Computer Science Education Research in Israel

Michal Armoni and Judith Gal-Ezer

1 Introduction

This chapter is devoted to research on computer science (CS) education conducted in Israel by Israeli CS educators and researchers. CS as an independent discipline has been offered in Israeli universities since the 1960s. The first department of CS in Israel was established in 1969 in the Technion, and the first program was a graduate program. Very soon thereafter, in the middle of the 1970s, computers were introduced into the K-12 arena, mainly in two tiers. The first was intended to teach CS as an independent subject and the second was intended to exploit the potential of computers to teach and learn other subjects, mainly scientific ones, such as mathematics and physics (e.g., Refs. [36, 50]). Later, when computers became more prevalent and the government came up with the goal and the slogan of "a computer for every child", which they were eager to achieve, a third tier was added, independent of CS: using computer applications (e.g., for writing documents and preparing presentations).

Gradually, CS departments, schools, or faculties were established in all the universities and most of the colleges. Later on, most of them also started to offer programs towards obtaining a CS teaching certificate. Obviously, over the years, they have occasionally updated their CS programs. Similarly, the K-12 CS subject has been updated a few times, but the first major change took place in the 1990s, when the high-school CS curriculum was in fact developed again, from beginning to end, resulting in a coherent curriculum, based on solid well-rationalized educational

M. Armoni (✉)
Department of Science Teaching, Weizmann Institute of Science, Rehovot, Israel
e-mail: michal.armoni@weizmann.ac.il

J. Gal-Ezer
Mathematics and Computer Science Department, The Open University of Israel, Ra'anana, Israel
e-mail: galezer@cs.openu.ac.il

M. Apiola et al. (eds.), *Past, Present and Future of Computing Education Research*,
https://doi.org/10.1007/978-3-031-25336-2_18

principles; it emphasized the fundamental and scientific concepts of the discipline, rather than just programming or technology [47, 51].

CS educational research existed even before well-established curricula were published and implemented, and just like the CS programs, it has evolved and flourished, dealing with all educational levels, from young children to advanced undergraduate and even graduate students.

Research on K-12 CS education occupies a large portion of computing education research in Israel. It is independent of the educational system, but at the same time, it has rich and fruitful connections with the educational system. For example, as part of the development and implementation of the new curriculum in the 1990s, educational research teams in the universities were funded by the Ministry of Education to develop different parts of the curriculum, including matching textbooks. These teams (including university researchers as well as teachers) performed research-based design and investigated the gradual implementation of the curriculum in schools, in the classrooms of teachers who volunteered to pilot the new courses. New research directions have often emerged from these research-based design projects. In other cases, new research directions were initiated by the researchers, and their outcomes have influenced decisions taken by the educational system. Moreover, conducting research in schools requires permission from the Ministry of Education, and this is usually granted (provided it meets required standards and regulations). In addition, to prepare teachers for the new courses, the development teams at the universities operated professional development courses for in-service CS teachers, and universities also offered programs for prospective CS teachers, towards obtaining a teaching certificate, thus opening up new directions of research focusing on teachers.

Considering everything, in Israel there are four elements that interact, creating a synergy that promotes K-12 CS education [86]: a well-established *curriculum*, accompanied by *research* conducted at the universities, a mandatory formal *teaching license* provided by the Ministry of Education, and *teacher training* programs towards obtaining a teaching certificate offered by the universities.

Currently, CS education is a very active research area, with researchers who reside in schools of education, science teaching departments or CS departments, guiding students towards master's or doctoral degrees in CS education.

Numerous papers on CS education research have been published since the 1970s, dealing with teaching and learning of various CS areas. Some focus on students – K-12, undergraduate, or even graduate students – and other focus on K-12 teachers, in-service as well as prospective teachers. They deal with many aspects of teaching and learning, such as instructional approaches, visualization, assessment, gender, and affective factors, to name but a few. Obviously, it is impossible to mention here all or even most of these publications, or even touch on all the topics and aspects with which they deal.

Instead, we decided to tell the tale of Israeli computer science education research through the perspective of the discipline itself. Like other academic disciplines, CS is underlain by fundamental ideas and concepts [120] that define its nature; it is a discipline that is more than programming and is concerned with computational problems – understanding, analyzing, and solving them. These ideas and concepts cut across the discipline, recurring in different fields and contexts, from basic to advanced ones. They can and should be taught at various levels, to K-12 and university students, revisited throughout the curriculum, because only a spiral teaching of them [37] can help students perceive them and help them understand the nature of our discipline. We chose to organize this chapter around such ideas, which were often addressed by CS education research conducted in Israel.

To contextualize the research work reviewed in this chapter, we will start by elaborating more on curricular issues, in Sect. 2. Section 3 will review research conducted by Israeli researchers that deal with fundamental concepts and ideas of CS. We will conclude and look forward to the future in Sect. 4.

2 Curricular Issues

As noted above, the history of CS as an independent discipline in Israeli universities began in the 1960s, starting with a graduate program offered by the Technion; then it was followed by other universities, and later by colleges. Today, universities in Israel offer undergraduate CS programs and graduate CS programs towards master's and Ph.D. degrees. The first undergraduate programs were designed on the basis of the first published ACM curriculum recommendations in 1968 [1] and have been updated over the years as the discipline and its education have evolved. In addition, as the computing field continued to evolve, the recommendations have been updated to include new computing disciplines, reflecting the new reports for the new disciplines written by the ACM and IEEE task forces. All the programs offered today by Israeli universities and colleges are regulated by the Israeli Council for Higher Education.

Research on CS education at higher education institutes has also evolved. Some of the educational efforts were accompanied by educational research or took advantage of the educational research results and many studies examined teaching and learning processes taking place during undergraduate CS programs (e.g., Refs. [2, 19, 105, 108, 116]).

In the middle of the 1970s, CS was introduced into high schools in Israel. The K-12 system in Israel is centralized; it is under the responsibility and control of the Ministry of Education, namely, all curricula for all subjects and for all age levels are determined by the Ministry of Education, and every school must adhere to them. The subject of CS was first taught in technological schools that offered practical studies. Nevertheless, these programs also included theoretical subjects (e.g., finite automata). These programs were updated to include Spreadsheet and the LOGO programming language, and formed an intermediate curriculum. The content of the curriculum was determined by a committee appointed by the Minister of Education, as is customary in the Israeli educational system. The committee was advised by the supervisor on CS education in the Ministry of Education. The implementation of this intermediate program was accompanied by research, praising the features of

LOGO on the one hand, but also challenging the prevailing opinions, for example, that meaningful learning through LOGO can take place without the assistance of teachers or instructors (e.g., [95]).

At the beginning of the 1990s, the Minister of Education appointed a new committee, composed of CS researchers and educators, officers from the Ministry of Education, and CS high-school teachers. This committee was assigned the task of designing a new high-school CS curriculum. Its goal was not to train the students to become CS professionals. Rather, the goal was to introduce the students to the discipline, so later they would be able to make an informed choice regarding their path in higher education. This curriculum [51] opened a new area of research in Israel, significantly expanding the Israeli CS education research community. Although this curriculum has been updated over the years after its initial implementation, the main principles that guided the designers still stand today.

A key criterion for defining the core subjects to be included in the curriculum was longevity. As we know even better today, the technology changes rapidly, far more than the basic ideas of CS, which have lasting and fundamental value. As a result of this criterion, the program covered CS knowledge and skills that are independent of specific computers or programming languages. The designers of the curriculum followed nine underlying principles, the most important and relevant world-wide are as follows:

- CS is a full-fledged scientific subject, the same as other sciences;
- The program should focus on the key concepts and foundations of the field;
- The program should include mandatory and elective courses (from which the teachers can choose their preferable courses);
- Conceptual and experimental issues should be interwoven – The Zipper Principle;
- Two quite different problem-solving paradigms should be taught;
- New course material must be developed for both students and teachers;
- Teachers certified to teach the subject must have adequate formal CS education.

Most of these principles apply to every subject and in particular, every scientific subject, but at that time, this was not clear regarding CS, and it is still not common knowledge in all countries even today. Indeed, the program introduced both facets of the discipline – the theoretical and the practical – and as the zipper principle states, both were interwoven throughout the program. The teaching of different problem-solving paradigms, namely, different approaches to problem solving, is implemented by using programming languages of different paradigms in different parts of the curriculum.

Requiring teachers, who are the corner stone of the implementation of any curriculum whatsoever, to have a formal education in CS is a major challenge for every country that implements or aims at implementing a new CS curriculum, since it requires a sort of a bootstrapping process, starting to teach CS when there are not enough certified teachers nor suitable programs for training pre-service CS teachers. Fortunately, Israel succeeded in this challenge, since even before the wide implementation of the curriculum, which followed the pilot phase, the

Ministry of Education invested massive efforts and funding to address this issue. Professional development courses that focused on how to teach the new curriculum were provided to all teachers. Those who did not have a CS background were provided with hundreds of hours of training programs, introducing them to the discipline.

