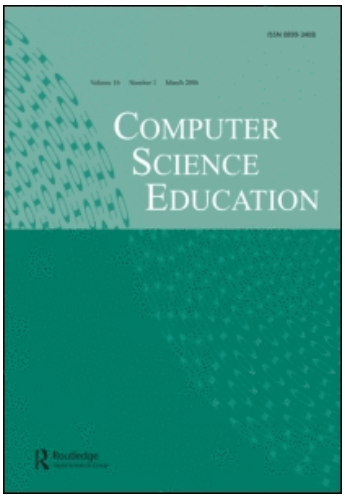

Computer Science Education
Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t713734307

Reductive thinking in computer science

Michal Armoni [a]; Judith Gal-Ezer [a]; Orit Hazzan [b]
[a] Computer Science Department, The Open University of Israel, [b] Department of Education in Technology and Science, Technion—Israel Institute of Technology,

Online Publication Date: 01 December 2006

To cite this Article Armoni, Michal, Gal-Ezer, Judith and Hazzan, Orit(2006)'Reductive thinking in computer science',Computer Science Education,16:4,281 — 301
To link to this Article: DOI: 10.1080/08993400600937845
URL: http://dx.doi.org/10.1080/08993400600937845

Computer Science Education
Vol. 16, No. 4, December 2006, pp. 281–301

Routledge
Taylor & Francis Group

Reductive Thinking in Computer Science

Michal Armoni[a]*, Judith Gal-Ezer[a] and Orit Hazzan[b]
[a]*Computer Science Department, The Open University of Israel;* [b]*Department of Education in Technology and Science, Technion—Israel Institute of Technology*

This paper discusses the role of reduction in computer science and describes a study on undergraduate students' perception of the concept of reduction. Specifically, based on an analysis of students' answers to questions addressing different computer science topics, we present several findings regarding the ways in which undergraduate students conceive of and apply reduction. These findings can be interpreted within the framework of the tension that exists between the following two factors: The need to think in terms of high levels of abstraction, on one hand, and the fact that reduction introduces a new approach to be used in problem-solving situations, on the other. In addition the paper suggests several teaching applications.

1. Introduction

Reduction is a problem-solving heuristic useful both for the theoretical aspects of computer science (CS), such as computability and algorithmics, as well as for many other CS facets, such as software design. Essentially, solving a problem by reduction means transforming it into a simpler problem (or problems) with a known solution, and constructing, or deducing, the solution to the original problem based on the solution(s) of the problem(s) to which the original problem was reduced.

This paper presents a study that addresses reductive thinking of undergraduate CS students in different contexts. First, we examined the tendency of undergraduate CS students, at different stages of their studies, to use reductive solutions in problem-solving situations in a variety of CS areas. Results of such a study can teach us about the development of a reductive mode of thinking by undergraduate CS students. Second, based on studies in mathematics and in science education that show that transfer—both inter-disciplinary and intra-disciplinary—is problematic (e.g. Noss & Hoyles, 1996; Nunes, Schliemann, & Carraher, 1993), we checked whether students transfer their reductive thinking from the area of algorithmics, in which reduction is usually taught explicitly, to the area of formal language theory, in which reduction is not necessarily taught explicitly despite the fact that it is an effective problem-solving heuristic in this area.

*Corresponding author. 108 Ravutski St., Raanana, 43107, Israel. E-mail: michal@openu.ac.il

ISSN 0899-3408 (print)/ISSN 1744-5175 (online)/06/040281-21 © 2006 Taylor & Francis
DOI: 10.1080/08993400600937845

Since reduction is a useful scientific problem-solving heuristic in general, and is used widely in CS in particular, we suggest that CS students should learn this heuristic as part of their undergraduate studies. Accordingly, based on the results of this study, we propose some didactic approaches aimed at developing reductive thinking.

In section 2 we discuss reductive thinking as a habit of mind. In section 3 we present the research setting. Section 4 presents and discusses the research results, interpreting the findings within the framework of the tension between the need to think in terms of high level of abstraction, and the introduction of reduction as a new problem-solving approach. Based on these results, section 5 suggests teaching applications and concludes.

2. Reduction as a Habit of Mind in CS

Habits of mind are heuristics, problem-solving approaches (Cuoco, Goldenberg, & Mark, 1997). In the context of CS, we talk about abstraction, reduction, successive refinements, and so forth, all of which are very helpful approaches in many problem-solving situations; at the same time, however, such concepts cannot be conveyed by a rigorous set of rules that can be applied automatically in different concrete problem-solving situations. Rather, in order to apply them successfully, one must gain experience in their application, be aware of their potential contribution, and recognize which heuristic might be helpful in different situations.

Reduction is one of the most broadly useful habits of mind used to carry out arguments, mainly in computability theory. A reductive strategy reduces the complexity of a solution by recognizing building blocks—subproblems with known solutions—and using them as black boxes. This enables us to ignore the details of the solution of the reduced-to problem, and rely on the fact (proven in some other context) that such a solution exists and that it can be obtained if necessary. For example, if in order to develop a solution for algorithmic problem A, we can reduce problem A to another algorithmic problem B with a known solution, we can use the solution to B as a black box, relying on its correctness. In contrast, if we construct a new solution for problem A, we have to prove the correctness of all its components (even when the solution is based largely on B's known solution and only slightly alters it).

Reduction may be one of those habits of mind that lead us to think about a problem in terms of a higher level of abstraction. Such a perspective offers the thinker a more global view of the problem, and enhances the possibility of strategic planning and an intuitive feel for the problem. The difference between the two ways of problem-solving—reductive and direct—is related to our limited capacity to simultaneously access large amounts of information from our short-term memory (Miller, 1956). When dealing with a problem on a lower level of abstraction, we have no free space left for the broader picture; hence we tend to "lose the forest for the trees." In contrast, when we think in terms of high abstraction level, we may temporarily relinquish some of the precision, but in exchange, we gain on the global and intuitive

side. Among other advantages of this habit of mind, thinking in terms of reduction allows us to focus on structures in terms of their properties rather than their actual components.

