Cogito, Ergo Sum! Cognitive Processes of Students Dealing with Data Structures

Dan Aharoni

Department of Education in Technology and Science, The Technion, Israel
IBM Research Labs in Haifa, Israel

aharoni@netvision.net.il

Abstract

A research that has just recently been finished, investigated thinking processes that occur in the minds of students dealing with data structures. The research findings are pointed out in this paper, and two of them are elaborated. One is the phenomenon of *programming-context thinking*. This type of thinking stems from comparatively low level of abstraction gained by students in a data structures course. Programming-context thinking is the cause of other phenomena found in the research, and one such phenomenon — *perception of a data structure as static or dynamic* — is also elaborated. Implications for data structures instruction are discussed.

Apart from presenting the research results, this paper serves as an example of cognitive research — a kind of research that is still not broadly enough done in Computer Science Education. It is one purpose of this paper to manifest the need for more such research.

1 Introduction: The Need for Research of Cognitive Processes

"Cogito, ergo sum!" — "I think, therefore I am!" — said Descartes (Figure 1), and by doing so he focused our attention to the very essence of a human being, namely: *Thinking*. In a more professional fashion we might say that Descartes focused our attention to *cognitive processes*.

Cognitive processes research have long become a most important trend in educational research, especially in science education research. Cognitive processes are the backbone of *Constructivism*, a dominant approach in learning theories (see, for example, [9]), which has major implications for teaching [5] as well as for research.

The very heart of constructivism is the view of the learner

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advant -age and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE 2000 3/00 Austin, TX, USA
© 2000 ACM 1-58113-213-1/00/0003...$5.00

Figure 1. René Descartes (1596 – 1650)

as an active entity, and the view of knowledge as being constructed in the learner's mind, rather than being transmitted from the teacher to the learner [6]. It is an obvious conclusion from this approach, that research of learning products is not sufficient, and there is a need for research of cognitive processes that lead to these products.

Indeed, in Mathematical Education there is a substantial amount of study of mental processes. There is even a special community that deals with Psychology of Mathematics Education (PME); see, for example, [14]. Computer Science Education (CSE), on the other hand, is still in its infancy, compared to Mathematics Education, and there is still very little research done in the field of Psychology of Computer Science Education (Is this an opportunity to establish such a field — PCSE?).

Ben-Ari pointed out that constructivism is still not enough appreciated in CSE, and he showed "how the theory can supply a theoretical basis for debating issues and evaluating proposals" [2]. The present paper introduces some results of a research which purposes, questions and methods are dictated by the constructivist approach. The research focused on Data Structures (DS), and in accordance to constructivism it investigated mental processes in students dealing with DS. Due to space limitations, only two of the results are presented here. The results presented here may also serve as examples for outcomes of this kind of research, as well as a call for enlarging PCSE research in the future.

2. Research Design and Findings

The research — "*Undergraduate Students' Perception of Data Structures*" — focused on thinking processes which occurred when undergraduate students dealt with DS. As a first study of mental processes involved in dealing with DS, the research sought for phenomena specific to the investigated domain, phenomena that may be used in the future to build a more solid theory. Also, There is substantial evidence that *quantitative* research hardly provides an understanding of the knowledge and of the thinking processes in the learner's mind (An eye-opening example of this fact may be found in Erlwanger's study [7], and a broad discussion is given in [3] pp. 1–57). Hence the research made use of *qualitative* methods, a major paradigm in contemporary educational research [3, 10].

The research was a case study, and involved 9 Computer Science (CS) majors in a major Israeli university, participating in an introductory DS course. These students have already studied basic DS concepts in a previous CS introductory course ("Introduction to Computer Science").

Semi-structured observational interviews were used as the main data collection tool. The interview questions covered topics such as: data structures in general, arrays, stacks, queues, linked lists, and the construction of a data structure to fit the requirements of a given problem.

There is hardly any research on mental processes involved in thinking about CS concepts. However, the domain of DS is mathematical in its nature: It deals with abstract entities and operations on them, much like what is done in mathematics. Hence, methods and theoretical frameworks from research on mathematical thinking were used.