After the design of the curriculum was completed, working teams were set up in the Technion, the Hebrew University, the Weizmann Institute of Science, and the Open University of Israel, sometimes two working groups in one institution. Each group was assigned one or two of the program units. Each unit was reviewed by the committee's members, and then initially implemented in a small number of schools. This implementation was accompanied by research for each of the units and the entire program. Each team consisted of researchers, educators, as well as CS high-school teachers, since their pedagogical knowledge and practical experience from their classrooms were invaluable.

Not long after the program was implemented, universities and teacher colleges started to offer complete programs towards obtaining a CS teaching certificate (e.g., Ref. [46]), thus ending the bootstrapping phase. In these programs the teachers acquire pedagogical knowledge and pedagogical content knowledge. They learn about instructional approaches and methods for teaching CS effectively, current curricula, the history of the discipline and its nature, employing current technology for teaching CS, and more. In addition, the "Machshava" center for computer science teachers was established [90], offering teaching resources and professional development activities (e.g., Ref. [91]). Today, professional development courses for CS in-service teachers are regularly offered by the Ministry of Education and by the universities. The massive and continuous efforts to educate, train, and support the teachers in their daily mission has been widely researched (e.g., Refs. [26, 35, 81, 115]), thus paving the way to new pedagogies and teaching tools.

In 2011, the Ministry of Education initiated a technological leadership program for excellent middle-school students, called the Science and Technology Excellence Program (STEP). CS was included in this program [133, 134], alongside the reinforcement of Mathematics and Physics. The goal was to expose students to the principles of CS, teach them to think in a systemic fashion, draw logical conclusions, and mainly develop abstraction and imagination abilities. The designers thought that learning CS and internalizing the principles of the discipline may contribute to success in other subjects learned in school. The curriculum introduced the learners to concepts in algorithmic thinking (also referred to as computational thinking), through unplugged and programming activities. After a few years, the CS part of this program was replaced by a new curriculum. It follows a problem-based and abstraction-based approach to algorithmic problem solving. The entire process of problem solving is carried out by moving between the four abstraction levels of algorithmic problem solving: the *problem* level, the *algorithm* level, the *program* level, and the *execution* level. In the 7th grade, algorithmic solutions are implemented in the event-driven language Scratch, whereas in the 8th and 9th grades the students use Python.

As K-12 computer science became more prevalent in countries worldwide, with some countries designing or adapting curricula that start from elementary school (and sometimes even before that), Israel started debating whether the pipeline of CS education can and should be extended down to elementary school [16]. In 2015, it was decided to start teaching CS in elementary schools to 4th–6th grade students. These grades were chosen because younger students are still coping with mother tongue acquisition and basic mathematics. A curriculum called *computer science and robotics* was designed by the Ministry of Education for a soft introduction to computer science. Algorithms are implemented in Scratch.

This curriculum is currently taught in a few hundred schools; it is still in a pilot phase, and it has already undergone some changes. However, its implementation poses a considerable challenge. First, teaching CS to young children, without reducing it only to programming (let alone coding), is a difficult educational task. Second, once again, we are facing the bootstrapping process of training teachers when training programs for prospective CS elementary teachers do not yet exist, and although we have a nice cadre of high-school teachers (though not enough), there are hardly any elementary school teachers who have the disciplinary knowledge required to teach CS. The Ministry of Education and the Ministry of Finance allocated funding for the purpose of teacher training and every teacher that plans to teach this curriculum takes a training course of a few dozen hours. Research on this issue is now taking place (e.g., Refs. [22, 43]).

3 Fundamental Ideas and Concepts of CS

There are many fundamental ideas and concepts of CS. Here we will focus on those that are addressed by Israeli CS education researchers. This section is organized into seven subsections, the first six of which deal with some CS fundamental ideas and concepts, respectively, starting with abstraction (Sect. 3.1), followed by problem-solving paradigm (Sect. 3.2), correctness and efficiency (Sect. 3.3), non-determinism (Sect. 3.4), concurrency (Sect. 3.5), and Reduction (Sect. 3.6). The seventh section will deal with research about problem-solving strategies.

3.1 Abstraction

Abstraction is the core of CS, its most fundamental idea. CS experts use abstraction in different contexts and move freely between levels of abstraction. In fact, many of the ideas described below in the following sections, are manifestations of abstraction. Abstraction can be used for modeling, formalization, generalization, ignoring details, transfer among domains, and more. Since abstraction is also a central idea in the context of the mature discipline of mathematics, several publications discussed the differences and similarities between these disciplines regarding this idea (e.g., Refs. [3, 96]).

Haberman et al. examined the teaching and learning of logic programming as a second problem-solving paradigm for high-school students (see Sect. 3.2). A major focus of this research project was the concept of abstract data types (ADT), through which the declarative and procedural aspects of logic programming can be taught. ADTs ignore or hide the concrete organization of data as well as the implementation details of manipulating the data. Haberman et al. introduced a didactic method for teaching ADTs, which uses evolving boxes – black, white, and gray, and explored its effectiveness and cognitive aspects [76, 79, 80]. This approach was shown to be effective for many students, although some used it only partially or violated the hiding of details in various ways.

The teaching and learning of ADTs was also studied in the context of the procedural paradigm, for example, the cases of the ADTs of a binary tree and a linked list in an advanced high school course [74, 75, 117]. Aharoni [2] studied the teaching and learning of ADTs in the context of a data structures course for undergraduate students. He showed the students' tendency to reduce the level of abstraction, working with ADTs from the perspective of a process rather than the perspective of an object.

The duality of process-object in terms of abstraction is a known framework from mathematics education [121]. Hazzan extended and generalized this framework, demonstrating students' tendency for reducing the level of abstraction when introduced to a new concept. She employed this framework in mathematics as well as in computer science [83], for undergraduate students in various curricular contexts. For example, computability theory [84] and graph algorithms [85]. Ginat and Blau [68] used Hazzan's framework to show the tendency of senior undergraduate CS students to reduce the levels of abstraction in the context of algorithmic problem solving. Based on Hazzan's framework, Armoni [11] introduced a didactic framework for teaching procedural abstraction for novices, differentiating between four levels of abstraction [111]: problem, algorithm (object), program, and execution. The effectiveness of this framework was shown in the context of an introductory CS course for middle-school students [124], indicating an even extended effect for girls [123]. Ginat [66] relied on the same 4-level hierarchy to show that following an intervention in which declarative observations were employed in an algorithmic problem-solving course for undergraduate students, the students were able to work at higher levels of abstraction.

Pattern-Oriented Instruction (POI) is an instructional approach for teaching problem-solving, using algorithmic patterns, which by nature employ procedural abstraction. Indeed, the integration of this approach in an introductory CS course for high-school students was shown to improve students' abstraction skills [104, 106]. Haberman and Muller [77] analyzed and compared the instructional approach of POI with an ADT-oriented approach regarding abstraction. Ginat and Menashe [69] utilized algorithmic patterns to define a taxonomy for assessing students' learning outcomes regarding algorithmic problem solving. This taxonomy was built on the SOLO taxonomy [34], which takes the perspective of abstraction. The higher is the level reached by a student, the more abstract is the employment of algorithmic patterns in the student's algorithmic solution. Ginat and Menashe used this taxonomy to assess high-school students' solutions for several algorithmic problems.

Students' difficulties with abstraction were shown by several researchers, in different levels and curricular contexts. For example, Omar et al. [107] in the context of decomposition and reuse, with high-school students; Haberman et al. [82] in the context of procedural abstraction and in particular the concept of an algorithm, with high-school students; Lavy et al. [94, 108], in the context of an undergraduate course on object-oriented programming; Ginat and Alankry [67] in the context of concatenation of formal languages in a high-school course on computational models; Ginat et al. [70] in the context of algorithmic problem solving, with both undergraduate and high-school students; Ginat [64] in the context of the abstract notions of 'as-if' and 'don't-care' for solving algorithmic problems, with undergraduate students.

In contrast, Alexandron et al. [6] found that a course on the scenario-based paradigm had a positive effect on abstraction skills, of both graduate and high-school students. In a very different context of a high-school course on computational science, Taub et al. [128] found that when developing simulations of physics phenomena, moving between levels of abstraction in the context of CS facilitates moving between levels of abstraction in the context of physics.