Reduction is expressed in several contexts across the CS curriculum. In algorithms and in computability and complexity theories, reduction is usually one-to-one, from the original problem to an already-solved problem. In computability and complexity theories, reductions are used to prove undecidability or completeness of algorithmic problems, while in algorithms, reductions are used to obtain algorithmic solutions. In software engineering and in automata and formal language theory, reduction is usually one-to-many, from the original problem to a few easier or already-solved problems, obtained by decomposing the original problem. In formal language theory, reductions are used to classify formal languages using closure properties, while in software engineering, reductions are expressed by modularization.

Not surprisingly, the centrality of reduction has also been recognized in areas other than CS. For example, in his classic book on mathematical problem solving, Polya (1957) recommended asking students, after presenting them with a new problem, what other, previously solved problem, the given problem reminds them of.

Based of what has been said so far, we suggest that reduction seems to be one of the habits of mind that should span the entire undergraduate CS curriculum, as indeed occurs in some courses.

However, although reduction is a core CS concept, it is not an easy one to teach. This can be explained mainly, but not solely, by the fact that reduction is a "soft" concept—a concept that cannot be taught through rigorous formalism. In other words, unlike rigid concepts that can be characterized by rigid, formal rules, and like other soft concepts such as abstraction, recursion, encapsulation and a programming paradigm, it is not enough to present a full, comprehensive and concrete definition of reduction, nor is it sufficient to lay out specific rules related to reduction. Furthermore, the difficulty in teaching soft concepts may be rooted in their *generality*—the fact that they can be applied in different domains with respect to different kinds of problems; however, in order to explain them, such concepts also need to be illustrated with *specific* cases.

As hinted above, reduction is strongly connected to another central CS concept—abstraction. Simply put, abstraction is a cognitive means that enables us to concentrate on the essential features of a topic, and ignore details that are not relevant at a specific stage of problem-solving situations. Abstraction is essential in solving complex problems, as it enables the problem solver to think in terms of conceptual ideas, rather than details. Abstraction can be expressed in different ways, all of which guide us to ignore irrelevant details at specific stages in order to overcome complexity. Due to space limitations, we will not be able to review all these aspects of abstraction. Relevant literature is Hoare (1986) and Abelson and Sussman (1986).

Reduction is connected to abstraction in two aspects: First, in order to reduce a problem to another problem, one needs to identify a *connection* between the two; to ignore their differences and identify similarities that allow for reducing one problem to the other. This is in line with the definition of abstraction given by Hershkowitz, Schwarz, and Dreyfus (2001): "An activity of vertically reorganizing previously

constructed mathematics into a new mathematical structure.'' To adjust the definition to the case described in this paper, we suggest defining abstraction as: ''An activity of vertically reorganizing structures into a new structure or structures.'' Since the problems involved in the reduction may at first sight seem totally unrelated, reduction essentially means looking at the given problem through some prism, enabling us to identify structures in it, which are not necessarily clear at first sight, and coincide with or resemble those of another problem. Second, abstraction is expressed by the *implementation* of reduction by using the concept of a black box; that is, ignoring the details of the elements on which the reduction is based.

The connection that exists between the two concepts enables us to define levels of reductive thinking based on the definition of abstraction. We describe the level of reductive thinking as the conceptual gap between the original problem and the problem to which it is being reduced. Specifically, the wider the gap is, the higher the level of abstraction required for reorganizing the given structures into a new one, and the higher the level of reductive thinking; the lower a gap is ranked, the more straightforward the problem solving approach.

The centrality of abstraction in CS problem-solving processes serves later in this paper to explain students' conception of reduction by their tendency to reduce the level of abstraction (Hazzan, 1999, 2003), a behavior reported in various contexts in mathematics and computer science (Hazzan, 1999, 2003). This behavior is probably induced by the difficulty in handling abstractions meaningfully. The mental process of reducing abstraction makes the solution more concrete and thus more mentally accessible to these students.

Reductive thinking, as demonstrated by high school students' solutions to questions dealing with computational models, was discussed in Armoni, Gal-Ezer, and Tirosh (2005). The findings showed that many students preferred direct, non-reductive solutions, even in cases in which reductive solutions could have significantly decreased the complexity of the solution. It was also found that most of the students who constructed reductive solutions chose straightforward reductions, for which a lower level of reductive thinking is required, even if other, less straightforward, reductive solutions could have been more rewarding in terms of design complexity. The study presented in the current paper continues that research. Specifically, in this paper we present results of a study that addresses reductive thinking of undergraduate CS students in various contexts.

3. Research framework

The research was conducted in two phases, as described in the following sections. All questions to which we refer in what follows appear in the Appendix.

3.1. Preliminary Phase

Based on the previously mentioned study of high school students, we were motivated to examine the use of reduction by undergraduate CS students when solving

computational model problems. The population included 63 students taking the course ''Automata and Formal Languages,'' in a top-ranked institution, in which the CS department has very high entrance requirements. This is a mandatory course in the undergraduate CS program, usually taken in the second semester of the 2nd year. It covers theoretical and abstract topics such as finite automata (deterministic and non-deterministic), regular languages, closure properties of regular languages, regular expressions, context-free grammars, context-free languages, canonic forms of context-free grammars, pushdown automata, closure properties of context-free languages, pumping lemmas for regular languages and context-free languages. The course is taught in a standard manner: The students attend a weekly two-hour lecture, and two one-hour tutorial sessions per week (participation is not mandatory). The teaching process is for the most part based on standard textbooks such as Hopcroft and Ullman (1979). The research population consisted of all the students in the course who submitted the last assignment, with the exception of two students who explicitly asked that their assignments not be used for research purposes.

Among the questions in the sixth and last course assignment, were two questions which we used to examine the use of reduction. These questions asked to prove that given languages are regular (Question I) or context-free (Question II). The questions were originally written for high school pupils (Armoni et al., 2005), and thus would not be considered difficult for university students. Also, since the assignment was handed out 2 weeks before the submission date, students had sufficient time to think about the questions and consult with other students, and since it was a pair assignment, most students did not cope with it alone. Fifty-six students submitted the assignment in pairs and seven students submitted individual assignments.

A direct automaton for the language L of Question I is very large, and thus, this question directs the student towards a reductive solution, in which the given language L is decomposed into simpler sublanguages. However, there are many possible reductive solutions. The most obvious one is induced by the phrasing of the question: decomposing L into two sublanguages, corresponding to the two conditions (1 and 2) in the definition of L. Each of these languages can be proved to be regular either by designing a finite automaton accepting it, or by writing a regular expression corresponding to it.

