Jl. of Computers in Mathematics and Science Teaching (2013) **32**(3), 265-284

On Teaching Abstraction in Computer Science to Novices

MICHAL ARMONI
Weizmann Institute of Science, Israel
michal.armoni@weizmann.ac.il

Abstraction is a key concept in computer science (CS), and one of the most fundamental ideas underlying its practice. However, teaching this soft concept to novices is a very difficult task, as has been emphasized by many computer science education (CSE) experts. This paper discusses this issue and suggests a general framework for teaching abstraction in CS to novices, a framework that would fit into most kinds of introductory courses. While this paper draws on some anecdotal evidence to support its claims, it is not an empirical work. Rather, it builds on the research literature and experience to outline some concrete rules that can contribute to teaching abstraction.

INTRODUCTION: ABSTRACTION AND ITS ROLE IN CS

Abstraction has been acknowledged by CS experts as a key concept in CS, a fundamental CS idea that underlies the work of a computer scientist. In his Turing award lecture Dijkstra noted that "the effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer" (Dijkstra, 1972; p. 864). According to Dijkstra, abstraction is "the only mental tool by means of which a very finite piece of reasoning can cover a myriad of cases" (p. 864). In the same spirit, Wing (2006) looked at abstraction as one of the characteristics of computational thinking since "computational thinking is using abstraction and decomposition when attacking a large complex task or designing a large complex system" (p.33). This theme is often reflected in educational decisions

concerning the teaching of CS at various levels. The task force chaired by Denning, which was commissioned to give a definition of CS and then propose a teaching paradigm based on this definition, described CS as a discipline characterized by three processes – theory, abstraction (modeling) and design (Denning et al., 1988). Subsequent task forces that have been asked to design CS curricula base their recommendations on this description of CS, and have included abstraction as a key concept on their curricula.

Abstraction falls nicely into Schwill's theoretical framework of fundamental CS ideas (Schwill, 1994), and satisfies his four criteria: the horizontal criterion of applicability in multiple ways in various areas of CS; the vertical criterion of the ability to be demonstrated and taught on every intellectual level; the criterion of time, i.e., clearly observed in the historical development of CS and of continuous relevance in the longer term; the criterion of sense, i.e., to relate to everyday language and thinking.

The beauty and strength of abstraction as a tool for simplification are rooted in the fact that it is a multi-level tool. Using abstraction means working on multiple levels of abstraction, and CS experts specifically relate to the skill of moving between levels of abstraction as a central skill for computer scientists. For example, Wing (2006) noted that "Thinking like a computer scientist means more than being able to program a computer. It requires thinking in multiple levels of abstraction" (p. 35). According to Knuth "they [natural computer scientists] are individuals who can rapidly change levels of abstraction, simultaneously seeing things "in the large" and "in the small"" (Knuth, in Hartmanis, 1994, p. 39). Dijkstra's appreciation of abstraction and the ability to move between levels of abstraction is repeatedly demonstrated in his writings. For example, in one of his notes (Dijkstra, 1975) he wrote

"[T]his oscillation between "the representation" and "what it stands for" is an intrinsic part of the programmer's game, of which he had better be aware! (It could be also this oscillation, which makes programming so difficult for people untrained to switching between levels.)" (p. 1).

Hartmanis, another Turing Award winner, referred to abstraction in his Turing Award lecture (Hartmanis, 1994) as follows:

"One of the defining characteristics of computer science is the immense difference in scale of the phenomena computer science deals with. From the individual bits of program and data in the computers to billions of operations per second on this information by the highly complex machines, their operating systems and the various languages in which the problems are described, the scale changes through many orders of magnitude" (p. 39).

These quotes show that there can be various kinds of levels of abstraction, depending on the context or point of view, though all range from "in the large" to "in the small', as Knuth put it: they can correspond to a certain scale of magnitude or quantity, a scale with a certain "physical" air, as suggested by Hartmanis. Levels of abstraction can also correspond to a scale of meaning, oscillating, as Dijkstra put it, from "how" (the representation) and "what" (it stands for). Aho and Ullman (1992) pointed out that "[C]omputer scientists … must create abstractions of real-world problems that can be represented and manipulated inside a computer" (p. 1). This also induces a scale of meaning; namely, a problem on a higher level, and a solution on a lower level.

This paper discusses abstraction in CS and focuses on the challenge of teaching abstraction to CS novices. It is organized as follows. Section 2 discusses the role of abstraction in CS education. Section 3 surveys work regarding the teaching of abstraction. Section 4 presents and illustrates our framework for teaching abstraction to CS novices. The concluding remarks appear in Section 5.

THE ROLE OF ABSTRACTION IN CS EDUCATION

Once the importance and significance of abstraction in CS has been established, the next question is whether abstraction is an acquired skill. Another look at the quotes above suggests that Knuth sees abstraction as a skill that is natural for some individuals who can be considered natural computer scientists. Dijkstra, on the other hand, used the verb "to train" hinting perhaps that with the right training people can easily switch between levels. These two points of views differ as to the role of abstraction in CS education; namely, should it be viewed as a precondition or as an outcome?

Abstraction as a Precondition for CS Education – Relevant Work

Following the trail pointed at by Knuth's point of view, there is a certain skill, the skill of abstraction, which is vital, a precondition for those intending to become computer scientists or to study computer science. This skill can be evaluated, and proper assessment tools would be important in the admission processes.

This perspective was explored by Bennedsen and Caspersen (2006, 2008). They looked for correlations between success on standard tests de-

signed to assess general abstraction ability and success in CS. In the first paper (Bennedsen & Caspersen, 2006), success in CS was operationalized as success in an object-first version of the first CS introductory course (CS1), and no correlation was found.

