Reducing Abstraction in High School Computer Science Education: The Case of Definition, Implementation, and Use of Abstract Data Types

VICTORIA SAKHNINI and ORIT HAZZAN
Technion – Israel Institute of Technology

The research presented in this article deals with the difficulties and mental processes involved in the definition, implementation, and use of abstract data types encountered by 12th grade advanced-level computer science students. Research findings are interpreted within the theoretical framework of *reducing abstraction* [Hazzan 1999]. The article describes the research setting and findings and concludes with some pedagogical implementations.

Categories and Subject Descriptors: K.3.0 [**Computer and Education**]: General

General Terms: Algorithm, Design, Human Factors

Additional Key Words and Phrases: problem solving, abstraction, reducing abstraction, computer science education, abstract data types

ACM Reference Format:
Sakhnini, V. and Hazzan, O. 2008. Reducing abstraction in High School Computer Science Education: The case of definition, implementation and use of abstract data types. ACM J. Educ. Resour. Comput. 8, 2, Article 5 (June 2008), 13 pages. DOI = 10.1145.1362787.1362789. http://doi.acm.org/10.1145.1362787.1362789.

1. INTRODUCTION

A review of the existing computer science (CS) education literature indicates that relatively few studies address the issue of mental processes in general, and mental processes of high school CS students in particular. Furthermore, as far as our literature survey indicates, no study has been conducted to date on high school students' understanding of abstract data types. This article attempts to close this gap, at least partially.

Specifically, the current research addresses the difficulties and mental processes involved in the definition, implementation, and use of abstract data types that are encountered by 12th grade CS students when designing and

Table I. Reducing Abstraction in High School and Undergraduate Mathematics and Computer Science Education.

	Mathematics	**Computer Science**
High school education	Various topics, e.g., functions [Hazzan and Zazkis 2005]	The study presented in this article.
Undergraduate education	Differential equations [Raychaudhuri 2001] Abstract algebra [Hazzan 1999]	Data structures [Aharoni 1999, 2000] Computability [2003a; 2003b] Graph theory [Hazzan and Hadar 2005]

constructing software systems. The article is based on the PhD research work conducted by the first author under the supervision of the second author. The research findings are interpreted within the theoretical framework of *reducing abstraction* [Hazzan 1999].

Until now, the theoretical framework of reducing abstraction has been applied to explain undergraduate students' understanding of CS subjects such as computability [Hazzan 2003a; 2003b], data structures [Aharoni 1999; 2000], and graph theory [Hazzan and Hadar 2005], as well as various mathematical concepts [Hazzan 1999; Raychaudhuri 2001; Hazzan and Zazkis 2005]. In this study, the framework of reducing abstraction is applied for the first time, as far as we know, in the interpretation of *high school* students' understanding of *CS concepts*, namely abstract data types. Table I summarizes the position of the research presented in this article with respect to the use of the theoretical framework of reducing abstraction.

Section 2 of this article describes the research background. Section 3 presents the research findings within the theoretical framework of *reducing abstraction*. In Section 4 presents the contribution of this research and some pedagogical implications.

2. RESEARCH BACKGROUND

2.1 Research Goal and Research Questions

The goal of the current research was to document and analyze difficulties and thinking processes involved in the definition, implementation, and use of abstract data types by high school students when engaged in the design and construction process of software systems.

In order to achieve the research goal, two specific research questions were formulated:

(1) What difficulties do students face when solving problems dealing with abstract data types?
(2) What heuristics do students use when solving problems dealing with abstract data types?

2.2 The Investigated Curriculum: "Software Design"

The research population included Israeli high school students who were studying Software Design, an advanced-level CS study unit. According to Gal-Ezer and Harel [1999], the unit's main goal is to teach the basic principles of software system design. It also develops abstract thinking skills, especially by defining, selecting, and using abstract data types. Particular data structures are taught, such as lists, stacks, and binary trees.