In all these contexts, the level of programming in a high-level language was a lower level of abstraction, below the levels of the problem and its algorithmic solution. In contrast, Schocken and Nisan developed a course in which abstraction is a major recurring context, and in which students move between even lower levels of abstraction below the programming level. In a series of projects the students design a computer system in a bottom-up manner. The students start from the very low level of logic gates, and gradually move up through hardware and software levels (e.g., computer architecture, machine language, assembly, compiler, and operating systems) [118].

3.2 A Problem-Solving Paradigm

Problem solving lies at the heart of CS, and hence, a problem-solving paradigm is an important CS idea. It expresses the expertise of computer scientists as problem solvers, who among other things, are experts in choosing the appropriate, best fit way of thinking by which they approach the problem at hand. In other words, they choose the appropriate form of modeling the problem space, or yet in other words – abstracting the problem space. Thus, every problem-solving paradigm has its own abstraction method. Furthermore, other recurring CS ideas and concepts (e.g., recursion) can be relevant to all or several paradigms but expressed in a different form in each of them.

A more concrete perspective on problem-solving paradigms is expressed through programming languages, where each problem-solving paradigm can be implemented by different programming languages. Although this concrete perspective is certainly important, the more abstract level of problem-solving paradigm is necessary to understand this idea (rather than just using it).

In general, educational research on problem-solving paradigms can be divided into a few types: how to teach a certain paradigm, what should be the first paradigm taught, how does the first paradigm affect the teaching of other paradigms, students' difficulties when learning a specific paradigm, comparing different paradigms regarding different aspects, and the meta-type of learning the idea of a problem-solving paradigm.

Typically, undergraduate CS programs include more than one problem-solving paradigm, where the first is introduced in CS1 and additional ones are introduced in more advanced courses. Several Israeli studies examined the idea of a problem-solving paradigm in the context of higher education. Some dealt with students' understanding of specific OO concepts [25, 94, 108]. Not surprisingly, some dealt with the controversial issue of Object-first-vs-object-later, which extensively occupied the international CS education community [53, 73, 132].

Alexandron et al. examined issues concerning the teaching of the scenario-based paradigm in a graduate course, using the LSC (Live Sequence Charts) programming language [39]. In the scenario-based approach, "a program consists of a set of multi-modal scenarios. The execution mechanism follows all scenarios simultaneously, adhering to them all, so that any run of the program is legal with relation to the entire set of scenarios" [23]. They found that when using LSC, the students tended to adopt an external and usability-oriented view, whereas when using another paradigm of their choice to solve a programming challenge, they adopted an internal and implementation-oriented view [5]. In their work they also referred to the "mother tongue" issue. They found [4] that previous programming experience can affect the learning of scenario-based programming, leading students to use familiar programming patterns in a manner that interferes with the new concepts, resulting in their poor usage and even unexpected behavior of the students' artifacts. The paradigm of scenario-based programming is also used for Plethora, a new educational environment for teaching computational problem solving to elementary-school students [23].

The issue of a problem-solving paradigm was widely examined in the context of K-12 CS education. As mentioned in Sect. 2, one of the underlying principles of the high-school CS curriculum [51] is that students should be exposed to more than one programming paradigm, to "another language, of radically different nature, that suggests alternative ways of algorithmic thinking" (p. 76). In line with this principle, several courses dealing with different paradigms were developed, and extensive research accompanied their development and enactment. For example, several papers (e.g., [93]) reported on the learning of functional programming (in Scheme), examining students' conceptions of automated assignment [109] and the functional evaluation process [92], and the conceptual conflict between the procedural and the functional paradigms demonstrated by the students [92]. An interesting paper examined phenomena known from mathematics education, in the context of functional programming [110]. They found evidence of a clash between the conception of functions as actions on objects and their formal conception, a clash that occurred between the conception of a function as a change and its conception as a mapping, as well as between the conception of a chain of actions and the conception of composition of functions. Interestingly, this study led to further examination of the conception of functions in a mathematical context [97].

Teaching and learning the logic paradigm (using Prolog) was also reported in several papers, mainly through the perspective of abstract data types (see Sect. 3.1) and their use for knowledge representation, as well as problem solving (e.g., Ref. [80]). The concept of recursion was studied both in the context of the functional paradigm [98] and the logic paradigm [75], thus nicely demonstrating the idea of a problem-solving paradigm.

Numerous papers have dealt with the object-oriented (OO) paradigm, following the gradual shift in the high-school CS introductory course, from the procedural to the OO paradigm, which took place more than 15 years ago (e.g., Refs. [99, 112, 114, 122, 129]). These mostly focused on the learning of OO concepts and ideas. Haberman and Ragonis [78] discussed the similarities and differences between the OO paradigm and the logic paradigm and suggested establishing links between the two courses in the high-school program, thus supporting the learning of each paradigm, as well as promoting a coherent conception of the idea of a problem-solving paradigm.

Alexandron et al. [8, 9] developed and implemented a high-school course on scenario-based programming using LSC. They found that the course encouraged abstract thinking [6] and provided a good context for learning the concept of non-determinism [7].

Israeli middle-school students who study CS are introduced to the event-driven paradigm by means of the Scratch environment. Their learning of CS concepts in this course was reported by Meerbaum-Salant et al. [102]. A subsequent study [21] examined the transition from the event-driven paradigm (9th grade) to the OO paradigm (10th grade), by comparing the achievements of 10th-grade CS students who studied CS in middle school with the achievements of those who did not. They found that although some differences in understanding some CS1 concepts could be identified throughout the school year, by the end of the school year there were no significant differences. In another study, Meerbaum-Salant et al. [101] identified patterns of programming used by the students, which were not consistent with standard habits of programming (for example, modularization). Gordon et al. [72] argued that these habits are consistent with the scenario-based paradigm, thus hinting at its naturalness for the young students.

Finally, Stolin and Hazzan [127] reported on a course on programming paradigms for pre-service teachers, which was organized around the theme of abstraction and dealt with four paradigms (functional, procedural, OO, and concurrent). Their study investigated the students' understanding of the concept of a programming paradigm, particularly their way of relating to this concept when discussing the different paradigms.

3.3 Correctness and Efficiency

Correctness and efficiency are fundamental CS concepts when designing, analyzing, or reasoning about algorithms. They are relevant for any algorithm for a computational problem, in any model of execution. These concepts can be learned at various levels of rigor, for example, by using rigorous proofs of correctness and big-O analyses of complexity, by using verbal arguments, or by using representative test cases and loop-based estimated counts.

Ginat [55] argued for using assertions as a pedagogical tool for proving (or justifying) correctness and measuring efficiency, as well as a tool for algorithmic design (see Sect. 3.7). As a tool for establishing correctness, such assertions can be used for tagging the algorithm, as is done in algorithmic verification [42, 87], for example, as loop invariants or as entry and exit assertions for parts of an algorithm (e.g., segments and solutions for sub-tasks). Tagging with loop invariants can serve to analyze efficiency by establishing a worst-case bound of the number of rounds until the invariant does not hold.

Ginat contended and illustrated that despite such assertions' perceived formal nature and difficulty, their use can be taught to students at a rather intuitive level. This pedagogical tool was embedded in the textbook for the first introductory CS course for 10th-grade students. It was developed during the implementation of the 1990s' curriculum at the request of the Ministry of Education (the textbook used the procedural paradigm and Pascal as a programming language. It has been out of use, since the problem solving-paradigm used in the introductory courses and the programming language were changed).

This book also included a chapter on efficiency, in terms of both time and space. Ginat [54] argued that unlike the concept of complexity, which requires formal mathematical tools, efficiency can be taught relatively early, after introducing repeated execution. Students are introduced to problems with multiple solutions that vary in their efficiency; they learn how to express the number of iterations in a loop in terms of input size, and to examine the need to use arrays whose sizes depend on the input size. This relatively gentle introduction can serve as the basis of the spiral teaching of efficiency. Later, in a mandatory unit dealing with data structures, the students are re-acquainted with efficiency, this time using the big-O-notations; however, in line with spiral teaching, the learning of efficiency is suitable for the age level of sophomore or senior high-school students.

Gal-Ezer and Zur [49] investigated students' misconceptions of efficiency and related achievements following the learning of this concept as introduced in the textbook. They found that intuitive rules [125] often govern students' conception of efficiency. Specifically, they tend to think that the shorter a program, the more efficient it is; the fewer variables in a program, the more time-efficient it is; two programs that include the same statements (no matter the order) are equally efficient, and two programs that accomplish the same task are also equally efficient.

Gal-Ezer et al. [52] also aimed at an early introduction of efficiency, in the context of a CS1 course for undergraduate students. The introduction was gradual, but unlike the introductory high-school course, it was more formal, also including the big-O notations. The instructional approach employed by Gal-Ezer et al. was found to be effective and students' achievements improved (compared to previous semesters).