In the case of the first sublanguage, corresponding to the first condition (which we will hereafter denote as L_1), a direct automaton is not trivial, with at least 16 states. A description of eight possible solutions, differing in the way the regularity of L_1 was proven, is given in Armoni and Gal-Ezer (2006). Here we will briefly say that each of these solutions can be associated with a certain level of reductive thinking. In the context of automata and formal languages, the level of reductive thinking is referred to in terms of the gap between the given language (under its given phrasing) and the languages to which it is reduced. In our case, the lowest level of reductive thinking is demonstrated by solutions in which the sublanguage L_1 is not decomposed at all, and a direct finite automaton is designed for it; a higher level of reductive thinking is reflected when the structure of the given language is reorganized into a new structure.

Similarly, Question II also has a number of possible solutions that differ in the level of reductive thinking underlying them, and range from a direct solution to a solution which decomposes L' in a non-trivial manner, yielding a very simple solution, in terms of the automata design complexity it involves. Four reductive solutions to Question II are described in Armoni and Gal-Ezer (2006).

In this phase of the research, the data was analyzed quantitatively. Based on this analysis, it was quite clear that even undergraduate students have some difficulty in fully perceiving the notion of reduction and acknowledging its advantages. In order to further explore this observation, we conducted the second phase of our research, in which we related to additional CS courses.

3.2. The Second Phase

In the second phase of the research, we examined the tendency of undergraduate CS students, at different stages of their studies, to use reductive solutions when solving problems in a variety of CS areas, starting from basic CS1 problems, through algorithmic problems in different problem-solving situations, to computational model problems. In addition, we checked whether students transfer their reductive thinking from the area of algorithmics, in which reduction is usually taught explicitly, to the field of computational models and formal language theory, in which reduction is not necessarily taught explicitly, despite the fact that it is an effective problem-solving heuristic in these areas.

This phase consisted of interviews and questionnaires. Three groups of students were involved in this phase.

- *Group 1:* Eleven freshmen who were asked to solve four CS1 algorithmic problems (Questions A – D).
- *Group 2:* Eight students at the end of their 2nd year of study or in the middle of the 3rd year, who had taken (or were about to finish) a course on algorithms as well as a course on computational models and the theory of formal languages. These students were asked to solve five algorithmic problems on the algorithms course level (Questions 1 – 3 and 7 – 8), three computational model problems (Questions 4 – 6), and the same four CS1 problems given to the first group (Questions A – D).
- *Group 3:* Nine prospective CS high school teachers who were asked to complete a questionnaire that presented the same four questions given to the first group of students (Questions A – D). These students were not interviewed, and only short discussions were held with some of them, to make sure that their solutions were properly understood.

The two questions (I and II) used in the preliminary phase, were also used in the second phase (4 and 6). However, based on the experience obtained during the preliminary phase, these questions were rephrased, excluding solutions based on regular expressions. This rephrasing aimed to decrease the possible effect of interfering factors, not related to reductive thinking.

Interviews with Group 1 and Group 2 were held towards the end of the academic year. These enabled us to gain insight into the factors affecting students' choices of solutions and to establish connections between students' choices and the level of reductive thinking used in the solution process. Accordingly, a qualitative data analysis approach was found to be appropriate in this phase.

4. Findings

Our findings relate to three aspects of how reduction is used: students' tendency to use reduction (section 4.1), students' attitude toward the legitimacy of reduction as a problem-solving strategy (section 4.2), and the actual way in which students apply reduction (section 4.3). This section illustrates that the findings can be interpreted within the framework of the tension between the need to think in terms of high levels of abstraction on one hand, and the fact that reduction introduces a new and different (for the students) approach to problem-solving, on the other. The tension stems from the opposite directions toward which they pull in terms of levels of abstraction: While the former calls for an increase in the level of abstraction, the latter causes students to reduce the level of abstraction (consciously or not) in different ways, in order to turn reduction—an alien (hence an abstract) approach, to a more familiar (hence, more concrete and less abstract) notion (Hazzan, 1999, 2003). The examples we use to demonstrate our findings can serve lecturers who teach these topics to CS students, when relating to reduction.

We chose to present the students' approach to reduction by using the game metaphor, addressing three aspects: players (section 4.1), rules (section 4.2) and the actual playing of the game (section 4.3). The game metaphor was found to be appropriate since, in general, it seems that students view reduction more as an ingredient of the learning ritual that characterizes educational environments, than as a rewarding problem-solving heuristic from whose use they can benefit.

4.1. Who Plays the Game and When: Is reduction used by CS students?

Our data analysis indicates that there is evolving development of reductive thinking. Specifically, 1st-year students (Group 1) rarely used reductive solutions, while the more mature students (Group 2 and Group 3) exhibited a higher level of awareness of the concept of reduction and its potential use in different problem-solving situations.

Indeed, it is reasonable to assume that second-semester students did not use reduction as a problem-solving heuristic simply because they had not yet been exposed to this approach explicitly, although indirect reference to reduction is made during the 1st year of study when concepts such as encapsulation and top-down design (demonstrated by procedures), and library units are taught, and at a later stage, through object-oriented principles.

As mentioned above, a necessary prerequisite for using reduction is the ability to identify relations and connections between different entities (problems, situations, conditions, etc.), an ability which is based, to a great extent, on abstraction.

This ability is probably improved with exposure to reductive solutions based on such relations, and it is thus reasonable that 1st-year students do not demonstrate high levels of this ability.

However, it seems that the more mature students also neglect to use reduction. For example, in Question A, students were asked to design an efficient algorithm that calculates the sum of all integers between 1 and 100 that are indivisible by 3. Many students (not all of whom were 1st-year students) did not recognize connections between this problem and arithmetic series, even when it was clear that knowledge of arithmetic series was available to them since they used it to solve other questions.

At the same time, in many cases, students who were aware of the option to use reduction, did not exhaust its potential. With respect to the above-mentioned question, some did reduce this problem to the problem of calculating the difference between two series—that of all integers between 1 and 100 and that of all numbers between 1 and 100 that are divisible by 3. While they recognized that the first series is an arithmetic series, they failed to recognize that the second one is an arithmetic series as well. A possible explanation is that the phrasing of the second series was not as standard as the phrasing of the first, inducing a wider gap, and thus recognizing it as an arithmetic series required a higher level of reductive thinking and of abstraction.