The problems students are asked to solve in this unit, as well as the solutions to these problems, do not have a repeated or common pattern. Most of the problems require students to define a new abstract data type for the design and construction of a software system. In most cases, there is also more than one way to implement the requirements of the given system and the problem has a variety of possible solutions.

2.3 Data Collection and Analysis

In order to learn about students' mental processes, a qualitative research approach was used. Data were collected in interviews with 16 students from six classes, in which the students were asked to solve relevant problems in a think-aloud fashion. The interviews enabled to deepen our understanding of the students' mental processes and of difficulties they face when involved in problem-solving processes related to abstract data types.

The 16 students were chosen from different levels. Their CS grades prior to the interviews were the main criteria for their selection. Out of the 16 students, 4 were very good students (with a grade between 85 and 100), 6 were rather good students (with a grade between 70 and 84), and 6 were weak students (with a grade between 50 and 69 out of 100). In addition, the selected students were cooperative and articulate and agreed to present their work and express their thoughts during the interviews. Each student was interviewed five times, each for 60-90 minutes.

In order to recognize students' mental processes during problem solving processes, the students were asked in the interviews to solve in a think–aloud fashion advanced questions that requested them to construct complex abstract data types. During the interviews the students could use paper and pencil. Though the tasks and the associated questions were prepared in advance, the interviews were open and the interviewees could express their thoughts openly. When needed, calcifications and follow-up questions were asked. All the interviews were recorded and documented by field notes. The field notes were aimed at documenting the interview flow and other reflective thoughts. In addition, students' writings and sketches were collected and analyzed. The interviews were transcribed and analyzed in a qualitative-inductive fashion as is accepted in qualitative research.

To further validate the data gathered in the interviews, observations made in three 12^{th} grade CS classes (composed of a total of 75 students) that studied the Software Design unit. In this article, however, we present data from the interviews.

3. RESEARCH RESULTS: REDUCING ABSTRACTION LEVELS WHEN SOLVING PROBLEMS ABOUT ABSTRACT DATA TYPES

During the data analysis process, Hazzan's theoretical framework of reducing abstraction [Hazzan 1999] was found to suit the analysis of students' mental processes involved in problem-solving processes related to abstract data types. According to this framework, in order to make unfamiliar concepts more familiar, students tend to reduce the level of abstraction of the concepts involved in different problem-solving situations. Specifically, we identified three mental processes by which students reduce the level of abstraction. These mental processes were defined previously by Hazzan and will be further explained later in the article:

—Abstraction level as a reflection of the quality of the relationship between the object of thought and the thinking person;

—Abstraction level as a reflection of the process-object duality;

—Abstraction level as a reflection of the degree of complexity of the thought concept.

It was also found that this framework can be used to explain students' *conscious* strategies in problem-solving situations, and not only *unconscious* mental processes, as originally described by Hazzan. Nevertheless, in both cases the level of abstraction of the concepts involved is reduced.

In what follows, we outline the theoretical framework with respect to students' understanding of abstract data types. Specifically, Sections 3.1 through 3.3 illustrate the ways in which students reduce abstraction according to the aforementioned three processes. Each section contains two subsections: "Unintentional Reduction of Abstraction," which deals with students' unconscious mental processes, and "Intentional Reduction of Abstraction," which addresses students' conscious strategies. The different findings are illustrated using excerpts from the interviews. All names of students quoted are fictional.

3.1 Abstraction Level as a Reflection of the Quality of the Relationship between the Object of Thought and the Thinking Person

Hazzan [2003a] describes this interpretation of abstraction as one that stems from Wilensky's [1991] assertion that whether something is abstract or concrete (or anything on the continuum between those two extremities) is not an inherent property of the object or thing itself "but rather a property of a person's relationship to an object" (p. 198). According to this interpretation of abstraction, a different level of abstraction is observed for each concept and for each person, which reflects previous experiential connections between the two. Specifically, the closer a person is to an object and the more connections he or she has formed to it, the more concrete (and the less abstract) he or she will feel towards it. Based on this perspective, some student mental processes can be explained by the students' tendency to make an unfamiliar idea more familiar or, in other words, to make the abstract more concrete.

