proof.* Independently published. http://www.lix.polytechnique.fr/Labo/Samuel.Mimram/teaching/INF551/course.pdf

Modeste, S. (2015). Impact of Informatics on Mathematics and Its Teaching. 3rd International Conference on History and Philosophy of Computing (HaPoC), Oct 2015, Pisa, Italy. pp.243–255, ff10.1007/978- 3-319-47286-7_17ff. ffhal-01615294f

Oldenburg, R. (2023). The didactical transformation of the concept of variables. In P. Drijvers, H. Palmér, C. Csapodi, K. Gosztonyi, & E. Kónya (Eds.), *Proceedings of the Thirteenth Congress of the European Society for Research in Mathematics Education (CERME13)* (pp. 624–631). Alfréd Rényi Institute of Mathematics and ERME. https://hal.science/hal-04418259v1

Oldenburg, R. (2024). Reification, Curry-Howard Correspondence, and Didactical Consequences. *Qeios*. https://doi.org:10.32388/S1074V

Oldenburg, R., & Stacey, K. (2024). Statistical Implicative Analysis of Students' Algebra Performance. *Qeios*. https://doi:10.32388/FL8ENG

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. (1960). *Word and object*. MIT Press.

Quine, W. v. O. (1969). Set Theory and Its Logic (Revised ed.). Harvard University Press.

Quine, W. v. O. (1981). *Mathematical Logic* (Revised ed.). Harvard University Press.

Sfard, A. (1991) On the dual nature of mathematical conceptions: Reflections on processes and objects as different sides of the same coin. *Educational Studies in Mathematics* 22, 1–36 (1991). https://doi.org/10.1007/BF00302715

Tall, D. (2014*). How Humans Learn to Think Mathematically*. Cambridge University Press.

Tarski, A. (1941). *Introduction to logic and to the methodology of deductive sciences* (O. Helmer, Trans.). Oxford University Press.

The Univalent Foundations Program (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study. https://homotopytypetheory.org/book

Thomas, M., & Tall, D. (1991). Encouraging versatile thinking in algebra using the computer. *Educational Studies in Mathematics*, 22(2), 125–147.

Wing, J. (2006). Computational thinking. *Communications of the ACM*, 49(3), 33–35.

Wittgenstein, L. (1958). *Philosophical Investigations* (G. E. M. Anscombe, Trans.). Blackwell.

Wolfram Research, Inc. (2024). *Mathematica* (Version 14.1) [Computer software]. Wolfram Research, Inc. https://www.wolfram.com/mathematica/