In a series of studies, Ben-David Kolikant et al. studied students' conceptions of correctness. Ben-David Kolikant and Pollack [32] found that high-school students were tolerant regarding some errors in their programs and were satisfied with programs that "worked in general" or "worked for many input examples". Later, Ben-David Kolikant [29] found that high-school students as well as college CS graduates were satisfied with "relatively correct programs". Ben-David Kolikant and Mussai [31] took the perspective of incorrectness, and investigated high-school students' misconceptions and their connections to the students' existing knowledge. They found that students' conception of incorrectness did not complement their conception of correctness. Rather, they tended to assume a gray area in which programs are partially incorrect, since they achieve part of their goals. This conception stemmed from a summative grading scheme that was often used by teachers to grade students' assignments, according to which a program may get a non-zero score if it achieves some of its goals or made some progress towards a goal. For these students, incorrect programs were only those that deserved a score of 0.

Ben-David Kolikant and Pollack [32] challenged the norm among high-school students and their teachers, according to which it is sufficient to successfully run a program in order to establish its correctness. Their instructional approach was intended to establish a new norm, according to which incorrect programs cannot be tolerated. Using mathematical problems, the students developed mathematically oriented programming skills, realizing that correct algorithmic solutions can be found by first analyzing the problem at hand and then concluding what the output should be for all possible subsets of inputs. These subsets can later serve for choosing representative test cases, thus achieving a high-coverage testing together with explanatory proofs. To this end, they used class discussions, since social contexts are known to be effective for acquiring norms.

The idea of correctness is especially challenging in the context of concurrent and distributed computing, since a successful run of a specific test case is insufficient even when arguing that a program is correct regarding this test case. All possible interleavings should be considered. Indeed, several studies demonstrated students' difficulties with this idea in a concurrent context (see Sect. 3.5).

3.4 Nondeterminism

Nondeterminism (ND) is a challenging abstract CS idea that is relevant in many CS areas and has many facets. For example, ND is manifested in the context of computational models through nondeterministic automata (NDA), as a way of augmenting expressiveness. Similarly, in the context of programming, ND can be manifested through nondeterministic algorithmic constructs, again augmenting expressiveness. In this context it represents the notion of *don't care*, since all possible computations are considered equally good. In both these contexts, using ND is a matter of choice. In contrast, in the contexts of concurrent or distributed computing, ND represents unpredicted behaviors stemming from a set of possible interleavings or timings, over which the designer has no control. Note that students may face this kind of ND at a relatively early age, since even the block-based language of Scratch, which is very popular in K-9 CS education, is inherently concurrent. In all these manifestations of ND, ignoring details or being prepared for unpredicted situations is inherent, hence the deep connection of ND with abstraction.

In the Israeli high-school program, ND is included in the elective course on computational models, through NDA. Armoni and Gal-Ezer [14] described the rationale and underlying considerations as well as the guidelines for teaching ND in high school as part of the computational models course. They also studied the ways in which the students used ND. The findings indicated that many students tended not to fully exploit the expression power of NDA and demonstrated several patterns of partial use of ND. Following this study, they continued to study the teaching and learning of ND at the undergraduate level in the same curricular context – an undergraduate course on automata and formal languages [15]. They analyzed students' errors when using ND, their tendency to use ND and the quality of their solutions regarding ND, and found various levels (sometimes unsatisfactory) of their tendency to use ND and the quality of using it. This motivated further exploration of the teaching and learning of this concept. In another study, Armoni, Lewenstein, and Ben-Ari [20] found that undergraduate students do not perceive NDA as unpredictable entities. However, a simple change in the teaching process of the course proved highly efficient and led to students having better perceptions of NDA and ND. These perceptions were the motivation for turning to explore the development and treatment of ND throughout the history of CS and CS education, in its various facets and manifestations [12]. It yielded a taxonomy of ND manifestations as well as recommendations for the teaching of ND, at all curricular levels.

The teaching and learning of ND at the high-school level was also studied by Alexandron et al. [7], as part of their research on teaching scenario-based programming in high school using LSC. LSC includes various forms of ND [39]. As a scenario-based language, there are no assumptions regarding ordering (unless explicitly stated otherwise, using the language constructs). In addition, some of LSC's constructs are nondeterminisic (for example, a weighted *select* construct, which can be viewed as a probabilistic version of Dijkstra's guarded commands [40]). Alexandron et al.'s study demonstrated that high-school students can perceive, understand, and effectively use these manifestations of ND.

Ginat [63] studied the computational notion of *don't care*. This was done in an algorithmic deterministic context, where students were expected to find or understand efficient solutions in which certain aspects of the problem were ignored. He pointed out several substantial difficulties students had regarding this notion. Despite the deterministic context, Ginat linked this to ND, viewing the notion of "*don't care*" as universal, in the sense that its essence is the same in both its deterministic and nondeterministic manifestations. In another study, Ginat and Alankry [67] studied high-school students' understanding and performance of concatenation in the context of the computational models course, and specifically, concatenation of formal languages. Concatenation of formal languages involves ND thinking (expressed in the nondeterministic canonical construction that based on two given automata, constructs a nondeterministic automaton for the concatenation language).

A noted above, ND is strongly related to concurrency. Several studies explored the teaching and learning of concurrency (see Sect. 3.5), but although the connection was acknowledged, the learning and teaching of ND was not discussed in those publications.

3.5 Concurrency

Concurrency is another abstract and challenging CS concept. As noted above, it is strongly related to ND. Namely, ND is inherent in any model of computation in which multiple interleavings are possible. Traditionally, concurrency is considered an advanced idea, and it is usually addressed in advanced courses on concurrent and distributed algorithms, computational models (e.g., through Petri nets), semantics of programming languages (e.g., through constructs such as Dijkstra's guarded commands), and more. However, as noted in Sect. 3.4, nowadays concurrency may be inherent even in computing environments such as Scratch, which are designated for young children.

In Israel, concurrency at the K-12 level was addressed in two educational research contexts:

1. The context of teaching concurrent and distributed computing.
2. The context of basic CS courses for young children using a concurrent language.

The 1990s' high-school CS curriculum [47] included a 45-hour course on concurrent and distributed computing (CDC). It was an advanced course, taken at grade 11 or 12, following a chain of two introductory courses and a course presenting a second paradigm, and in parallel with an advanced course on data structures and software design. Several studies accompanied the iterative development of this course. Ben-Ari and Ben-David Kolikant [24] described the course, the motivation for teaching it, and the pedagogical considerations behind it, and investigated the students' learning. They found that the students had some difficulties, for example, with considering multiple interleavings, or in general with the concept of correctness in a concurrent model; however, students' understanding evolved throughout the course, and by the end of the course, most of them could solve problems that required process coordination and they reached high achievements. Moreover, they could discuss and explain concurrent and distributed systems using concepts learned during the course. In another study, Ben-David Kolikant et al. [33] investigated students' mental models of semaphores. Because non-viable models were detected, indicating misconceptions of semaphores, the course was updated to avoid the development of these misconceptions. Ben-David Kolikant [27] has also studied students' preconceptions of concurrency. Students' prior knowledge on synchronization (computerized as well as human) was found to be rich, but when thinking of human agents, students tended to assume capabilities that are not viable in the context of computing. This includes, for example, being able to determine when one can stop waiting, being able to spontaneously adapt to a constant rate of actions, or being able to receive a message spontaneously, without the need to perform an action, whereas sending a message required an explicit action (as is the case with hearing and talking, respectively). In line with the theory of constructivism, the course was then updated to address these preconceptions. For example, the use of authentic settings from the world of computers was preferred to the use of imagined settings (such as in the "dining philosophers" problem). In another study, Ben-David Kolikant [28] analyzed the evolvement of students' understanding of synchronization. She found evidence of a pattern-based technique, where a set of pattern problems and their solutions is acquired and used to solve new given problems. Although this technique was efficient, it also limited students' performance when they faced a problem that did not meet a known pattern. Later, Ben-David Kolikant and Ben-Ari [30] suggested that these difficulties and perceptions stem from a cultural clash between the culture of computer users, from which most of the students arrive, and the professional CS culture. Instructional approaches aimed at resolving the clashes were shown to be effective. For example, the students could handle abstract problems (such as the "dining philosophers" problem), realizing that it represents a group of concrete problems.

Schwarz and Ben-Ari [119] examined the tendency of students who learned the CDC course to use state diagrams as a tool for explaining concurrent solutions, specifically for convincing others of or refuting their correctness. They found that in general, students preferred verbal arguments rather than the use of state diagrams. However, some of the students acknowledged the potential of state diagrams and used them, and others used them only occasionally, when verbal arguments failed. Schwarz and Ben-Ari conjectured that state diagrams facilitate the formation of mental models, even if they are not used often as an argumentation tool; hence, they recommended including them in the course.