Ginat [2001], Muller [2005] and Winslow [1996] presented similar findings, describing how students, in order to make unfamiliar concepts more familiar,

look for analogous problems, solutions, or concepts that might help them understand the unknown concept. In many cases, however, this step is not executed correctly.

3.1.1 *Unintentional Reduction of Abstraction.* This sub-section illustrates how students tend to make an unfamiliar concept more familiar by hanging on to a concept with which they are familiar from previous experience.

Example 1. Students believe that elements of a list are arranged and allocated in the computer's memory according to the same order by which they are arranged in a real list. In other words, the second element of the list is located close to the first element, the third element is located close and after the second element, and so on. Following are two excerpts from an interview with Roni that illustrate this phenomenon:

Excerpt 1:

Researcher: Please explain or define a list.

Roni: A list is a collection of elements, a list anchor,[1] and the end of the list.

Researcher: Is this a complete definition?

Roni: Oh, no. There is an order to the elements; we know which is first, which is second, and so on until the last one. If we start from the anchor we can scan the list till the end.

Researcher: How can you reach the anchor?

Roni: There is an operation called "list-anchor" that accepts a name of a list and returns a pointer to the anchor.

Researcher: How can you scan a list?

Roni: There is an operation called "list-next" that accepts a list and a pointer to an element in the list and returns a pointer to the next element in the list as it is located in the memory.

Researcher: Can you explain in greater detail what you mean by "next element in the list as it is located in the memory"?

Roni: Yes, sure. The list elements are located in the computer memory in the same order as they are arranged logically; otherwise we would not be able to find the next element.

Researcher: Why do you think that they should be located physically in the memory as they are logically?

Roni: Because that's how a list should be. Look at a list of student names . . . We can see the names ordered as they should be, the first one is on the top of the list, then the second one, and so on until the last one.

[1]An anchor is a metaphor used in this unit for the beginning of a list.

Excerpt 2:

Researcher: What is the run-time complexity of the operation "list-delete"?

Roni: It's O(n).

Researcher: What do you mean by n?

Roni: n is the number of the elements in the list.

Researcher: Why O(n)? Can you explain it to me please?

Roni: Because when we delete [an element] from the list, all of the elements must move back in order to remain positioned one after the other, with no spaces between them.

Roni tried to understand the unfamiliar abstract data type "list" by hanging on to a concept familiar to him from his daily life, for example, a list of names, in which the list's elements are physically located one after the other. The abstract data type "list" is not, however, the same as this name list concept.

3.1.2 *Intentional Reduction of Abstraction.* In order to make the unfamiliar more familiar, students consciously apply several strategies that use known and analogous patterns they have learned before. In most cases, students look for familiar problems that might help them solve the new problem they face. Indeed, this is a successful heuristics in problem-solving situations [cf., Polya 1973; Schoenfeld 1985].

Example 2. This example illustrates a heuristic used by students in order to make the unknown more familiar. According to this heuristic, students add properties, data, or rules to a given problem in order to make it more familiar; solve the known problem; and after solving the new but familiar problem, they neutralize the added elements in order to present a solution for the original problem.

Dan was asked to solve the following problem: "Write an algorithm that accepts two lists, L1 and L2. If L1 is a sub-list of L2 (elements of which are not necessarily in the same order), the algorithm returns the value 'true'. Otherwise the algorithm returns the value 'false'." Here is an excerpt from an interview with Dan, after he solved this problem:

Researcher: Can you please explain your solution?

Dan: Yes, I did not succeed in solving this problem so I added an assumption that the two lists are sorted. Then I solved the problem by scanning the two lists at the same time and checking if each element in L1 is also in L2.

Researcher: But the question does not mention that the two lists are sorted.

Dan: Well, I know, but as I said before I don't know how to do it if the two lists are not sorted, but I think I know how to neutralize this condition that I added before.