As it turns out, the ability to identify relations in which it is appropriate to use reduction may depend on the specific problems and the topics to which they refer. For example, all Group 2 students who used reduction tended to use it to solve problems that dealt with shortest paths in graphs (Questions 1 – 3), but none of them used reduction to solve Question 7 which was about list merging. Even though the algorithm designed for Question 7 by most interviewees was essentially the same as the algorithm for constructing a Huffman tree for a given alphabet and its corresponding frequency list, none of the students identified a connection between this problem and the Huffman code problem, not even at the reflection phase, after they had finished designing the algorithm. Only when they were asked directly if they saw any resemblance between Question 7 and the Huffman code problem, did some respond that they did.

In another question (Question 8), students were asked to design an efficient algorithm that constructs a minimal-weighted set of edges that contains at least one edge out of every circle in a given non-directed weighted graph. Most of the students who naturally used reduction for shortest-path problems could not identify the connection between this problem and spanning trees. It is reasonable to assume that while they were exposed to the idea of reductive solutions in the context of shortest paths in graphs, they were not exposed to such a use in the other contexts, and did not transfer the reductive strategy from one context to another.

The influence of the topics to which the problems relate on students' ability to identify relations and on their tendency to use reduction in solving these problems is closely related to the concept of *transfer* (Noss & Hoyles, 1996; Nunes, Schliemann, & Carraher, 1993). Clearly, in the context of reductive thinking, transfer may be relevant between sub-topics of a common topic (e.g. various classes of algorithmic problems) and also between different topics (such as algorithms and formal language

theory). Reduction is taught explicitly in the algorithms course, usually in tutorial sessions, where reductive solutions are often demonstrated. At the same time, reduction is not necessarily mentioned in the course on automata and formal languages, although it is a powerful tool for solving problems related to formal language theory. Exposure to reduction and its emphasis as a problem solving strategy in the algorithms course could have resulted in the development of a general tendency to use reduction, a tendency that could be applied in other contexts as well.

Our findings, however, clearly indicate that there was no transfer of the tendency to use reduction from the area of algorithmic problems to the area of computational models: **All** Group 2 students, even those who used reduction as a primary strategy for solving algorithmic problems, demonstrated a low level of reductive thinking when solving questions dealing with computational models—they either gave direct solutions, or, in some cases, used a standard decomposition directly induced by the phrasing of the question. This strengthens our first impression, based on the preliminary phase, that students do not significantly tend to use reduction when solving computational model problems.

Within the general framework we demonstrate in this paper, this section illustrates that the tension between the levels of abstraction that students exhibit is influenced by the topic of the problem.

4.2. Rules of the Game: The legitimacy of reduction

In several cases and situations, students expressed the notion that the use of reduction was not a legitimate option. This feeling of illegitimacy was found to depend on several factors:

- *Course framework*. Some students considered reduction legitimate only if it is contained in the scope of the same course. For example, while solving Question C (finding a second maximum), a Group 2 student realized that this problem is a specific case of the ''select'' problem, in which the i-th element of a given list should be returned. He remembered that he had been taught a linear-time algorithm for solving this general problem, but did not allow himself to use it since he did not remember the details (this point will be discussed below). He further said that since it was a very complicated algorithm ''no one remembered it and it was immediately turned into a black box [used to solve other questions in the same course].'' When asked why he did not use it as a black box during the interview, he answered: ''We are not now doing homework in the 'Data Structures' course. The idea is that in courses you have the context of what you have learned, which you can use, and then it disappears.''
- *Question framework*. Several 1st-year students felt uncomfortable using reduction that span between problems presented in different questions. They felt that problems could not be connected if they appeared in different questions. For example, a Group 1 student, solving Question C (the second-max problem), said

''This [Question C] is separate from [Question] B . . . , that is, it is not related to [Question] B. . . . It is a new question.'' After being ensured by the interviewer that he could use his solution to Question B (finding maximum), he ended up with a solution that changed the code he gave for solving Question B (adapting it to Question C in an incorrect manner, a pattern to which we relate below in the context of constructive reductions). Another Group 1 student, solving Question C, said repeatedly, as if not believing: ''Can I rely on what we have here [Question B]?,'' ''Should I solve it from scratch, or can I rely on what we have here?'' After giving a correct reductive solution for Question C, he added, ''And I can also solve it from scratch, of course,'' as if trying to ensure the interviewer he is capable of giving a complete solution.

- *Interview setting*. The same Group 2 student cited above, when relating to the course framework, also referred to the situation in which the question is solved: ''In an interview, say a job interview, if someone asks you something, you cannot say 'I was once shown an algorithm that does something,' but you should explain how it works, if not more than that.''
- *Black-box effect*. Some students said that a black-box reduction solution is ''cheating'' or ''silly,'' hence it is not a legitimate approach. They felt uncomfortable with hiding part of the solution inside a black box. This feeling also seemed to depend on some subjective factors. For some, it was not legitimate to use reduction to **simple** problems (such as maximum or sorting), ignoring the details of their solution, whereas it was legitimate to use reduction to more complex problems, for which it seemed reasonable not to remember all the solution details. For example, a Group 1 student tried to solve Question D (finding the most frequent element) using reduction to an AVL tree (we will return to this later, discussing reduction to solutions rather than to problems). When the interviewer said, ''you could have just used sorting,'' he said ''Yes, but that is trivial.''

  In contrast, other students felt that it was not legitimate to use reduction to previously solved **complex** problems, if they did not remember exactly how these complex problems were solved. Such use of reduction was referred to as ''cheating,'' since the main part of the solution was hidden inside a black box.

The different factors that influence students' beliefs about the legitimacy of using reduction can be explained by the fact that the students do not conceive of it as a rewarding problem-solving heuristic whose use, in fact, reflects high problem-solving skills. In the case of Group 1 students, this might stem from not being sufficiently and explicitly exposed to reduction. However, Group 2 students, who had been exposed explicitly to reduction, apparently still view it as legitimate only in the relatively narrow context in which it was taught.

Within the general framework we demonstrate in this paper, this section illustrates that the tension between the levels of abstraction that students exhibit is influenced by the way they conceive the legitimacy of reduction as a problem-solving heuristic.