It should be pointed out, though, that there are subtle differences between DS entities and mathematical ones. Some of these differences lie in the way the term *abstraction* is used in math vs. CS, a topic elaborated by Leron [11]. For the sake of the following discussion, it will suffice here to mention one such difference discussed in [11] — the difference in what is taken to be the *opposite* of "abstract" in the two disciplines. In mathematics, a common answer is, "the opposite of abstract is concrete". Thus, if students in an abstract algebra course complain (as they frequently do) that the stuff is too abstract, a standard response would be to give a "concrete" example. In CS, in contrast, the opposite of "abstract" usually means "dealing with the details of implementation in a particular machine or in a particular programming language."

The research findings include *cognitive* factors as well as *affective* ones. Cognitive factors found were: Low level of abstraction gained by the students; programming-context thinking (see section 3.1); pragmatism (Seeking the practical use of each DS); "complexity above all"; using a specific programming language syntax within pseudo-code; perception of DS as static or dynamic (see section 3.2); perception of DS's ability to be empty; the idea that all the elements in a DS have the same type; conflicting mental structures for the same DS; constraint-oriented thinking (Solving a problem while referring to the problem's constraints, rather than starting with the problem demands and imposing the constraints later); visual representations of DS; conjectures on prototypes of DS categories; beliefs concerning DS.

Affective factors found in the research were: Avoidance of detailed algorithms, and avoidance of possible algorithms with high complexity.

The research findings couldn't, of course, be presented here to their full extent. Here they were only set out by their names. In the next section, two of them are discussed as examples, together with a discussion of their implications for DS instruction. Genuine interview excerpts (translated to English) are included; the names of the interviewees are fictitious.

3 Two of the Cognitive Processes Involved in Dealing with DS

3.1 Thinking Types Related to Programming

Let us consider the following question:

What is an array?

Notice that this question is completely general, in the sense that it doesn't refer to any particular kind of array. As such, a general answer may be expected. From the DS domain point of view, a "general answer" would be an answer that refers to an *abstract* array, something like this:

> An array is a collection of ordered pairs (index-set, value), where all the index-sets are distinct, together with the operations INSERT (inserting a new pair into the array) and GET (returning the value at a specified index).

This, of course, is not *the* correct answer, if there is such an answer anyway; it is merely *one possible* general answer.

The above question was posed to student interviewees, as well as to CS professionals. The majority of the answers may be summarized in Roy's answer:

> Roy: An array is a continuous area in the [computer's] memory, which holds elements of the same type. We can access each element by specifying its index, which is a whole number.

Let us analyze Roy's answer:

- Roy is talking about the array as being implemented in some computer's memory, namely: in some computer program. He uses **programming oriented thinking**.
- Moreover, Roy talks about the array as occupying a continuous area of the computer's memory, as holding elements of the same type, and the indices are seen as whole numbers. These properties don't necessarily hold for arrays in *any* programming language; they do hold, however, for a particular

programming language Roy usually uses — the C programming language. Thus, Roy uses **programming-language oriented thinking**, i.e. thinking tied to a specific programming language.

Here is another segment from an interview with Ann. She was asked to build some algorithm, again — a general algorithm, and not a computer program. She described her algorithm in which a stack was used. During the discussion she said ("`I`" is a shorthand for "`Interviewer`"):

Ann: What I'll do in the beginning, [...] I'll do POP until I get to the beginning of the stack.

I: *How do you know that you got the beginning of the stack, by the way?*

Ann: [Pauses for a few seconds] I know the beginning address [of the stack], don't I?

Ann's last answer is correct, thinking of the stack as being implemented in C. But this answer refers to the *implementation* of the stack, rather than to an *abstract* stack that could be used in the general algorithm she was building. The fact that she isn't referring to the abstract stack can be seen from her talk about the beginning address: Dealing with an abstract DS, there is no meaning whatsoever to an address. Ann's thinking is programming-language oriented too.

These were only two examples from a vast amount of evidence to programming oriented thinking and to programming-language oriented thinking. Due to space limitations no other examples will be presented here.