Researcher: How?

Dan: I will add code that sorts these two lists, the way we learned already, prior to my solution.

Dan had already learned how to write an algorithm that accepts two sorted lists and checks if the first list is a sub-list of the second one. In this case, Dan was given a new, hence unfamiliar, problem. In order to make the problem familiar, he added a condition to the new problem and made it similar to the familiar one. Then, in order to solve the given problem, after completing the solution of the more familiar problem, he neutralized the additional condition by sorting the two lists first.

3.2 Abstraction Level as a Reflection of the Process-Object Duality

The interpretation of reducing abstraction discussed in this sub-section is based on the process-object duality, suggested by some theories of concept development in mathematics education [Beth and Piaget 1966; Dubinsky 1991; Sfard 1991, 1992]. Theories that discuss this duality distinguish mainly between *process conception* and *object conception* of mathematical notions. Dubinsky [1991] describes the transition from process conception to object conception as a "conversion of a (dynamic) process into a (static) object." Process conception implies that a mathematical concept is regarded "as a potential rather than an actual entity, which comes into existence upon request in a sequence of actions" [Sfard 1991, p. 4]. When conceiving of a mathematical notion as an object, this notion is captured as a consolidated entity. Thus, it is possible to examine it from various points of view, to analyze its relationships to other mathematical notions, and to subject it to various operations. According to these theories, when a mathematical concept is learned, its conception as a process precedes its conception as an object, and is less abstract than it [Sfard 1991]. Based on this interpretation of abstraction, the conception of a concept as a process can be interpreted as being on a lower level of abstraction than its conception as an object; in other words, the abstraction level is reduced.

3.2.1 *Unintentional Reduction of Abstraction.* This interpretation of abstraction reduction is manifested in this research mainly by the students' tendency to use detailed code in a specific programming language, which is familiar to them from their previous studies and reflects process conception, rather than to use given operations that capture the nature of concepts and their use reflect object conception of the said concepts.

Example 3. *list-next(L,p)* is an interface operation defined in the study unit Software Design, addressed in this article. It operates on lists and moves a pointer to the next element in the list. In the context of this example, students tend to write p:=pˆnext, which is the expression of this operation in Pascal, the programming language they are familiar with, instead of using the operation *list-next (L,p)*. In the following excerpt Tali explains the source of this tendency and indicates the way it fits into the framework of reducing abstraction.

Researcher: Please explain this line to me: p:=pˆnext.

Tali: Well, I want to move to the next element in the list, and this is the way to do it, isn't it?

Researcher: Let's look at the interface of the list unit.

Tali: Ok, I did! What's wrong? Look here, in the implementation part: In order to move to the next element, you write p:=pˆnext;

Researcher: What about p:=list_next(L,p)? In the interface it says list_next (L,p).

Tali: Yes, but I prefer p:=pˆnext. I learned all about pointers in Pascal and I have written many programs that work with lists using pointers, so why not? This is Pascal. We must use pointers in order to work with lists.

Researcher: What if we write p:=list_next(L,p) instead of p:=pˆnext. Is this still a correct answer?

Tali: Well . . . yes, but . . . I don't know . . . Still, I'm more familiar with pointers and I am sure that it works correctly.

3.2.2 *Intentional Reduction of Abstraction.* Intentional reduction of abstraction is manifested in this interpretation of abstraction by the students' tendency to use only basic operations and interfaces that were previously taught in class, instead of defining and using new operations that simplify the solution, an approach that would reflect an object conception of the concepts. It should be mentioned, however, that although the solution obtained using this conscious approach is in most cases correct, it does not reflect good programming style.

Example 4. Ron, who adopted this approach, was interviewed about it. Here is how he explains this approach:

Researcher: Can you please explain your solution?

Ron: Why? Is it incorrect?

Researcher: No, it is correct. But what I mean is, why didn't you identify new operations to help you to reach a simpler solution?