4.3. Playing the Game: Applying reduction

Findings of the preliminary research stage showed that for Question I, about a third of the students gave solutions which can be graded as high in terms of level of reductive thinking. Almost a quarter of the students chose solutions which can be graded as exhibiting no, or a very low level of, reductive thinking. About 30% of the students gave a solution based on regular expressions whose level of reductive thinking can be graded as relatively low. For Question II, a majority of the students (60%) chose a direct solution, ranked lowest in terms of reductive thinking level, and 20% chose the solution ranked highest in terms of reductive thinking level. These findings are detailed in Armoni and Gal-Ezer (2006).

The second phase of the research helped us to deepen our understanding with respect to these phenomena. Based on analysis of data gathered in the second phase, we now present six phenomena that show how students apply reduction in different problem-solving situations.

Preferring constructive reduction over existential reduction. For computational model problems, two kinds of reduction can be used: constructive reduction and existential reduction. In both kinds, the given language is decomposed into a few simpler languages, and automata are designed for each of these languages. In constructive reduction, known construction algorithms are then invoked to obtain an automaton for the given language. In existential reduction, all that is needed at that stage is to cite the appropriate closure properties, thus showing the existence of an automaton for the given language instead of actually constructing such an automaton.

Our findings show that, in general, solutions based on constructive reduction are favored over solutions based on existential reduction since the constructive solutions are perceived by the students to be more complete, although they are aware of the existential solution and its validity.

The following example demonstrates the tendency to constructive reduction. The interviewee was an excellent student at the end of the fourth semester of the very challenging track of a double major in mathematics and computer science. He demonstrated a significant tendency to use reduction in solving algorithmic questions, and it was actually his primary strategy.

When solving Question 4, he decomposed the given language into five base languages, one corresponding to the second condition, and four languages over the alphabet $\{a, b, c\}$ for the first condition: $\{w|\#_a(w) + \#_b(w) = 0\}$, $\{w|\#_a(w) = \#_b(w) = 1\}$, $\{w|\#_a(w) = \#_b(w) = 2\}$ and $\{w|\#_a(w) = \#_b(w) = 3\}$, where the notation $\#_\sigma(w)$ denotes the number of appearances of the letter σ in the word w.

He used constructive reduction, saying that he could combine these automata, designed for the base languages, using the construction algorithm which yields a non-deterministic automaton, accepting the union language. The same strategy was employed in solving Question 5, for which the student decomposed the given language into two base languages. After designing the corresponding automata he was about to use the construction algorithm which yields a non-deterministic automaton

again, accepting the union language. But then he said, ''Actually, I don't even have to design this automaton. That is, I can say this is an automaton and this is an automaton, each accepting that language and this is the union of these two languages, and we have closure.''

This shows that in terms of Bloom's taxonomy (Bloom, Mesia, & Krathwohl, 1964), this student knew existential reduction and viewed it as a legitimate strategy. Still, the reflection phase, conducted after this student completed his solutions for Questions 4 – 6, seemed to indicate flaws in his perception of existential reduction.

Referring to his solution for Question 5, the student was asked why he chose to decompose the given language into two base languages, using the union operator, and whether he could think of an alternative decomposition. He replied that he could not think of any other way, and that what motivated his choice was the ''or'' connective, connecting two independent conditions. When confronted with the option of using the ''and'' connective, resulting in a finer decomposition, he could not see how this could be done, since the ''and'' connective requires two conditions to hold at once (in his words: ''I must take care of both''). The conversation continued as follows (**I** = interviewer, **S** = student):

I: OK. But perhaps you could have still coped with each condition by itself.
S: With this [the form condition] by itself, and this [the other two conditions] by itself? No, because I can read the [input] word only once and not twice. That is, if it is done separately, I would have read the word twice. Once we see if this [the form condition] holds, and...
I: Right, but when you solved it, you told me that you could have connected these two automata, but you also said ''Actually, I don't have to, I can actually say this [the first base language] is regular, this [the second base language] is regular.''
S: Here I can use... De-Morgan and such, and... but, but...
I: There you could have coped with each part separately.
S: Yes, OK, right.
I: And say—this [the first base language] is regular, this [the second base language] is regular, this [the given language] is regular.
S: I agree. I don't think I could have done it with closure under intersection.
I: The property of closure under intersection also induces a construction algorithm, if you really insist.
S: Right. I think using De-Morgan rules is the correct way, since I didn't think I could design an intersection automaton.
I: A Cartesian-product automaton.
S: Yes, but it is quite a headache to do that. A Cartesian-product automaton is good for formal proofs, not for the design of an automaton we really need.
I: On the other hand, here you only need to prove that it [this language] is regular.
S: Yes.
I: You don't have to design an automaton.
S: Right, right, it really could have been...
I: Better? Would it have been of any use? Would you have chosen this decomposition of three languages?
S: No, it would have split... it requires first to read *a* and *b*, but it doesn't complicate things that much, so I don't think it would have mattered that much. In any case, I should... Yes, OK, it would have simplified the automaton a bit.

This student knows that the regular languages are closed under union, and also under intersection. He knows that in order to prove the regularity of a given language he can use closure properties, ending up with an existential proof rather than a constructive one. He also knows that if needed, it is possible to use a construction algorithm which outputs an automaton accepting an intersection language (a Cartesian-product automaton), but that this algorithm is quite tedious, resulting in a potentially large automaton. Using Schoenfeld's terminology (1985), we can say that all these facts belong to his resources, and that, in fact, he knows all he needs to come up with an existential regularity proof, using an ''and'' connective. Still, he does not feel it to be possible.

A possible explanation is that even if this student understands existential proofs on some level, in order to perform an existential regularity proof using some base languages, he needs to realize, to feel the details of the construction algorithm underlying this existential proof. This is relatively easy for him, when the proof is based on the union operation, since he easily remembers how to connect the two base automata, resulting in a non-deterministic union automaton. However, he does not think of such an algorithm for the intersection algorithm (dismissing the Cartesian-product construction algorithm on the grounds of being impractical), and thus cannot produce existential reasoning for the intersection operation.

