The strange fate of abstract thinking

Alexandre V Borovik

University of Manchester

The use of Information and Communication Technology (ICT) in mathematics teaching exploits the possibility of easy generation of concrete examples and encourages in students a bottom-up, inductive style of thinking. This increases the gap between mathematics as it is taught and the research level mathematics dominated by hierarchically structured top-down abstract thinking. Interactive representation of mathematics via user-friendly interfaces is made possible by advances in computer science. Paradoxically, computer science itself requires a level of abstract thinking that far exceeds the one normally acquired by mathematics graduates. The paper discusses a possible remedy: the use of high level functional programming languages such as Haskell as a tool for handling abstract mathematical structures.

Introduction and disclaimers

This short paper belongs to a series of publications (see, for example, Borovik 2011) informed by my work in the Education Committee of the London Mathematical Society [LMS] and the Research Committee of the Association for Learning Technology [ALT]. I support the LMS's Position Statement on ICT in Higher Education (London Mathematical Society, 2011). But, to avoid any ambiguity, I have to emphasise that the present paper is written by me in my private capacity and does not necessarily reflect views of the LMS, or of the ALT, or any other institution or organisation. The limited volume of the paper forces me to focus it on just one facet of the immensely complex problem of mathematical education in the modern world: development of abstract thinking. I restrict my recommendations to university level teaching of mathematics.

Abstract thinking in modern society

Although my paper is about university level teaching of mathematics, let us start by looking at school level mathematics and Information and Communication Technology (ICT).

INTELLECT, a trade association for information technology industry in the UK (its members include Accenture and IBM, among others), recently proposed in their submission to the National Curriculum Review (Intellect 2011) that

> *ICT as a statutory subject in its current form effectively discourages students from progressing to the more advanced computing courses [...] In place of ICT classes for all students, INTELLECT believes potentially gifted tech students should be encouraged into the core science, engineering and maths subjects.* [...]
>
> *INTELLECT has recommended to the DfE that computing should be a discrete subject available to pupils from Key Stage 3 onwards with options to follow a progression path where they learn increasingly more advanced skills.* (Mitchel 2011)

This statement is interesting because it sheds light on the intrinsic flaw at the core of the current concept of the use of ICT in teaching: it is the word “use”. The “use” produces “users”, not “producers” or “creators”. The IT industry discovered that teaching children to “use” ICT does not encourage them to become creators of ICT. The IT industry wants schools to teach actual computing, that is, development of algorithms and their subsequent implementation in computer code. (There are some interesting overlaps of INTELLECT statement and a recent policy statement of the ALT (2011).)

The purpose of my paper is to highlight an important aspect of information and computer technology: learning modern computing (that is, computer programming) requires a very high level of abstract thinking. This position is actively promoted by Kramer (2007). An anonymous contributor to discussion of Kramer's paper in my blog *Mathematics under the Microscope* (Borovik 2007), a professional computer scientist, left the following comment:

> *“I would caution everyone not to confuse “mathematical thinking" with “the thinking done by computer scientists and programmers".*
>
> *Unfortunately, most people who are not computer scientists believe these two modes of thinking to be the same. The purposes, nature, frequency and levels of abstraction commonly used in programming are very different from those in mathematics.”*

I fully accept this reminder: indeed, I agree that the demands to abstract thinking even at relatively elementary stages of study of computer science considerably exceed similar demands of traditional undergraduate mathematics. This statement may appear to be extreme, but let us not to jump to conclusions and look first at a very simple example.

A simple but illuminating example: polymorphism in Matlab

I suggest to have a look at Matlab, an industry standard software package for mathematical (mostly numerical) computations which is widely used in university level mathematics teaching. I apologise to my computer scientist colleagues who on a number of occasions explained to me that Matlab is nothing more but a glorified calculator. Of course, at the level of a novice user (like our students), it is a big powerful calculator; but I will try to show you that the innards of Matlab are quite sophisticated. I choose Matlab because, for a lay person, it provides an easy, even if limited, insight into what is going on in computer realizations of basic mathematical structures, like natural numbers - what could be simpler? I emphasise: there is nothing special in my examples, they are routine; similar examples can be found in many and can be found in many programming languages and software packages.