Ron: I prefer to use the basic operations; I know them well, I know how to use them, and they are very basic and simple operations. Why complicate myself with new complex ones?

3.3 Abstraction Level as a Reflection of the Degree of Complexity of the Concept of Thought

According to this interpretation of abstraction levels, the more complicated a concept is, the more abstract it is. Accordingly, this subsection illustrates how students reduce the level of abstraction by reducing the complexity of concepts with which they think. This is done (either consciously or unconsciously) by breaking down an unfamiliar concept of thought into its known components and solving the known subcomponents, that is, solving problems with simpler, less abstract concepts than the original ones. This discussion is especially relevant for the analysis presented in this article, since it addresses compound concepts such as list of trees and stack of queues.

3.3.1 *Unintentional Reduction of Abstraction.* The unintentional reduction of abstraction in this case is demonstrated by the students' tendency not to consider all of the components, data, and limitations mentioned in a given problem, but instead, to first deal with only part of the issues—those that are known. In other cases, students tend to break problems down into known subproblems and solve each subproblem independently, without combining the subproblem solutions into a solution of the original problem. These two cases are illustrated in Examples 5 and 6, respectively.

According to Schoenfeld [1985], mistakes such as that described above can be classified as mistakes made at the control level. In the same spirit, Taconis et al. [2001] noted that, in addition to basic knowledge, solving problems requires meta-cognitive skills such as critical thinking. In addition, the solution must be checked to see whether it addresses all of the components, data, and restrictions mentioned in the problem. As we see here, students tend to skip this step in this manifestation of reducing the level of abstraction.

Example 5. This example illustrates students' tendency not to consider all of the components, data, and limitations mentioned in a given problem, but rather to deal first with only part of the issues—those that are known.

Sarah was asked to write an algorithm that accepts two lists of students, S1 and S2, and returns a list that includes names of all the students in both lists, but without repetitions (i.e., students who appear in both lists should only be listed once in the algorithm's output list). The following excerpt is taken from an interview with Sarah that took place after she finished solving the problem.

Researcher: Can you explain how you solved this problem?

Sarah: Well . . . yes . . . I recalled that we did something similar when we wrote an algorithm in class that accepts two lists of numbers and merges them into one list, so I did something like that.

In this excerpt, Sarah *unconsciously* reduced the problem complexity by ignoring the requirement "without repetition" that appeared in the problem and dealing only with the requirements that are already known.

Example 6. Students' tendency to break a problem down into known subproblems and solve each subproblem independently, without combining the subproblem solutions to a solution of the original problem, is illustrated by an excerpt taken from an interview with Dan. Dan was asked to solve a problem concerning a simpler version of the Domino game whereby tiles are set only in a head-to-head manner (i.e., so that they form a straight line).

One of the problem requirements was to write an algorithm that accepts a list of domino tiles and another tile and prints "left" if the tile can be placed (according to the rules) only on the left-hand end of the domino line, "right" if it can be placed only on the right-hand end, "both" if it can be placed on both ends, and "no" if the tile cannot be placed. After Dan finished solving the problem, he was asked to explain his solution:

Researcher: Can you please explain your solution?

Dan: Yes, first I wrote a routine that checks if we can place the tile on the left-hand end; then I wrote a routine that checks if we can place the tile on the right-hand end; then I wrote a routine that checks if we can place the tile on both ends. Finally, I wrote the main algorithm that calls the first routine, then the second, and then the third.

As we can see, if a domino tile can be placed on both ends, Dan's algorithm will print "left", "right", and "both". In the context of the discussion presented in this article, Dan solved each subproblem independently, but did not combine them to obtain a solution for the original problem. Had he done so, the algorithm would print only the word "both". According to the interpretation of reducing abstraction discussed in this sub-section, Dan's unintentional failure to see the problem as a whole, remaining at the detail level without considering the required connections between the subproblems, reflects a reduction of the level of abstraction.