The teaching and learning of concurrency was also studied in the context of an introductory CS course for middle-school students; it used the block-based Scratch environment [102]. As an event-driven language, concurrency is manifested in Scratch in two ways: by several sprites executing scripts concurrently (Type-I concurrency), and by a sprite executing more than one script simultaneously (Type-II concurrency). Type-I concurrency was more intuitive for the students, but they had difficulties with Type-II concurrency, probably due to a reduced perception of the concept (i.e., identifying it with its specific aspect of synchronization or with a concrete process such as the set of instructions for message passing). Meerbaum-Salant et al. pointed out that although Scratch is perceived by educators as mostly suitable for young children, because of its friendly colorful block-based interface and the colorful animated artifacts, one should bear in mind that it also poses complex learning challenges, such as the abstract concept of concurrency (which cannot be avoided even in rather simple projects). Interestingly, when self-designing projects, students exhibited habits of programming that preferred fragmentized concurrency rather than sequential modularization [101], suggesting, as argued by Gordon et al. [72], that thinking in scenarios (where concurrency is inherent) is natural for the students.

3.6 Reduction

Reduction in CS education is mostly mentioned in the context of courses on the theory of computer science, where it is used to prove non-decidability or difficulty of problems. Reduction is also useful in algorithmic design, when reducing a problem to be solved to another, already solved problem. From a generalized point of view, solving a problem by reduction means transforming the problem at hand into other problems (one or more), which are already solved or are easier to solve, and using their solutions as black boxes for obtaining a solution to the problem at hand. Using this perspective, reduction is a CS recurring concept, relevant in almost every area of CS.

Gal-Ezer and Trakhtenbrot [48] studied the classic use of reduction in an undergraduate course on the theory of computer science (computability and complexity). They identified five misconceptions, indicating that even in this familiar context, the teaching of reduction has yet to improve.

Armoni et al. [18] looked into other manifestations of reduction. They examined high-school students' use of reduction in the context of the course on computational models. In this context, reduction can be used to classify formal languages and design automata using closure properties or known construction algorithms. According to the findings, many students neglected to employ reduction, even when using reduction could lead to elegant and shorter solutions. They also found that when reduction was employed, the solution's level of reduction (in terms of the cognitive distance between the problem at hand and the reduced-to problem) was often relatively low.

The next step was to examine the same issue in the context of undergraduate students [13]. Although the students' employment of reduction was somewhat better, it was not as good as one could hope, and in general, the findings indicated an unsatisfactory level of reduction-related skills.

In a study that followed, Armoni et al. [19] extended the curricular perspective. The study focused on first- and third-year undergraduate CS students and on CS graduates who learned towards obtaining a CS teaching certificate, and examined their use of reduction on CS1 questions as well as questions related to the algorithms course or the computational models course. It was a qualitative interview-based study. The findings indicated that the development of reduction-related skills evolves over time. First-year students barely ever used reduction, whereas the more mature students exhibited higher levels of awareness of the concept of reduction as well as of its potential use in different problem-solving situations. However, even they did not sufficiently exploit the power of reduction. For many students there was a clash between the abstract nature of reduction (expressed by the use of black boxes) and the tendency to work at lower levels of abstraction. They had difficulties in connecting between problems, and their use of reduction was often limited to certain curricular contexts and was not transferred to others. They doubted the legitimacy of using reduction in some contexts and did not perceive it as a rewarding problem-solving heuristic, whose use reflects high problem-solving skills. Often they did not use reduction properly, tending to open the black box and confusing a problem and its solution.

This was also studied in a quantitative study [10], focusing on the algorithms course at the undergraduate level; the findings were consistent with previous ones: often students' solutions by reduction were of a low level of abstraction, lacking a clear black-box component. The black box was corrupted, for example, by opening it and referring to the inner details of the solution hidden in it. It also seemed that students did not fully understand the nature of reduction, often failing to relate it to problems.

Based on all the findings above, a framework for teaching reduction was developed and investigated in an undergraduate algorithms course [44]. Integrating this framework into the course had a positive effect on the students' use and understanding of reduction.

3.7 Problem-Solving Strategies

CS experts deal with computational problems. They analyze them, classify them, and try to solve them if possible. Hence, computational problem solving constitutes a major component of CS, and as such, it makes use of CS fundamental ideas and concepts, among which are those that were covered in Sects. 3.1, 3.2, 3.3, 3.4, 3.5, and 3.6. In particular, some of them can also serve as problem-solving strategies. For example, reduction (Sect. 3.6), and specifically the use of black-box solutions to other problems, is an effective problem-solving strategy. Similarly, each problem-solving paradigm (Sect. 3.2) also constitutes a high-level strategy for problem-solving, since by choosing a paradigm we also choose how to address the problem, and some choices may be better than others, depending on the problem's characteristics; they may induce an easier or more natural path to a solution.

Algorithmic patterns (APs) were mentioned in Sect. 3.1 as the core of the beneficial instructional approach of POI, which has a positive effect on the acquirement of abstraction skills. They also constitute a problem-solving strategy that can be used to obtain elegant and efficient solutions to computational problems. APs are algorithmic solutions for problems that are canonic (for example, searching, finding the maximal element, and scanning of adjacent elements in a sequence), in the sense that they can be used in the solutions of many other problems. Unlike reduction, in which one also uses a solution to one problem to solve another, APs are not closed in black boxes, and hence can be used in various ways, including concatenation, interleaving, and nesting. The use of APs for problem solving was studied, for example, by Ragonis [113] in the context of prospective teachers, and by Muller [103], Ginat and Menashe [69] and Ginat et al. [71] in the context of high-school students.

Embedding assertions was discussed in Sect. 3.3 as a means for justifying correctness and measuring efficiency. It can also be used as a strategy for algorithmic design, as declarative insights into the problem at hand, which can lead to elegant algorithms. These are often more efficient than those based on operative thinking to which students often tend [59]. Ginat discussed and studied students' use of declarative reasoning vs. operative reasoning in additional publications (e.g., Ref. [66]).

Armoni and Ginat [17] identified and characterized the fundamental idea of reversing, or thinking in reverse. Recursion is a private case of reversing, and there are more, for example topological reversing, logical reversing, and mathematical inversion. Each of these forms is a rewarding problem-solving strategy, which requires the solvers to go beyond the natural forward thinking.

Ginat identified and reasoned about a rich variety of problem-solving strategies (e.g., on-the-fly computations [62], inductive progress [65], and binary perspectives [56]), although he did not always accompany his illuminating observations with empirical investigations. One of these strategies is decomposition, which is strongly connected to abstraction, expressing moving down the levels of abstraction. There are various ways to use decomposition, for example, top-down task decomposition or data decomposition (e.g., decomposing a range into two parts, as in divide-and-conquer). Ginat [57, 58] discussed this strategy, introducing various (sometime very sophisticated) forms of it. Ginat studied the ability of CS graduates, who taught CS or math (in high-school or college) to solve a non-standard task whose solution requires a sort of geometrical decomposition [61]. He found that most of the graduates did not turn to this strategy and failed to reach a correct solution. Omar et al. [107] studied high-school students' use of decomposition (by means of functions) when solving programming tasks. Similar to Ginat, they found unsatisfactory use of this form of decomposition.

Besides failing to employ certain rewarding strategies, insufficient mastery of problem-solving strategies may also be expressed by adopting ineffective strategies. Ginat [60] found that excellent high-school CS students often turn to the *design-by-keyword* strategy, in which their choice of solution is influenced by specific words or phrases in the problem description, which often misguide them to non-efficient or incorrect solutions.

4 Concluding Remarks

This chapter presented a review of Israeli research on computer science education. It is by no means an exhaustive review, which could hardly be done in one chapter.

Our decision to focus on studies that deal with fundamental ideas and concepts of computer science has led us to leave out of this chapter many other important studies. For example, there is rich body of research on visualization in computer science education, on teacher preparation and professional development, and more. By focusing on computer science we also left out many interesting publications on software engineering and in particular on project design. However, we see the issue of the nature of CS as a major aspect of CS education, especially since research worldwide indicates that there are many misconceptions regarding the nature of the disciplines among the general public and hence also among our students who are newcomers to CS. Addressing these misconceptions and the inaccurate image of CS includes also stressing what CS is. Research on its fundamental ideas and concepts is highly important for achieving this educational goal.