The following fragment of text is a screen dump of me playing with natural numbers in Matlab.

```
>> t= 2

t =     2

>> 1/t

ans =    0.5000
```

What you see here is a basic calculation which uses floating point arithmetic for computations with rounding; lines starting with the prompt '>>' are my input; unmarked lines are Matlab 's response.

Next, let us make the same calculation with a different kind of integers:

```
>> s=sym('2')

s =    2

>> 1/s

ans =  1/2
```

Here we use "symbolic integers", designed for use as coefficients in symbolic expressions. You can see that in the first example 1/2 was rounded as 0.5000, in the second case 1/2 is written as it is, as a fraction.

Since Matlab keeps in its memory the values of the variables s and t, we may force it to combine the two kinds of integers in a single calculation:

```
>>  1/(s+t)

ans =  1/4
```

We observe that the sum $s + t$ of a floating point number t and a symbolic integer s is treated by Matlab as a symbolic integer. Examples involving analytic functions are even more

striking:

```
>> sqrt(t)
ans =      1.4142
>> sqrt(s)
ans =      2^(1/2)
>> sqrt(t)*sqrt(s)
ans =      2
```

We see that Matlab can handle two absolutely different representations of integers, remembering, however, the intimate relation between them.

Matlab is written in C++. When represented in C++, even the simplest mathematical objects and structures may appear in the form of (a potentially infinite variety of) classes linked by mechanisms of *inheritance* and *polymorphism*. This is a manifestation of one of the paradigms of the computer science: if mathematicians instinctively seek to build their discipline around a small number of "canonical" structures, computer scientists frequently prefer to work with a host of similarly looking structures, each one adapted for a specific purpose.

When I talk about abstract thinking, I really mean serious mathematical abstraction which remains beyond the grasp of many (perhaps most) mathematics undergraduates. Indeed, managing mathematical concepts behind, say, the notorious "inheritance and polymorphism" of C++ and C# (and many other programming languages) involves de-facto use of concepts of category theory (even if the words "category theory" are not mentioned). The mathematician reader would perhaps agree that it is not that difficult to learn how Matlab treats different versions of integers – but to write a slim and efficient code that allows their blending into one system requires, indeed, some ability to simultaneously treat mathematical objects as equal and different, as identical and distinct – which means the ability to look at them from a higher and very abstract point of view.

Without the ability for higher level abstract thinking texts like the following routine fragment from an on-line tutorial on programming in C# will remain impenetrable for the reader:

> *"When you derive a class from a base class, the derived class will inherit all members of the base class except constructors, though whether the derived class would be able to access those members would depend upon the accessibility of those members in the base class. C# gives us polymorphism through inheritance. Inheritance-based polymorphism allows us to define methods in a base class and override them with derived class implementations. Thus if you have a base class object that might be holding one of several derived class objects polymorphism when properly used allows you to call a method that will work differently according to the type of derived class the object belongs to."* (Sivakumar 2001)

This quote has been intentionally taken from the very first Google hit in my search for "inheritance and polymorphism in C" – to emphasise its generic nature.

Computer Aided Assessment

``Teaching to the test'' is a dangerous but underestimated trend that slowly erodes the fundamentals of mathematical education. For that reason, Computer Aided Assessment (CAA) deserves special attention in our discussion of the uses of ICT in mathematics teaching and learning.

Many currently available CAA systems for mathematics teaching suffer from serious technical deficiencies -- such as the predominance of multiple choice tests and lack of functionality for easy and unconstrained entry of mathematical formulae; see Sangwin (2006) and Sangwin and Ramsden (2007) where these issues are analysed and some remedies are offered.

We can safely assume that the teething problems of the new technology will be cured soon; however we still need to understand the unavoidable limitations of CAA: they are better suited for testing routine procedural skills rather than creative thinking and understanding of highly abstract concepts.