3.3.2 *Intentional Reduction of Abstraction.* The Divide and Conquer method is a strong tool for solving complicated problems: All that is needed is to break down a problem into subproblems, solve the subproblems and merge those solutions into a solution for the original problem. Thus, a direct solution for a complicated problem is avoided.

The Divide and Conquer method can be described as an intentional technique for reducing the level of abstraction since it reduces the concept complexity and, as mentioned before, the more complicated a concept is, the more abstract it is. Accordingly, dividing a complicated problem down into subproblems (hence, less complicated problems) implies that the level of abstraction is reduced, albeit only temporarily.

In the context of the research presented in this article, students use this approach in one of two intentional ways: (a) By dividing the unknown complex problem into known subproblems, solving each subproblem and then combining the subproblem solutions to obtain a solution for the original problem; or (b) by first treating only the familiar parts of the problem and then improving the solution by dealing with the parts not dealt with during the first phase of the solution. These two cases are illustrated in Examples 7 and 8, respectively.

Example 7. This example illustrates how students divide an unknown complex problem into known subproblems, solve each subproblem separately, and then combine the subproblem solutions to obtain a solution for the original problem.

Jacob was asked to solve the problem presented in Example 6. He wrote three routines as in Example 6. Then, when he wrote the main algorithm, he combined the subproblem solutions to obtain a correct solution for the original problem as described below:

The algorithm first checks whether the tile can be placed on the both ends of the domino line.

If yes, it prints "both",

else it checks whether the tile can be placed on the right-hand end.
If yes, it prints “right”,
else it checks whether the tile can be placed on the left-hand end.
If yes, it prints “left”.
else it prints “no”;
End of algorithm.

Unlike Dan, whose strategy was described in Example 6 and who solved each subproblem independently but did not combine them correctly to obtain a solution for the original problem, Jacob worked on the detail level only during the first phase of the solution process. In other words, he reduced the level of abstraction only temporarily: After working with the details, he considered the connections between the different partial solutions and constructed a correct solution for the original given problem.

Example 8. This example illustrates how students first address only the problems’ known parts and then improve the solution by dealing with the parts they did not deal with during the first phase.

Mary was asked to write an algorithm that implements a given, 7-rule game. Mary explained in her interview that she is familiar with four of the seven rules and knows how to deal with them.

Mary: I know how to deal with four of the seven rules, so I will start writing the solution for these four rules. At least I will submit part of the solution; it’s better than nothing.

After she had written the solution for the four rules she said:

Mary: Well, now, I have to deal with the rest of the rules, three of them.

Mary then continued to solve the problem by improving her solution, and dealt with the rules she had decided not to deal with during the first phase of the solution process.

4. CONCLUSIONS

The findings presented in this article address problem-solving situations involving abstract data types. The research findings contribute to the existing knowledge regarding thinking processes in general, and those of CS high school students in particular. Specifically, this research illustrates that the theoretical framework of reducing abstraction, originally developed in the context of mathematics education, can also be used to explain phenomena observed in high school CS education.

The findings have pedagogical implications as well, as described in what follows. It should be clear, however, that these recommendations have yet to be examined in a research framework.

First, according to Section 3.1, teachers should be aware that students tend to deal with unknown problems by relying on terms and ideas that are familiar to them either from their everyday life or from their previous studies. Therefore,

(1) Differences between new problems and familiar ones should be explicitly discussed with the students.
(2) Students should be exposed to vast number of problems of different types and levels in order to improve their ability to deal with unfamiliar problems. Specifically, students should be exposed to many *analogous* problems in order to learn how to identify analogies between problems and to exploit the knowledge derived from the already solved problems. Students should also be exposed to problems that have a *false analogy*, which does not actually exist even though some clues might indicate otherwise.

Second, according to Section 3.2, in order to guide students to conceive of abstract data types as objects, we recommend separating the definition and use of abstract data types from their implementation. One way to do this is to teach abstract data types early on, without teaching their implementation.