Israeli researchers cooperate with the international community of computing education, leading to extensive research that was not represented in this chapter. Specifically, Israeli researchers have been taking part in efforts initiated or motivated by international groups. These include for example, working groups [88, 89, 100, 130] and international educational committees [38, 41, 45, 126, 131]. These activities have had an impact worldwide and Israel became an influential factor regarding the development of K-12 CS curricula, their implementation, and their research. Some of these efforts examined the current K-12 CS curricula in the US and Europe [126, 131], and then touched upon challenging issues such as designing K-12 curricula, or at least an appropriate framework [38] that can serve all European countries. The main concerns of the different teams were, and still are, as follows: the nature of CS, and specifically, the confusion that still exists between CS and other areas such as ICT and digital competencies [131], inclusion [38], and the challenge of training knowledgeable teachers [41].

Future work is already underway. The Israeli CS education research community explores a wide range of issues, elaborating on themes mentioned above as well as others. CS education research has developed worldwide and is widely appreciated. Hence, more aim at enrolling in graduate research programs in this area, and more research is done. Additional computing areas have been developed and call for research regarding their teaching and learning at all levels. Furthermore, older curricula need to be updated and implemented, with research conducted in parallel. New problems arise, and researchers find more challenges to address. The Israeli CER community will broaden the connection with the World CER community and hopefully will continue to contribute to the area of CS education and CS education research.

References

1. ACM Curriculum Committee on Computer Science (1968). Curriculum 68: Recommendations for academic programs in computer science. *Communications of the ACM* 11(3), 151–197.

2. Aharoni, D. (2000). Cogito, ergo sum! Cognitive processes of students dealing with data structures, In *Proceedings of the 31stSIGCSE Technical Symposium on Computer Science Education*, 26–30.
3. Aharoni, D., and Leron, U. (1997). Abstraction is hard in computer-science too. In *Proceedings of the Conference of the International Group for the Psychology of Mathematics Education* (PME), 2:9–16.
4. Alexandron, G., Armoni, M., Gordon, G., and Harel, D. (2012). The effect of previous programming experience on the learning of scenario-based programming. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*, 151–159.
5. Alexandron, G., Armoni, M., Gordon, G., and Harel, D. (2014). Scenario-based programming, usability oriented perception. *ACM Transactions on Computing Education* 14(3), 21:1–23.
6. Alexandron, G., Armoni, M., Gordon, M., and Harel, D. (2014). Scenario-based programming: reducing the cognitive load, fostering abstract thinking. In *Proceedings of the 36th International Conference on Software Engineering* (ICSE), 311–320.
7. Alexandron, G., Armoni, M., Gordon, G., and Harel, D. (2016). Teaching nondeterminism through programming. *Informatics in Education* 15(1), 1–23.
8. Alexandron, G., Armoni, M., Gordon, M., and Harel, D. (2017). Teaching scenario-based programming: an additional paradigm for the high school computer science curriculum, Part 1. *Computing in Science & Engineering* 19(5), 58–67.
9. Alexandron, G., Armoni, M., Gordon, M., and Harel, D. (2017). Teaching scenario-based programming: an additional paradigm for the high school computer science curriculum, Part 2. *Computing in Science & Engineering* 19(6), 64–71.
10. Armoni, M. (2009). Reduction in CS: a (mostly) quantitative analysis of reductive solutions to algorithmic problems. *Journal on Educational Resources in Computing* 8(4), 11:1–30.
11. Armoni, M. (2013). On teaching abstraction in computer science to novices. *Journal of Computers in Mathematics and Science Teaching* 32(3), 265–284.
12. Armoni, M., and Ben-Ari, M. (2009). The concept of nondeterminism: its development and implications for education. *Science & Education* 18(8), 1005–1030.
13. Armoni, M., and Gal-Ezer, J. (2006). Reduction – an abstract thinking pattern: the case of the computational models course. In *Proceedings of the 37thSIGCSE Technical Symposium on Computer Science Education*, 389–393.
14. Armoni, M., and Gal-Ezer, J. (2006). Introducing non-determinism. *Journal of Computers in Mathematics and Science Teaching* 25(4), 325–359.
15. Armoni, M., and Gal-Ezer, J. (2007). Non-determinism: an abstract concept in computer science studies. *Computer Science Education* 17(4), 243–262.
16. Armoni, M., and Gal-Ezer, J. (2014). Early computing education – Why? What? When? How? *ACM Inroads* 5(4), 54–59.
17. Armoni, M., and Ginat, D. (2008) Reversing: a fundamental idea in computer science. *Computer Science Education* 18(3), 213–230.
18. Armoni, M., Gal-Ezer, J., and Tirosh, D. (2005). Solving problems reductively. *Journal of Educational Computing Research* 32(2), 113–129
19. Armoni, M., Gal-Ezer, J., and Hazzan, O. (2006). Reductive thinking in computer science. *Computer Science Education* 16(4), 281–301.
20. Armoni, M., Lewenstein, N., and Ben-Ari, M. (2008). Teaching students to think nondeterministically. In *Proceedings of the 39thSIGCSE Technical Symposium on Computer Science Education*, 4–8.
21. Armoni, M., Meerbaum-Salant, O., and Ben-Ari, M. (2015). From Scratch to "real" programming. *ACM Transactions on Computing Education* 14(4), 25:1–15.
22. Armoni, M., Gal-Ezer, J., and Ulmer, C. Professional development of primary school teachers participating in a pilot project on teaching computer science to fourth graders. In preparation.
23. Armoni, M., Gal-Ezer, J., Harel, D., Marelly, R., and Szekely, S. (In Press). Plethora of skills: a game-based platform for introducing and practicing computational problem solving, to be published in: H. Abelson & K. Siu-Cheung (Eds.) *Computational Thinking Curricula in K-12: International Implementations*. MIT Press. Cambridge, MA.

24. Ben-Ari, M., and Ben-David Kolikant, Y. (1999). Thinking parallel: the process of learning concurrency. In *Proceedings of the 4th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education*, 13–16.
25. Benaya, T., and Zur, E. (2008). Understanding object oriented programming concepts in an advanced programming course. In *Proceedings of the 2nd International Conference on Informatics in Secondary Schools: Evolution and Perspective* (ISSEP), *Lecture Notes in Computer Science* (LNCS 5090), 161–170.
26. Ben-Bassat Levy, R., and Ben-Ari, M. (2007). We work so hard and they don't use it: acceptance of software tools by teachers. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (ITiCSE), 246–250.
27. Ben-David Kolikant, Y. (2001) Gardeners and cinema tickets: high school students' preconceptions of concurrency. *Computer Science Education* 11(3), 221–245.
28. Ben-David Kolikant, Y. (2004). Learning concurrency: evolution of students' understanding of synchronization. *International Journal of Human-Computer Studies* 60(2), 243–268.
29. Ben-David Kolikant, Y. (2005). Students' alternative standards for correctness. In *Proceedings of the 1st International workshop on Computing Education Research* (ICER), 37–43.
30. Ben-David Kolikant, Y., and Ben Ari, M. (2008). Fertile zones of cultural encounter in computer science education. *Journal of the Learning Science* 18(1), 1–32.
31. Ben-David Kolikant, Y., and Mussai, M. (2008) "So my program doesn't run!" Definition, origins, and practical expressions of students' (mis)conceptions of correctness. *Computer Science Education* 18(2), 135–151,
32. Ben-David Kolikant, Y., and Pollack, S. (2004) Establishing computer science professional norms among high-school students. *Computer Science Education* 14(1), 21–35.
33. Ben-David Kolikant, Y., Ben-Ari, M., and Pollack, S. (2000). The anthropology of semaphores. In *Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education*, 21–24.
34. Biggs, J.B., and Collis, K.F. (1982). *Evaluating the Quality of Learning: The SOLO Taxonomy (Structure of the Observed Learning Outcome)*. Academic Press.
35. Brandes, O., and Armoni, M. (2019). Using action research to distill research-based segments of pedagogical content knowledge of K-12 computer science teachers. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education* (ITiCSE), 485–491.
36. Breuer, S, Gal-Ezer, J., and Zwas, G. (1990). Microcomputer laboratories in mathematics education. *Computers and Mathematics* 19(3), 13–34.
37. Bruner, J.S. (1960). *The Process of Education*. Harvard University Press. Boston, MA.
38. Caspersen, M., Diethelm, I., Gal-Ezer, J., McGettrick, A., Nardelli, E., Passey, D., Rovan, B., and Webb, M. (2022). *Informatics References Framework for School.* https://www.informaticsforall.org/the-informatics-reference-framework-for-school-release-february-2022/
39. Damm, W., and Harel, D. (2001). LSCs: Breathing life into message sequence charts. *Formal Methods in System Design* 19(1), 45–80.
40. Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8), 453-457.
41. Ericson, B., Armoni, M., Gal-Ezer, J., Seehorn, D., Stephenson, C., and Tree, F. (2008). *Ensuring Exemplary Teaching in an Essential Discipline: Addressing the Crisis in Computer Science Teacher Certification, Final Report of the CSTA Teacher Certification Task Force*. ACM. New York, NY.
42. Floyd, R. W. (1967). Assigning meaning to programs. In *Proceedings of Symposia in Applied Mathematics, American Mathematical Society* 19, 19–32.
43. Friebroon-Yesharim, M., and Armoni, M. (2022). The tale of an intended CS curriculum for 4th graders, the case of abstraction. In *Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education* (ITiCSE), pp. 623.