As to algorithmic problems, students may perform constructive reductions as well. In such cases, the solution to the reduced-to problem is not viewed as a black box, but rather its details are used to obtain a solution to the original problem. For example, four Group 1 students and three Group 2 students gave constructive reductive solutions to Question C (the second-max problem), in which they changed the code they wrote for Question B (the maximum problem), and adapted it (in many cases, incorrectly) for Question C.

A Group 1 student, solving Question A (in which one has to calculate the sum of all numbers between 1 and 100, which are indivisible by 3), reconstructed the formula for calculating the sum of an arithmetic series. When asked about it, he said that he identified the series as an arithmetic series (whereas other students who also developed this formula, did not identify it), but since he didn't remember the exact formula he felt he had to reconstruct it. He did not choose to ''close'' it, even temporarily, in a black box titled ''sum of arithmetic series,'' use it for the new solution, and only then cope with the contents of the black box; rather, he designed a direct solution, part of which calculates the sum, thus working on a lower level of abstraction.

Here is another example of constructive reduction for solving algorithmic problems, an example which also demonstrates levels of reduction in the context of algorithmic problems: In the course on algorithms, students learned that a maximum matching problem can be solved using reduction to a maximum flow problem. One of the homework assignments included problems that could be solved using reduction to maximum matching. The teaching assistant reported that many students reduced the given problems to maximum flow problems, thus working more concretely, by going deeper into the inner black box, and ending up with a more ''complete'' algorithm.

We reexamined this phenomenon when analyzing students' solutions to a question given on a final exam in the ''Algorithms'' course at the Open University of Israel. This question could also be solved using reduction to maximum matching. However, of the 22 students who solved the problem, two gave totally incorrect solutions (using neither maximum matching nor maximum flow) while all the rest reduced the problem to a maximum flow problem. Twelve of them stated that this was actually a maximum matching problem, but still chose to go deeper into the inner black box, reducing the given problem all the way to a maximum flow problem.

Reduction only when the details are known. Very similar to the phenomenon of the tendency to constructive reductions is the phenomenon of applying reduction only when the details are known. In some cases, students used reduction to a problem only after they were sure that they fully remembered the algorithm that solves that problem. For example, students recalled the details of BFS which solves the shortest paths problem in an undirected graph, before they reduced Question 1 to that problem. The reduction was then formulated correctly as a black-box reduction to a problem. However, these students could not establish the connection between the two problems before they made sure they knew how the reduced-to problem is solved.

Reduction to a solution (rather than to a problem). This phenomenon addresses situations in which students explicitly say that they reduced the problem they were solving to a *specific solution* of another problem. One of the students in Group 2 said explicitly: ''I am trying to think of relevant *algorithms* that I already know that can do something similar,'' rather than thinking of similar *problems*, as suggested by Polya (1957).

Here are several examples taken from students' solutions, in several contexts, demonstrating several levels of this phenomenon:

- *Shortest paths:* Reducing the problem of finding a shortest s-t path in a {1, 2}-weighted undirected graph to *BFS* (Question 1), rather than to the problem of finding a shortest path in a non-weighted graph.
- *Sorting:*
  - **Example 1**: Reducing the problem of finding the most frequent element in a given list of numbers (Question 4) to *quicksort* (or mergesort, bubblesort, heapsort, or tree-sort in the case of other students), rather than to sorting.
  - **Example 2**: Reducing the problem of finding the most frequent element in a given list of numbers (Question 4) to an AVL-tree. In this case, the level of abstraction exhibited may be considered even lower than that of the previous examples, since this student didn't stop the reduction at the algorithm (balanced-tree-sort) level, but, rather, reduced the problem to a specific data structure. In fact, this student was looking for a data structure that enables extracting a minimal element in O(log n) time for obtaining a sorted list in O(nlogn) time, which could obviously be achieved by reduction to sorting, had he freed himself of the implementation details.

- *Computational models:* A Group 2 student solved Question 5 using decomposition into two base languages. However, she described her strategy using the following words: ''We can again use the non-determinism property.'' That is, she did not consider the construction algorithm that yields a union automaton as a black box, but rather related to its exact implementation, as an algorithm that adds an initial state and connects the two base automata using non-deterministic transitions. For her, the essence of that algorithm was not the problem it was solving, but rather its implementation.

These examples demonstrate a tendency to identify a problem with its solution. Such a tendency may interfere with the ability to hide a solution inside a black box, thus working on a lower level of abstraction. This interference may inhibit students from reaching simpler solutions.

Not taking full advantage of the power of reduction when it is observed. Some students who used reduction for some problems did not exploit it in other cases, although they were already ''half-way there.'' For example, when asked to design an algorithm that finds the most frequent element in a given list of numbers, one Group 1 student said that it was difficult because the list was not sorted, but he did not follow this line of thought and use reduction to sorting. In a few cases, when solving computational model problems, students identified a certain decomposition of a given language to sublanguages, but their solution eventually used either a less refined decomposition, or used no decomposition at all. For example, when solving Question 5, one of the students said ''So we divide it in two. First, the sum of a appearances is even, and second, the number of c appearances is equal to or greater than 3. Let's start with the easier one.'' Still, he did not follow this decomposition, but rather gave **one** direct automaton. Another student solving the same question identified three conditions and then said, ''my conditions are this [the word has the form $a^n b^m c^k$, n, m, $k \geq 1$] with this [the number of a appearances is even], and this [the word has the form $a^n b^m c^k$, n, m, $k \geq 1$] with this [the number of c appearances is equal to or greater than 3].'' He then designed two automata, for each of the two combined conditions, instead of three simpler automata, for each of the three atomic conditions.

Considering reductive solutions as inefficient. In some cases, students considered a reductive solution but applied a direct solution or a solution based on a lower level of reductive thinking due to efficiency considerations. Students sometimes felt that direct solutions were more efficient since they were tailored to the problem at hand, even when they knew that the two solutions—the direct and the reductive—had the same time complexity (and sometimes even the same actual time cost). For example, one of the students in Group 1 considered a reductive solution for Question C (the second-max problem), in which the problem was reduced to the problem of Question B, of finding the maximum element (in the original list, and then in the list obtained after deleting the maximum element). Then he claimed this solution to be inefficient

compared to the direct solution, in which two variables are used in one traversal of the list. He realized that the two solutions shared the same time complexity, but was bothered by the existence of two loops in the reductive solution versus the one (more complex) loop in the direct solution.