Let us summarize the three types of thinking related to programming: We saw **programming oriented thinking**, we saw **programming-language oriented thinking**, and, of course, we can talk of a **programming-free thinking**. We shall refer to the first two types as **programming-context thinking**.

As was discussed above, what differentiates between these three types of thinking is the level of abstraction they relate to, as can be seen in Figure 2.

It is important to understand, that there isn't any claim here whatsoever that the students were incapable of programming-free thinking; they could use such thinking — and at times they did, especially in situations where they *had no choice* but to think abstractly. The examples above only showed, that they used programming-context thinking whenever they could.

And, in fact, why shouldn't they think concretely? Here we come to one explanation of the phenomenon: The *convenience* of concrete thinking. They are used to the C programming language, at class, and for their homework. Pulling themselves away from the familiar environment and moving to an abstract discussion is something they were not used to since they were not asked to do so frequently enough. They haven't been challenged enough during the course to use abstraction, and so didn't feel the

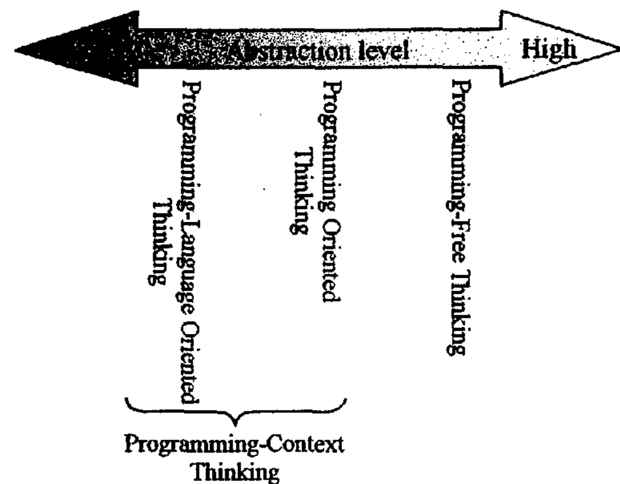

Figure 2. Abstraction levels of thinking types related to programming

need for it. Using the most convenient cognitive tools for problem solving exemplifies the *coping theory* presented in [12], and the *principle of necessity* presented in [8].

The three thinking types express stages of abstraction presented in the *Actions-Process-Object* model, which is considered today as one of the central models of concept formation. It is discussed by many researches, e.g. [4], [13], and a simplified version of it is illustrated in Figure 3.

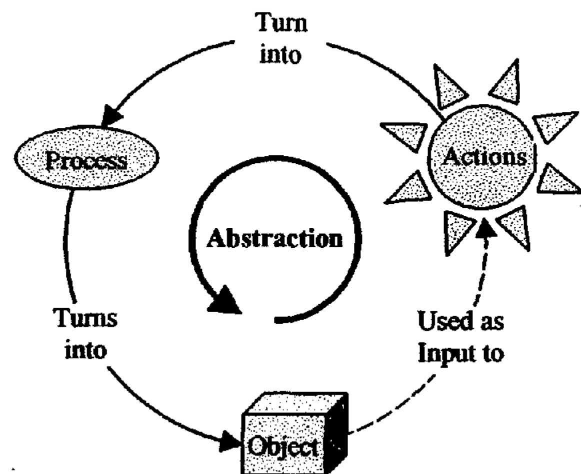

Figure 3. A simplified version of the Actions-Process-Object model

According to this model, the formation of a new concept of a mathematical entity begins with *actions* performed on some physical or mental objects. When the learner gains the ability to refer to these actions using symbols and in input/output manner, without carrying out the specific actions themselves, we say that the actions were transformed into a *process*. The last stage is transforming the process into an *object*: The learner can now refer to the entity much like she refers to a physical one. She sees it as

a whole static entity, recognizable at a glance, which may be thought of without any reference to its process view. Now the new formed object may be used as input to new actions which will be transformed to a process, then to a new — more abstract — object, and so forth. This whole procedure is, in fact, one procedure of abstraction.

