Selected Passages from Correspondence with Friends 1 no. 3 (2013), 9–12

THE STRANGE FATE OF ABSTRACT THINKING

ALEXANDRE BOROVIK

1. *Abstract thinking in mathematics and computer science*

I wish 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 [**3**]. An contributor to discussion of Kramer's paper in the blog *Mathematics under the Microscope* [**1**], 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 timely 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.

2. *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.

2010 *Mathematics Subject Classification* .

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 variety of classes linked by mechanisms of inheritance and polymorphism. This is 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 mathematics undergraduates. 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). Indeed 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 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 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. [**7**]

3. *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 this discussion.

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 [**5**, **6**] 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 [**4**] 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 by Fried [**2**]: his IT solution for teaching of vector calculus

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

But, in this 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. Interactive representation of mathematics via user-friendly interfaces is made possible by advances in computer science—but the latter requires a level of abstraction that far exceeds the one that can be provided by ICT tools for teaching and learning as they are evolving now.

References

1. A. Borovik, Mathematics under the microscope. Blog. `http://micromath.wordpress.com`. 2007.
2. M. Fried, Classroom assessment vs. student satisfaction. Notices AMS, 58 no. 2 (2011) 229.
3. J. Kramer, Is abstraction the key to computing? Communications of the ACM, 50 no. 4 (2007) 37–42.
4. London Mathematical Society. Use and misuse of information and computer technology in the teaching of mathematics at HE institutions: Position Statement. `http://tinyurl.com/lms-ict-caa`. 2011.
5. C. Sangwin, Assessing elementary algebra with STACK. `http://tinyurl.com/sangwin-stack`. 2006.
6. C. J. Sangwin, and P. Ramsden, Linear syntax for communicating elementary mathematics. Journal of Symbolic Computation, 42 no. 9 (2007) 902–934.
7. N. Sivakumar, Introduction to inheritance, polymorphism in C#. `http://tinyurl.com/sivakumar`. 2001.

Email `alexandre.borovik >>> at <<< gmail.com`