Similarly, a Group 2 student solved Question C by using the same reduction to Question B and then said, ''That's the big O trick, you can do it in a very inefficient way, because it can be done with just one pass, I'm sure.''

Another Group 2 student gave an incorrect direct solution to the second-max problem, during which he also considered a reductive solution, using reduction to a sort problem, but dismissed it, due to its relatively costly complexity. After completing the direct solution he was asked if he could use the same method of relying on an already-solved problem to come up with an efficient algorithm for the second-max problem. He answered, ''I could have used MAX. I did consider it, but my intuition not to use it was based on complexity reasons.''

Underrating the difficulties of non-reductive strategies. In several cases, students tended to underrate the difficulties they encountered when trying to solve a problem directly or by using low-level reduction. After encountering significant difficulties during the solution process, and confronted with the possibility of a reductive strategy which may have induced an easier solution, some students tended to state: ''It was not as complicated,'' ''I could have done it my way if I had worked on it a little longer,'' ''I know how to do it,'' or ''It would not have made much of a difference.'' Such statements may reflect the conflict students face between not acknowledging reduction as a rewarding problem-solving technique on one hand, and the difficulties in handling details when reduction is not applied, on the other.

The first three phenomena presented in subsection 4.3 can be related to students' difficulties regarding the black box concept. We saw that students sometimes rely on the specific solution to the reduced-to problem, a solution that is supposed to be hidden inside a black box, whether by actually using its details to construct a solution to the original problem, or by at least remembering its details when performing the reduction, or by identifying the reduced-to problem with one of its possible solutions. It is as if these students feel the need to know what is inside the black box, and relate to it directly. This demonstrates again the clash between the tendency to reduce the level of abstraction, making the solution more concrete, and the need to work on a relatively high level of abstraction, while performing reduction, due to its black box characteristic.

In general, in subsection 4.3 the tension between the levels of abstraction that students exhibit is influenced by the way they estimate the contribution of abstraction to their problem solving skills.

5. Discussions and Summary

In the above we showed how students conceive of and apply reduction. Specifically, we showed that the above-described phenomena can be interpreted

within the framework of the tension existing between thinking in terms of different abstraction levels. We suggest that the tension between the levels of abstraction that students exhibit is influenced by the topic with which the problem deals, the way they conceive the legitimacy of reduction as a problem-solving heuristic, and the extent to which they estimate that abstraction contributes to their problem solving skills.

Specifically, as previously stated, when we examine the set of all findings, we can identify two factors that may explain them: First, the need to think in terms of high levels of abstraction; and second, the new approach that reduction introduces into problem-solving situations. These two factors, in fact, clash. On one hand, the need to think in terms of high levels of abstraction requires students to marshal their abstraction skills; on the other hand, the fact that reduction introduces a new approach to problem-solving, with which the students are not familiar from their previous studies, leads them (unconsciously in most cases) to reduce the level of abstraction (Hazzan, 1999, 2003). The combination of these two factors causes a unique situation in which even those students capable of demonstrating high levels of abstract thinking, may refrain from it.

Our observations may shed some light on students' problem-solving performance in different CS courses. In other words, we may find students who do well in a particular CS course but cannot apply their skills outside the narrow confines of the problems presented in that course. Specifically, while specific course material may be grasped and well-performed by students, if they fail to gain general habits of mind needed for solving problems in other topics within CS that require the same habits of mind, their performance may be less successful.

Accordingly, our aim in educating our students, beyond teaching the actual CS material itself, should also include the ability to employ various habits of mind that characterize one area of CS in more general contexts and, when required, to synthesize several of these ways of thinking. Since CS is about transfer in the sense of building abstractions, it seems reasonable to look for ways to help students develop modes of thought that are essential for doing CS. As far as we know, these skills, which computer scientists take for granted, are rarely discussed explicitly in CS courses. In other words, although each course in the CS curriculum has a significant role, we suggest that, when appropriate, in addition to the course material, habits of mind be highlighted as well. This perspective well complements the idea that CS is, in fact, a unique problem-solving field.

Based on the above findings, and on our view of reduction as an important problem-solving heuristic, we now present several teaching applications that may promote reductive thinking.

A. Introduce the idea of reductive thinking as early as possible in CS1: This recommendation is not limited, of course, only to reduction, but also to other heuristics and soft ideas such as abstraction, as well as to concepts such as efficiency (Ginat, 1996, 2001).

B. Demonstrate reduction in different situations and in various contexts: Instructors can relate explicitly to the reductive nature of solutions presented and to the advantages of these solutions over direct or lower-level reductive solutions. In this context, the didactic strategy for a course on computational models described in Armoni and Gal-Ezer (2005) may be useful in other courses as well. We plan to adapt this strategy to other CS courses and to verify its effectiveness in terms of developing reductive thinking.
C. Control the use of reduction: As suggested by Schoenfeld (1985), the control mechanism is an important ability in problem-solving situations. Withrespect to reduction, we should educate our students to be aware of how they actually use reduction. For example, on what assumptions do they base their reductive solution, do they properly validate the correctness of the reduction, etc.

Finally, our findings may motivate similar research among R&D people and programmers in industry. It would be interesting to find out how these computing professionals use reduction (if at all), their attitude towards the legitimacy of using reduction, and their ability to correctly use black boxes. These questions are of general interest, but also relevant in the context of their formal education, whether in software engineering, computer science or computer engineering.

Acknowledgement

We would like to thank Yechiel Kimchi, Yuval Ishai and Dudu Amzallag from the Department of Computer Science of the Technion for their cooperation in the different stages of this research. This research was supported by the E. and J. Bishop Research Fund.

References

Abelson, H., & Sussman, G.J. (1986). *Structure and interpretation of computer programs.* Cambridge, MA: MIT Press and McGraw-Hill.

Armoni, M., & Gal-Ezer, J. (2005). Teaching reductive thinking. *Mathematics and Computer Education*, *39*(2), 131–142.

Armoni, M., & Gal-Ezer, J. (2006). Reduction—An abstract thinking pattern: The case of the computational models course. In *Proceedings of the 37th ACM SIGCSE Technical Symposium on Computer Science Education* (pp. 389–393). New York, NY: ACM.

Armoni, M., Gal-Ezer, J., & Tirosh, D. (2005). Solving problems reductively. *Journal of Educational Computing Research*, *32*(2), 113–129.