44. Gaber, I., Armoni, M., and Statter, D. (2021). Teaching reduction as an algorithmic problem solving strategy. In *Proceedings of the 3rd International Conference on Computer Science and Technology in Education* (CSTE), 19–26.
45. Gagliardi, F., Hankin, C., Gal-Ezer, J., McGettrick, A., and Meitern, M. (2016). *Advancing Cybersecurity Research and Education in Europe: Major Drivers of Growth in the Digital Landscape*. https://www.acm.org/binaries/content/assets/publipolicy/2016_euacm_cybersecurity_white_paper.pdf
46. Gal-Ezer, J., and Harel, D. (1998). What (else) should CS educators know?, *Communications of the ACM*, 41(9), 77–84.
47. Gal-Ezer, J., and Harel, D. (1999). Curriculum and course syllabi for high-school computer science program. *Computer Science Education* 9(2), 114–147.
48. Gal-Ezer, J., and Trakhtenbrot, M. (2016): Identification and addressing reduction-related misconceptions, *Computer Science Education* 26(2–3), 80–103.
49. Gal-Ezer, J., and Zur, E. (2004). The efficiency of algorithms – misconceptions. *Computers and Education* 42(3), 215–226.
50. Gal-Ezer, J., and Zwas, G. (1984). An Algorithmic Approach to Linear Systems. *International. Journal of Mathematics Education in Science and Technology* 15(4), 501–519.
51. Gal-Ezer, J., Beeri, C., Harel, D., and Yehudai, A. (1995). A high-school program in computer science. *Computer* 28(10), 73–80.1
52. Gal-Ezer, J., Vilner, T., and Zur, E. (2004). Teaching efficiency at CS1 level: a different approach. *Computer Science Education* 14(3), 235–248.
53. Gal-Ezer, J. Vilner, T., and Zur, E. (2009). Has the paradigm shift in CS1 a harmful effect on data structures courses: a case study. In *Proceedings of the 40th Technical Symposium on Computer Science Education* (SIGCSE), 126–130.
54. Ginat, D. (2001). Early algorithm efficiency with design patterns. *Computer Science Education* 11(2), 89–109.
55. Ginat, D. (2001). Loop invariants, exploration of regularities, and mathematical games. *International Journal of Mathematical Education in Science and Technology* 32(5), 635–651.
56. Ginat, D. (2002). Effective binary perspectives in algorithmic problem solving. *Journal on Educational Resources in Computing* 2(2), 4–12.
57. Ginat, D. (2002). On various perspectives of problem decomposition. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, 331–335.
58. Ginat, D. (2003). Decomposition diversity in computer science—beyond the top-down icon. *Journal of Computers in Mathematics and Science Teaching 22*(4), 365–379.
59. Ginat, D., (2003). Seeking or skipping regularities? Novice tendencies and the role of invariants. *Informatics in Education* 2(2), 211–222.
60. Ginat, D. (2003). The novice programmers' syndrome of design-by-keyword. In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE), 154–157.
61. Ginat, D. (2008). Design Disciplines and Non-specific Transfer. In *Proceedings of the International Conference on Informatics in Secondary Schools: Evolution and Perspectives* (ISSEP), *Lecture Notes in Computer Science* (LNCS, 5090), 87–98.
62. Ginat, D. (2009). On the non-modular design of on-the-fly computations. *Inroads – SIGCSE bulletin* 41(4), 35–39.
63. Ginat, D. (2009). The overlooked don't-care notion in algorithmic problem solving. *Informatics in Education* 8(2), 217–226.
64. Ginat, D. (2010). The baffling CS notions of "as-if" and "don't-care". In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (SIGCSE), 385–389.
65. Ginat, D. (2014). On Inductive Progress in Algorithmic Problem Solving. *Olympiads in Informatics* 8, 81–91.
66. Ginat, D. (2021). Abstraction, declarative observations and algorithmic problem solving. *Informatics in Education* 20(4), 567–582.
67. Ginat, D., and Alankry R. (2012). Pseudo abstract composition: the case of language concatenation. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE), 28–33.

68. Ginat, D., and Blau, Y. (2017). Multiple levels of abstraction in algorithmic problem solving. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. 237–242.
69. Ginat, D., and Menashe, E. (2015). SOLO taxonomy for assessing novices' algorithmic design. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (SIGCSE). 452–457.
70. Ginat, D., Shifroni, E., and Menashe, E. (2011). Transfer, cognitive load, and program design difficulties. In *Proceedings of The 5th International Conference on Informatics in Secondary Schools: Evolution and Perspective* (ISSEP), *Lecture Notes in Computer Science* (LNCS 7013), 165–176.
71. Ginat, D., Menashe, E., and Taya, A. (2013). Novice Difficulties with Interleaved Pattern Composition. In *Proceedings of the 5th International Conference on Informatics in Schools: Situation, Evolution and Perspective* (ISSEP), *Lecture Notes in Computer Science* (LNCS 7780), 57–67.
72. Gordon, M., Marron, A., and Meerbaum-Salant, O. (2012). Spaghetti for the main course?: observations on the naturalness of scenario-based programming. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE), 198–203.
73. Green, A., Armoni, M., and Ginat, D. Object-first vs. object-Second. In preparation.
74. Haberman, B. (2002). Frames and boxes – A pattern-based method for manipulating binary trees. *Inroads – SIGCSE Bulletin* 34(4), 60–64.
75. Haberman, B. (2004). High-School students' attitudes regarding procedural abstraction. *Education and Information Technologies* 9(2), 131–145.
76. Haberman, B. (2008). Formal and practical aspects of implementing abstract data types in the prolog instruction. *Informatica* 19(1), 17–30.
77. Haberman, B., and Muller, O. (2008). Teaching abstraction to novices: Pattern-based and ADT-based problem-solving processes. In *Proceedings of the 38th Annual Frontiers in Education Conference* (FIE), F1C:7–12.
78. Haberman, B., and Ragonis, N. (2010). So different though so similar? Or vice versa? Exploration of the logic programming and the object-oriented programming paradigms. *Issues in Informing Science and Information Technology* 7, 393–402.
79. Haberman, B., and Scherz, Z. (2009). Connectivity between abstraction layers in declarative ADT-based problem-solving processes. *Informatics in Education* 8(1), 3–16.
80. Haberman, B., Shapiro, E., and Scherz, Z. (2002). Are black boxes transparent? High school students' strategies of using abstract data types. *Journal of Educational Computing Research* 27(4), 411–436.
81. Haberman, B., Lev, E., and Langley, D. (2003). Action research as a tool for promoting teacher awareness of students; conceptual understanding. In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE), 144–148.
82. Haberman, B., Averbuch, H., and Ginat, D. (2005). Is it really an algorithm? The need for explicit discourse. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (ITiCSE), 74–78.
83. 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.
84. Hazzan, O. (2003). Reducing abstraction when learning computability theory. *Journal of Computers in Mathematics and Science Teaching* 22(2), 95–117.
85. Hazzan, O., and Hadar, I. (2005). Reducing abstraction when learning Graph Theory. *Journal of Computers in Mathematics and Science Teaching* 24(3), 255–272.
86. Hazzan, O., Gal-Ezer, J., and Blum, L. (2008). A model for high school computer science education: the four key elements that make it! In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, 281–285.
87. Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM* 12(10), 576–580.