Going back to our three types of thinking, it seems that *programming-free thinking* may be invoked only if the concept of the DS at hand has already been developed to its *object* stage, the only stage that enables thinking about abstract DS. If the concept is still in its *process* stage, we are witnessing *programming oriented thinking*, where the learner still has to think of the DS as being implemented within some program — not necessarily using some specific programming language. *Programming-language oriented thinking* is seen when the concept is still in its *actions* stage, where the learner must refer to the specific arrangement of the concrete DS in the computer's memory, to specific addresses within that DS implementation, and to specific operations performed within that implementation.

We can summarize, then, that DS concepts were developed in the students' minds to the stage of a process. Sometimes that stage hasn't even been fully formed, and the concepts are in transition from the actions stage to the process stage. There are times when the students can refer to abstract DS, but mostly they do not do so. This shows that the formation of the concept's object view has started, but the object is still a weak mental structure that doesn't take control unless there is no other option. It was argued that the students didn't develop an adequate level of abstraction because they haven't been sufficiently exposed to the need and usefulness of high abstraction levels. Thus, they are not skilled in posing *abstraction barriers* [1, 11] that are necessary for programming-free thinking.

The three types of programming related thinking are the base explanation for other phenomena found in this research. The next section demonstrates one such phenomenon.

3.2 Perception of a DS as Static or Dynamic

Let us go back to the question "what is an array?" Here is Joy's answer to that question, and a segment of the discussion that followed it:

Joy: Well, like, an area in the memory [...] like, like such a table in the memory, which is a specified place, like, a defined number of cells, [...] to hold information in some...

I: What does it mean a defined number?

Joy: Like, <u>something that cannot be changed</u> [emphasis added], and... Like, information can be saved and each cell can be accessed by, by its name, the index.

Joy sees the array as a *static* DS, in the common meaning of "its size cannot be changed". During the interviews Joy used the C programming language, and here she is talking about this language's mechanism for arrays, which are static data types. On another interview, where Joy dealt with a question of implementing an array using stacks only, she was asked:

I: OK. When is it worthwhile, anyway, using such algorithms, implementation of an array using a stack?

Joy: [Pauses for a few seconds] I don't know, in fact one can, like, make an array which size is not limited, because a stack — you can in fact continue it.

In this case, Joy associates with the array a new property not referred to up to now: The *dynamic* nature. And why? Because it is implemented using a stack and Joy sees a stack as inherently dynamic, and consequently — the array is seen as dynamic. She ignores the fact that previously she referred to an array as a static DS.

We see that when Joy refers to some DS, she associates with it either dynamic nature or static nature. She later summarized it as follows:

A stack can grow to some, like infinite length if you can say so, and an array is constant [in length]. In trees also if we build some tree it has a policy of a binary tree, search tree, etc., this time they have dynamic allocation, they can grow.

Each of the interviewees showed similar perception, and most of the time they referred to arrays as static, to linked lists, stacks and queues as dynamic, and to trees as dynamic but their nodes — static.

It seems that this situation is rooted in the fact that whenever possible, the students use programming-language oriented thinking. This kind of thinking causes the students to think of the DS as residing within some computer program. The students are used to the C programming language, and so when they refer to an array — they see C's array mechanism, which is static. When, on the other hand, they refer to stacks and queues — they cannot refer to them as predefined data types, since such data types are not included in C. In this case they have to invoke programming-free thinking, namely: Thinking about the *abstract* DS "stack" and "queue", which are dynamic. The same holds for trees: C doesn't have "tree" as a predefined data type, and so its abstract facet must be considered; but when implementing a tree, C's `structures` are usually used for nodes implementation, so the students can — again — use programming-language oriented thinking, which leads them to see these nodes as static.

4. Implications for Data Structures Instruction

In discussions with instructors of a DS course they specifically pointed out that understanding abstract DS was one of the main goals of the course. The lecturers also expressed their belief that students do not gain the expected ability to deal with abstract DS. The research presented here supported this belief with systematic evidence.

The finding that the students gained only low level of abstraction, compared to what was expected by the

instructors, and compared to what might be a more useful level of abstraction for problem solving, raised the need to accelerate the abstraction process during DS courses. As we saw, the phenomenon of low abstraction level stems from the fact that students were not sufficiently exposed to situations where abstract thinking would prove to be much more useful than less abstract one. Hence, a remedy may be the creation of such situations for the students.