Bloom, B.S., Mesia, B.B., & Krathwohl, D.R. (1964). *Taxonomy of educational objectives* (Vol. 1, *The affective domain*; Vol. 2, *The cognitive domain*). New York: David McKay.

Cuoco, A., Goldenberg, E.P., & Mark, J. (1997). Habits of mind: An organizing principle for mathematics curriculum. *Journal of Mathematical Behavior*, *15*(4), 375–402.

Ginat, D. (1996). Efficiency of algorithms for programming beginners. In *Proceedings of the 27th ACM SIGCSE Technical Symposium on Computer Science Education* (pp. 256–260). New York, NY: ACM.

Ginat, D. (2001). Early algorithm efficiency with design patterns. *Computer Science Education*, *11*(2), 89 – 109.
Hazzan, O. (1999). Reducing abstraction level when learning abstract algebra concepts. *Educational Studies in Mathematics*, *40*(1), 71 – 90.
Hazzan, O. (2003). How students attempt to reduce abstraction in the learning of mathematics and in the learning of computer science. *Computer Science Education*, *13*(2), 95 – 122.
Hershkowitz, R., Schwarz, B. B., & Dreyfus, T. (2001). Abstraction in context: Epistemic actions. *Journal of Research in Mathematics Education*, *32*, 195 – 222.
Hoare, C.A.R. (1986). Mathematics of programming. *Byte*, *10*(8), 115 – 149.
Hopcroft, J.E., & Ullman, J.D. (1979). *Introduction to automata theory, languages and computations.* Reading, MA: Addison-Wesley.
Miller, G.A. (1956). The magical number seven plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, *63*, 81 – 97.
Noss, R., & Hoyles, C. (1996). *Windows on mathematical meanings.* Dordrecht, The Netherlands: Kluwer.
Nunes, T., Schliemann, A.D., & Carraher, D.W. (1993). *Street mathematics and school mathematics.* Cambridge, UK: Cambridge University Press.
Polya, G. (1957). *How to solve it* (2nd ed.). Princeton, NJ: Princeton University Press.
Schoenfeld, A.H. (1985). *Mathematical problem solving.* Orlando, FL: Academic Press.

Appendix: Questions used in the Research

Questions Used in the Preliminary Phase

Question I

Let L be the language over the alphabet $\{a, b, c\}$ which contains exactly all the words for which at least one of the following conditions holds:

1. The number of a's is equal to the number of b's, and the sum of a's and b's is bounded by 6.
2. The word includes the pattern abc and ends with the pattern bb.

Is L regular? Prove your claim.

Question II

Prove that $L' = \{a^n b^m c^k \mid n, k \geq 0, m = n + r, r \neq k, r \geq 0\}$ over the alphabet {a, b, c} is context free.

Questions Used in the Second Phase

Question A

Show how to efficiently compute the sum of all numbers between 1 and 100 that are indivisible by 3.

Question B

Design an algorithm that gets as input a list of distinct numbers, and outputs a number z such that for every x in the list $z > x$.

Question C
Design an algorithm that gets as input a list of distinct numbers, and outputs a number y such that there exists one number z which satisfies: $z > y$ and for every other x in the list $y > x$.

Question D
Design an algorithm that gets as input a list of integer numbers, and outputs a number that appears in that list a maximum number of times.

Question 1
Design an algorithm, as efficient as you can, which gets as input an undirected graph $G = (V, E)$ with a weight-function $w{:}E \rightarrow \{1, 2\}$, and two nodes $s, t \in V$, and finds the weight of a shortest path between s and t in G.

Question 2
Design an algorithm, as efficient as you can, which gets as input a connected undirected graph $G = (V, E)$, in which every edge is colored with A, B or C, and also gets two nodes $s, t \in V$, and finds a path with a minimal number of A-colored edges, among all the paths between s and t in G that start with a B-colored edge and end with a C-colored edge.

Question 3
Design an algorithm, as efficient as you can, which gets as input an undirected graph $G = (V, E)$ and an edge e in that graph, and either finds a simple odd-lengthed cycle in G that passes through e, or decides that no such cycle exists.

Solve the following two questions without using regular expressions:

Question 4
Let L be the language over the alphabet $\{a, b, c\}$ which contains exactly all the words for which at least one of the following conditions holds:

1. The number of a's is equal to the number of b's, and the sum of a's and b's is bounded by 6.
2. The word includes the pattern abc and ends with the pattern bb.

Is L regular? Prove your claim.

Question 5
Let $L = \{w = a^n b^m c^k | (n,m,k > 0)$ and $((\#_a(w)$ is even) or $(\#_c(w) \geq 3))\}$ over the alphabet $\{a, b, c\}$, where $\#_\sigma(w)$ stands for the number of appearances of the letter σ in the word w. Prove that L is regular.

Question 6
Prove that the language $L = \{a^n b^m c^k | n, k \geq 0, m = n + r, r \neq k, r \geq 0\}$ over the alphabet $\{a, b, c\}$ is context free.

Question 7

Assume we are given k lists of numbers $-L_1, \ldots, L_k$. Each of these lists is sorted and our goal is to merge them into one sorted list. For that purpose we get an algorithm A, that merges two sorted lists into one sorted list, and we can use it as a black box. The time cost of applying this algorithm on two lists is exactly the sum of the lengths of these two lists.

Design an algorithm, as efficient as you can, which gets the k lists as input and determines on what inputs should algorithm A be applied on each phase, such that the total merge cost is minimal.

For example, for three lists L_1, L_2, L_3 one can first merge L_1 and L_2, and then merge the resulting list with L_3. In that case the total time cost of the merging process is $2 \cdot |L_1| + 2 \cdot |L_2| + |L_3|$. But, one can also merge L_1 and L_3, and then merge the resulting list with L_2, and in that case the total time cost of the merging process is $2 \cdot |L_1| + 2 \cdot |L_3| + |L_2|$. Your algorithm should choose the way which yields a minimal total time cost.

Question 8

Design an algorithm, as efficient as you can, that given an undirected graph $G = (V, E)$ with a weight-function $w: E \rightarrow R^+$, finds a subset $E' \subseteq E$ such that E' contains at least one edge out of every cycle in G, and the total weight of E' is minimal.