88. Holz, H. J., Applin, A., Haberman, B., Joyce, D., Purchase, H., and Reed, C. (2006). Research methods in computing: what are they, and how should we teach them? In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education* (ITiCSE-WGR), 96–114.
89. Hubwieser, P., Armoni, M., Brinda, T., Dagiene, V., Diethelm, I., Giannakos, M. N., Knobelsdorf, M., Magenheim, J., Mittermeir, R., and Schubert, S. (2011). Computer science/informatics in secondary education. In *Proceedings of the 16thAnnual Conference Reports on Innovation and Technology in Computer Science Education – Working Group Reports* (ITiCSE-WGR), 19–38.
90. Israel National Center for Computer Science Teachers (2002). "Machshava": the Israeli National Center for high school computer science teachers, In *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE), pp 234.
91. Lapidot, T., and Aharoni, D. (2007). The Israeli summer seminars for CS leading teachers. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (ITiCSE), pp. 318.
92. Lapidot T., Levy D., and Paz T. (1999). Implementing constructivist ideas in a functional programming course for secondary school. In *Proceedings of the Workshop on Functional and Declarative Programming in Education*, 29–31.
93. Lapidot T., Levy D., and Paz T. (2000). Teaching functional programming to high school students. In *Proceedings of the International Conference on Mathematics/Science Education and Technology* (M/SET).
94. Lavy, I., Rashkovits, R., and Kouris, R. (2009). Coping with abstraction in object orientation with a special focus on interface classes. *Computer Science Education* 19(3), 155–177.
95. Leron, U. (1985). Logo today: vision and reality. *The Computing Teacher* 12(5), 26-32.
96. Leron, U. (1987). Abstraction barriers in mathematics and computer-science. In *Proceedings of the Third International Conference on LOGO and Mathematics Education* (LME).
97. Leron, U., and Paz, T. (2014). Functions via everyday actions: Support or obstacle? The Journal of Mathematical Behavior 36, 126-134
98. Levy, D., Lapidot, T., and Paz, T. (2001). 'It's just like the whole picture, but smaller': Expressions of gradualism, selfsimilarity, and other pre-conceptions while classifying recursive phenomena. In *Proceedings of the 13th Workshop of the Psychology of Programming Interest Group* (PPIG), 249–262
99. Lieberman, N., Ben-David Kolikant, Y., and Beeri, C. (2011). Difficulties in learning inheritance and polymorphism. *ACM Transactions on Computing Education* 11(1), 4:1–23.
100. McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Ben-David Kolikant, Y., Laxer, C., Thomas, L., Utting, I., and Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *In Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education* (ITiCSE-WGR), 125–180.
101. Meerbaum-Salant, O., Armoni, M., and Ben-Ari, M. (2011). Habits of programming in Scratch. In *Proceedings of the 16thAnnual Joint Conference on Innovation and Technology in Computer Science Education* (ITiCSE), 168–172.
102. Meerbaum-Salant, O., Armoni, M., and Ben-Ari, M. (2013). Learning computer science concepts with Scratch. *Computer Science Education* 23(3), 239–264.
103. Muller, O. (2005). Pattern oriented instruction and the enhancement of analogical reasoning. In *Proceedings of the 1st International workshop on Computing Education Research* (ICER). 57–67.
104. Muller, O., and Haberman, B. (2008). Supporting abstraction processes in problem-solving through pattern-oriented-instruction. *Computer Science Education*, 18(3), 187–212.
105. Muller, O., and Haberman, B. (2009). A course dedicated to developing algorithmic problem solving skills – Design and experiment. In *Proceedings of 21stAnnual Workshop of the Psychology of Programming Interest Group* (PPIG), 9:1–9.

106. Nakar, L., and Armoni, M. (2022). Pattern-oriented instruction and students' abstraction skills. In *Proceedings of the 27th ACM Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE), pp. 613.
107. Omar, A., Hadar, I., and Leron, U. (2017). Investigating the under-usage of code decomposition and reuse among high school students: the case of functions. *Lecture Notes in Business Information Processing* 286, 92–98.
108. Or-Bach, R., and Lavy, I. (2004). Cognitive activities of abstraction in object orientation: an empirical study. *Inroads – the SIGCSE Bulletin* 36(2), 82–86.
109. Paz, T., and Lapidot, T. (2004). Emergence of automated assignment conceptions in a functional programming course. In *Proceedings of the 9th Annual SIGCSE Conference on Innovations and Technology in Computer Science Education* (ITiCSE), 181–185.
110. Paz, T., and Leron, U. (2009). The slippery road from actions on objects to functions and variables. *Journal for Research in Mathematics Education* 40(1), 18–39.
111. Perrenet, J., Groot, J.F., and Kaasebrood, E. (2005). Exploring students' understanding of the concept of algorithm: levels of abstraction. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (ITiCSE), 64–68.
112. Ragonis, N. (2010). A pedagogical approach to discussing fundamental object-oriented programming principles using the ADT SET. *ACM Inroads* 1(2), 42–52.
113. Ragonis, N. (2012). Integrating the teaching of algorithmic patterns into computer science teacher preparation programs. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE), 339–344.
114. Ragonis, N., and Ben-Ari, M. (2005). A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education* 15(3), 203–221.
115. Ragonis, N., and Hazzan, O. (2009). Integrating a tutoring model into the training of prospective Computer Science teachers. *The Journal of Computers in Mathematics and Science Teaching* 28(3), 309–339.
116. Rubinstein, A., and Chor, B. (2014). Computational thinking in life science education. *PLOS Computational Biology* 10(11), 1–5.
117. 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 Journal on Educational Resources in Computing* 8(2), 5:1–13.
118. Schocken, S., Nisan, N., and Armoni, M. (2009). A synthesis course in hardware architecture, compilers, and software engineering. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education* (SIGCSE), 443–447.
119. Schwarz, S., and Ben-Ari, M. (2006). Why don't they do what we want them to do? In *Proceedings of the 18thAnnual Workshop of the Psychology of Programming Interest Group* (PPIG), 266–274.
120. Schwill, A. (1994). Fundamental ideas of computer science. *Bulletin-European Association for Theoretical Computer Science* 53, 274–295.
121. Sfard, A. (1991). On the dual nature of mathematical conceptions: Reflections on processes and objects as different sides of the same coin. *Educational Studies in Mathematics* 22, 1–36.
122. Shmallo, R., and Ragonis, N. (2020). What is "this"? Difficulties and misconceptions regard the "this" reference. *Journal of Education and Information Technologies* 26(1), 733–762.
123. Statter, D., and Armoni, M. (2017). Learning abstraction in computer science: a gender perspective. In *Proceedings of the 12th Workshop in Primary and Secondary Computing Education* (WiPSCE), 5–14.
124. Statter, D., and Armoni, M. (2020). Teaching Abstraction in Computer Science to 7th Grade Students. *ACM Transactions on Computing Education* 20(1), 8:1–37.
125. Stavy, R., and Tirosh, D. (2000). *How Students (mis-)Understand Science and Mathematics: Intuitive Rules*. Teachers College Press. New York, NY.
126. Stephenson, C., Gal-Ezer, J., Haberman, B., and Verno, A. (2005). *The New Educational Imperative: Improving High School Computer Science Education, Final report of the CSTA Curriculum Improvement Task Force*. ACM. New York, NY.

127. Stolin, Y., and Hazzan, O. (2007). *Students'* understanding of computer science soft ideas: the case of programming paradigm. *Inroads – the SIGCSE Bulletin* 39(2), 65–69.
128. Taub, R., Armoni, M., and Ben-Ari, M. (2014). Abstraction as a bridging concept between computer science and physics. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education* (WiPSCE), 16–19.
129. Teif, M., and Hazzan, O. (2006). Partonomy and taxonomy in object-oriented thinking: Junior high school students' perceptions of object-oriented basic concepts. *Inroads – the SIGCSE Bulletin* 38(4), 55–60.
130. Utting, I., Tew, A. E., McCracken, M. E., Thomas, L., Bouvier, D., Frye, R., Paterson, J., Caspersen, M., Ben-David Kolikant, Y., Sorva, J., and Wilusz, T. (2013). A fresh look at novice programmers' performance and their teachers' expectations. In *Proceedings of the ITiCSE Working Group Reports Conference on Innovation and Technology in Computer Science Education – Working Group Reports* (ITiCSE-WGR), 15–32.
131. Vahrenhold, J., Nardelli, E., Pereira, C., Berry, G., Caspersen, M. E., Gal-Ezer, J., Kölling, M., McGettrick, A., and Westermeier, M. (2017). *Informatics Education in Europe: Are We All in the Same Boat?* ACM. New York, NY.
132. Vilner, T., Zur, E., and Gal-Ezer, J. (2007). Fundamental concepts of CS1: procedural vs. object oriented paradigm – a case study. In *Proceedings of the 12th Annual ITiCSE Conference on Innovation and Technology in Computer Science Education*, 171–175.
133. Zur-Bargury, I., (2012). A new curriculum for junior-high in computer science. In *Proceedings of the 17th ACM Annual Conference on Innovation and Technology in Computer Science Education* (ITiCSE), 204–208.
134. Zur-Bargury, I., Pârv, B., and Lanzberg, D. (2013). A nationwide exam as a tool for improving a new curriculum. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education* (ITiCSE), 267–272.