One way of putting the students in the appropriate situation is to provide them with a computerized environment containing pre-prepared DS, such as arrays, stacks, queues, trees of different kinds, etc. The implementation of these DS shouldn't be exposed to the students, at least not in the beginning of the course. The students should get assignments from the teachers, so they can "play" with the DS much like they would do with concrete objects. Thus, this environment is used as a toolbox, where the tools are DS.

Such environment poses proper *abstraction barriers* for the students, who are unable to do so by themselves, as we saw. This way, it is conjectured, the students will be able to develop the needed level of abstraction. Only after that level of abstraction is developed, the course can enter its next stage, where implementations of DS are also studied.

5. Conclusion

The goal of this paper was twofold: First, it meant to bring before the reader some findings from a research about thinking processes taking place while students think of DS.

Second, this paper should draw CSE researchers attention to this kind of research, namely: Scrutinizing cognitive processes. The understanding of processes which underlay the thinking of our students is extremely important, since it may lead us to ways of improving their learning and aiding them in developing knowledge which is more useful for problem solving.

Let this paper serve as a call for more cognitive research in Computer Science Education in the (hopefully near) future.

Acknowledgements

I would like to thank Prof. Uri Leron from the Department of Education in Technology and Science, in the Technion, Israel Institute of Technology, who supervised this research, enlightened my way through it, and also provided me with comments about this paper.

References

[1] Abelson, H., & Sussman, G. (with Sussman, J.). *Structure and implementation of computer programs.* Cambridge, MA: MIT press, 1985.

[2] Ben-Ari, M. Constructivism in Computer Science Education. Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education (SIGCSE '98), 257–261, Atlanta, GA, 1998.

[3] Bogdan, R.C., & Biklen, S. K. *Qualitative research for education.* Boston: Allyn And Bacon, 1992.

[4] Breidenbach, D., Dubinsky, E., Hawks, J., & Nichols, D. Development of the process conception of function. *Educational Studies in Mathematics, 23,* 247–285, 1992.

[5] Brooks, J.G., & Brooks, M.G. *The case for constructivist classrooms.* Alexandria, Virginia: Association for Supervision and Curriculum Development, 1993.

[6] Davis, R.B., & Maher, C.A. What do we do when we "do mathematics"? In R. B. Davis, C. A. Maher & N. Noddings (Eds.), *Constructivist views on the teaching and learning of mathematics* (Journal for Research in Mathematics Education (JRME), monograph No. 4, chap. 8, pp. 65–78). The National Council of Teachers of Mathematics, Inc., 1990.

[7] Erlwanger, S.H. Benny's conception of rules and answers in IPI mathematics. *The Journal of Children's Mathematical Behavior (JCMB), 1* (2), 7–26, 1973.

[8] Harel, G. Two Dual Assertions: The first on learning and the second on teaching (Or vice versa). *The American Mathematical Monthly, 105,* 497-507, 1998.

[9] Kilpatrick, J. What constructivism might be in mathematics education. In J.C. Bergeron, N. Herscovics, & C. Kieran (Eds.), *Proceedings of the eleventh International Conference for the Psychology of Mathematics Education (PME11): Vol. I* (pp. 3–27). Montréal, 1987.

[10] LeCompte, M.D., & Preissle, J. *Ethnography and qualitative design in educational research.* San Diego: Academic Press, 1993.

[11] Leron, U. Abstraction barriers in mathematics and computer science. In J. Hilel (Ed.), *Proceedings of the third International Conference for Logo and Mathematics Education.* Montréal, 1987.

[12] Leron, U., & Hazzan, O. The world according to Johnny: A coping perspective in mathematics education. *Educational Studies in Mathematics, 32,* 265-292, 1997.

[13] Sfard, A. 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.

[14] Zaslavskey, O. (Ed.) *Proceedings of the 23rd conference of the International Group for the Psychology of Mathematics Education (PME23).* Haifa, Israel: Technion, Israel Institute of Technology, 1999.