The London Mathematical Society (2011) warns that the use of CAA takes place in the environment shaped by already established cultural expectations and administrative demands:

> *"We should expect a pressure to switch to CAA not only in formative assessment and coursework tests, but also in course examinations. Indeed, experience shows that a formative CAA translates better to good exam results if the exams are set in the CAA format already familiar to students. There is a danger that if students see that the use of CAA for formative assessment helps to achieve desired test and exam results they are likely to make the CAA their learning tool of choice and ignore other forms of learning."*
>
> *"The main danger associated with the CAAs is that their easy availability will increase the already existing pressure to "teach to the test" – and, which could happen to be a much worse outcome – "to teach to the* **computerised** *test".*
>
> *"Paradoxically, the more successful a CAA the more harm it may bring to mathematics education in the long run."*

There are further paradoxes. An efficient formative assessment may improve students' performance but depress students' satisfaction with the course. I refer the reader to the cautionary tale told by Fried (2011): his IT solution used in undergraduate teaching of vector calculus

> "Led students to see they could work harder. Many of my students (certainly not all) interpreted that as a negative."

This opens quite a can of worms: evaluation of courses and the dangerous question: what aspects of learning and teaching are actually evaluated? See Zucker (2010) for a provocative discussion. This topic, however, lies outside of scope of this paper.

But, for the purpose of the present paper, I emphasise the principal deficiency of CAAs: In the present form, they do not allow to test, and hence teach, abstract thinking. The possibility of an easy on-the-fly generation of randomised concrete examples creates in teachers a temptation to let students to teach themselves and encourages in students a bottom-up, inductive style of thinking. The emphasis on learning-by-example, on learning-by-doing increases the gap between these first stages of learning and its more advanced forms, such as "learning by adopting an abstract conceptual framework and specialising it to concrete situations".

This increases the gap between mathematics as it is taught and the "research" level mathematics dominated by hierarchically structured top-down abstract thinking. And the worst paradox of all is that interactive representation of mathematics via user-friendly interfaces is made possible by advances in computer science – but computer science itself requires a level of abstract thinking that far exceeds the one that can be provided by ICT tools for teaching and learning as they are evolving now.

Haskell, a potential saviour of abstract mathematics?

To avoid ending my paper on a pessimistic note, I wish to briefly and very tentatively discuss a possible remedy: the use of high level functional programming languages such as Haskell (perhaps after proper adaptation) in the university level mathematics teaching and learning as a tool for handling abstract mathematical structures. This is something that has already been tried in a few universities around the world (Doets & van Eijck 2004). I am prepared to accept that the idea could happen to be more attractive to computer scientists who teach mathematics to their students than to mathematicians involved in the mainstream mathematics teaching; but I think that the demarcation line between mathematics and computer science is artificial and mathematics, for the sake of its health and stability as a discipline, should pay more attention to computer science.

Among the huge variety of programming languages Haskell is interesting because of its cult status and proselytising zeal of its fans. An example like the one below is the best explanation why this has happened. These four lines are an implementation of the Sieve of Eratosthenes, a classical algorithm for listing prime numbers (Hutton 2007):

```
primes          :: [Int]
primes          = sieve [2..]
sieve           :: [Int] => [Int]
sieve (p : xs)  = p : sieve [x | x <= xs, x `mod` p ≠ 0]
```

What happening here is that function *sieve* is applied to the infinite list

[2..] = [2, 3, 4, 5, 6, 7,...].

The function *sieve* takes as argument a list of integers and returns a list of integers; it accepts the first number *p* being prime and then calls itself recursively with a new list obtained by filtering out all multiples of *p* from the list. In Haskell, expressions are evaluated using call-by-name evaluation, not by call-by-value evaluation. This is like manipulating with symbols or formulae in a mathematical calculation and substituting the numerical values in an expression at the very last step. This method of evaluation results in the function *primes* returning a (potentially infinite, hardware limitations and the expected life span of the Solar System permitting) list of infinite numbers:

```
> primes
[2, 3, 5, 7, 11, 13, 17, ...
```

For a mathematician, the above four lines of Haskell code are a revelation.

First of all, the programme (and the computer running this programme) appear to successfully manipulate with infinite sets. Secondly, the modest four lines of code simply brush aside the centuries long dispute about the actual and potential infinity and blend the apparently irreconcilable concepts together in the spirit of “internal set theory” (Nelson 1997, Robert 2003): an infinite set is treated as a real object because it is given a name which is then used as a name of an actual object. On the other hand, we have access only to those elements in the set that we have already explicitly constructed. The set is like a cookies jar which contains only those cookies that we have put in it – but we can take the jar from a shelf and place it on a table regardless of how many cookies it contains.