Third, according to Section 3.3, students should be exposed to many problem solving methods, strategies and tools, such as the "Divide and Conquer" approach, mentioned in Section 3.3.2.

In general, and in line with Polya's recommendations related to problem solving processes [Polya 1973], in order to educate students to become problems solvers, they should be acquainted with tools and methods for (1) understanding a given problem; (2) solving the given problem; and (3) checking the correctness of their solutions in order to increase their control over their problem-solving processes.

REFERENCES

AHARONI, D. 1999. Undergraduate students' perception of data structures. PhD thesis, Technion – Israel Institute of Technology.

AHARONI, D. 2000. Cogito, Ergo Sum! Cognitive processes of students dealing with data structures. In *Proceedings of the ACM 31st SIGCSE Technical Symposium on Computer Science Education (SIGCSE'00)*, S. Haller, Ed. Austin, TX.

BETH, E. W. AND PIAGET, J. 1966. *Mathematical Epistemology and Psychology*. D. Reidel, Dordrecht, The Netherlands.

DUBINSKY, E. 1991. Reflective abstraction in advanced mathematical thinking. In *Advanced Mathematical Thinking*, D. Tall, Ed. Kluwer Academic Press, 95–123.

GAL-EZER, J. AND HAREL, D. 1999. Curriculum and course syllabi for high-school computer science program. *Comput. Sci. Educ. 9*, 2, 114–147.

GINAT, D. 2001. Early algorithm efficiency with design patterns. *Comput. Sci. Educ. 11*, 2, 89–109.

HAZZAN, O. 1999. Reducing abstraction level when learning abstract algebra concepts. *Educ. Studies Math. 44*, 71–90.

HAZZAN, O. 2003a. Reducing abstraction when learning computability theory. *J. Comput. Math. Sci. Teach. 22*, 2, 95–117.

HAZZAN, O. 2003b. How students attempt to reduce abstraction in the learning of mathematics and in the learning of computer science. *Comput. Sci. Educ. 13*, 2, 95–122.

HAZZAN, O. AND HADAR, I. 2005. Reducing abstraction when learning graph theory. *J. Comput. Math. Sci. Teach. 24*, 3, 255–272.

HAZZAN, O. AND ZAZKIS, R. 2005. Reducing abstraction: The case of school mathematics. *Educ. Studies Math. 58*, 1, 101–119.

MULLER, O. 2005. Pattern oriented instruction and the enhancement of analogical reasoning. In *Proceedings of the International Workshop on Computing Education Research (ICER'05)*. 57–67.

POLYA, G. 1973. *How To Solve It*. Princeton University Press, Princeton, NJ.

RAYCHAUDHURI, D. 2001. The tension and the balance between one mathematical concept and student constructions of it: Solution to a differential equation. Ph.D. thesis, Simon Fraser University, Vancouver, Canada.

SCHOENFELD, A. H. 1985. *Mathematical Problem Solving*. Academic Press: Orlando, FL.

SFARD, A. 1991. On the dual nature of mathematical conceptions: Reflections on processes and objects as different sides of the same coin. *Educ. Studies Math. 22*, 1–36.

SFARD, A. 1992. Operational origins of mathematical objects and the quandary of reification – The case of function. *The Concept of Function – Aspects of Epistemology and Pedagogy*, E. Dubinsky and G. Harel, Eds. MAA Notes.

TACONIS, R., FERGUSON-HESSLER, M. G. M., AND BROEKKAMP, H. 2001. Teaching science problem solving: An overview of experimental work. *J. Res. Sci. Teach. 38*, 4, 442–468.

WILENSKY, U. 1991. Abstract meditations on the concrete and concrete implication for mathematical education. In *Constructionism*, I. Harel and S. Papert, Eds. Ablex Publishing Corporation, Norwood, NJ, 193–203.

WINSLOW, E. 1996. Programming pedagogy – A psychological overview. *SIGCSE Bull. 28*, 3, 17–22.

Received August 2007; revised December 2007; accepted December 2007