For a teacher of mathematics, Haskell is a revelation because of its simplicity and remarkable power of abstraction amply demonstrated by the four lines of code for the Eratosthenes Sieve. Haskell allows, for example, to define natural numbers as successors of zero (thus implementing the Peano arithmetic) and then develop elementary number theory step-by-step, its theorems turning out to be algorithms and proofs -- their formal verifications.

This new approach, despite a number of teaching experiments around the world, is still mostly hypothetical. However, it raises a number of important methodological questions. For example, working in Haskell requires de-facto category-theoretic thinking (common in "research level" mathematics), not just the use of a language of naive set theory.

Meanwhile, there are claims that "without a cadre of people able and willing to engage in abstract thinking our technology-based society is doomed...” (Peter McB, a blog comment, 2007). Isn't it time to revisit the question of the purpose of mathematics education?

But Haskell provides an answer to another question: what is the role of ICT in teaching of more abstract chapters of mathematics and development in students a capability for abstract thinking?

Well, the answer is simple: students need to get first-hand experience of how ICT is made, not just how it is used.

Acknowledgements

On various occasions I was lucky to have a chance to discuss some points of this paper with Russ Abbott, Chris Budd, Jeff Kramer, Martin Hyland, Peter McBride, Chris Sangwin, and Seb Schmoller.

References

Association for Learnig Technology. (2011). *AoC/ALT Response to Royal Society Call for Evidence: Computing in Schools*. Retrieved on 28 April 2011 from http://www.webcitation.org/5yFs1OGBX.

Borovik, A. (2007) *Mathematics under the microscope*. Blog. http://micromath.wordpress.com.

Borovik, A. (2011). Information technology in university-level mathematics teaching and learning: a mathematician's point of view. *Research in Learning Technology, 19*(1), 73-85.

Doets, K. and J. van Eijck. (2004). *The Haskell road to logic, maths and programming*. London: King's College Publications.

Fried, M. (2011). Classroom assessment vs. student satisfaction. *Notices of the American Mathematical Society*, *58*(2), 229.

Hutton, G. (2007). *Programming in Haskell.* Cambridge: Cambridge University Press.

Intellect. (2011). *Intellect response to Department for Education call for evidence. Review of the national curriculum.* Retrieved 27 April 2011 from http://www.intellectuk.org/component/docman/doc_download/5127-intellect-response-to-national-curriculum-review-april-2011.

Kramer, J. (2007). Is abstraction the key to computing? *Communications of the ACM*, *50*(4), 37-42.

London Mathematical Society. (2011). *Use and misuse of information and computer technology in the teaching of mathematics at HE institutions: Position Statement*. Retrieved 24 April 2011 from http://www.lms.ac.uk/sites/default/files/Mathematics/policy_responses/ICT_statement.pdf.

Mitchel, S. (2011). Intellect: ICT classes should be scrapped. *PC Pro*, 20 April. Retrieved 25 April 2011 from http://www.pcpro.co.uk/news/education/366919/intellect-ict-classes-should-be-scrapped.

Nelson, E. (1977). Internal set theory, a new approach to NSA. *Bulletin of the American Mathematical Society*, *83*, 1165-1198.

Robert, A. M. (2003). *Nonstandard analysis*. Mineola, NY: Dover Publications.

Sangwin, C. (2006). *Assessing elementary algebra with STACK.* Retrieved 24 April 2011 from http://www.open.ac.uk/cetl-workspace/cetlcontent/documents/4607d31d634fd.pdf.

Sangwin, C. J., and P. Ramsden, (2007). Linear syntax for communicating elementary mathematics. *Journal of Symbolic Computation*, *42*(9), 902-934.

Sivakumar, N. (2001). *Introduction to inheritance, polymorphism in C#*. Retrieved 24 April 2011 from http://www.codeproject.com/KB/cs/csharpintro01.aspx.

S. Zucker, S. (2010). Evaluation of our courses. *Notices of the American Mathematical Society*, *57*(7), 821.