In the second paper (Bennedsen & Caspersen 2008), success in CS was operationalized as success in 10 mandatory courses of a CS undergraduate program (one math course and 9 CS courses). Again, no correlation was found, not even when the courses were grouped by type, or by their level of learning outcomes. However, in terms of individual courses, a correlation was found for the course dealing with algorithms and data structures, and a trend toward a correlation was found for the course on formal languages and computational models.

While these results might indicate that abstraction is not a necessary precondition for CS studies, the standard tests used by Caspersen and Bennedsen to assess *general* abstraction ability may not have grasped the notion of abstraction in CS.

To the best of our knowledge, this perspective has yet to yield significant results. There are no validated tools that assess abstraction ability in the context of CS, and no evidence for the necessity of abstraction ability as a precondition for CS education.

Abstraction as a Result of CS Education – Relevant Work

The alternative approach induced by Dijkstra's quote is that training computer scientists is also training to abstraction, that is, abstraction ability should not be seen as a precondition for a CS program, but rather as an outcome.

Several studies indicate that this outcome is difficult to achieve. Or-Bach and Lavy (2004) studied the abstraction skills of CS1 students, looking at abstraction in its narrow meaning in the context of object-oriented programming (OOP); namely working with abstract classes and polymorphism. They found that most students were unable to reach the desired level of abstraction in this context. Students' difficulties with abstraction in OOP were also reported by Sanders and Thomas (2007), for CS1 students, and by Turner, Quintana-Castillo, Pérez-Quiñones and Edwards (2008) for CS2 (the second CS introductory course) students. Haberman (2004) found that (high school and undergraduate) students felt less comfortable with an algorithm having a very high level of procedural abstraction than with algorithms that had lower levels of procedural abstraction. In another study,

Haberman, Averbuch, and Ginat (2005) showed that high school students had problems fully understanding an algorithm as an object. As discussed below this implies that these students perceived the notion of an algorithm on a lower level of abstraction. Aharoni (2000) investigated the thinking processes of undergraduate students taking a data structures course. His findings indicate that these students tended to think and reflect on data structures on a low level of abstraction – such as the level of implementation in a programming language.

Hazzan (2003) reported on students' tendency to reduce the level of abstraction when dealing with a relatively new concept. According to Hazzan, students' attempts to reduce the level of abstraction can be expressed in one of three ways:

1. Making a new concept more concrete by making it more familiar. Thus, if the students are already familiar with another concept that resembles the new concept to some extent, they tend to stretch the resemblance. They might (unconsciously) deduce that the new concept has some properties shared by the familiar concept. For example, when learning the regular operations of concatenation and power, students might tend to think of the familiar arithmetic operations of power and multiplication and wrongly deduce that $a^n b^n = (ab)^n$ where a and b are letters over some alphabet.
2. Perceiving a new concept as a process rather than as an object. This is a reflection of the process-object duality (Sfard, 1991). At a lower level a new concept is perceived as a process which acts on other objects. Only when a higher level of abstraction is reached can the new concept be grasped as an object in itself on which other objects can act. For example, at a lower level of abstraction a function is perceived as a process that acts on objects (e.g. on numbers). At a higher level of abstraction, a function is itself an object and thus at this level one can consider a function that acts on functions.
3. Reducing the complexity of the new concept, for example, by dealing only with a special case. For example, when students are introduced to sets they might tend to consider only finite sets.

The tendency to reduce the level of abstraction is doubly linked to the issue of training to abstraction. First, since students tend to work at lower levels of abstraction, they do not sufficiently practice or experience the oscillation between lower levels and higher levels of abstraction. Second, as a relatively new CS concept, abstraction itself might be processed only at

lower levels of abstraction (for example, at the level of implementation in a programming language, in the context of signatures of methods) and thus will not be fully understood and internalized.

On the basis of Hazzan's theory on the tendency to reduce the level of abstraction, Perrenet and his colleagues (Perrenet, Groote & Kaasenbrood, 2005; Perrenet & Kaasenbrood, 2006) investigated students' understandings of the concept of an algorithm. They defined a specific hierarchy of levels of abstraction:

- On the lowest level, the *Execution* level is an interpretation of an algorithm as a specific run on a specific concrete machine.
- At the next level, the *Program* level, an algorithm is a process; i.e., a program written in a specific programming language.
- At the next level, the *Object* level, an algorithm is an object, which is not associated to a specific programming language. Only at that level one can talk about complexity measures in terms of functions that depend on inputs.
- The highest level is the *Problem* level. At this level one is capable of considering a problem as an object with a life of its own, referring to attributes of the problem such as solvability or complexity. At this level one is also capable of dealing with a solution to a problem as a black box.

From now on, we refer to this hierarchy as the PGK-Hierarchy (Perrenet, Groote, Kaasenbrood). The Perrennet et al. studies (Perrenet et al., 2005; Perrenet & Kaasenbrood, 2006) found that most students were working at levels 2 and 3, and only a few students were working at levels 1 or 4. They also found that the level of abstraction increased as students progressed in their CS program. That is, during a certain year of studies the level generally increased with time, and the abstraction level of more senior students was higher than that of more junior students.

The PGK-hierarchy is a useful way to conceptualize students' abstraction abilities in various CS contexts. It can suit an introductory as well as more advanced CS courses. Reinterpreting the findings cited earlier, the Haberman et al. (2005) results indicate a partial understanding of level 3, the object level. Aharoni's findings (2000) can be interpreted as understanding the algorithmic components of data structures at level 2, the program level. In a joint work with Gal-Ezer and Hazzan (Armoni, Gal-ezer & Hazzan, 2006), we showed that even advanced undergraduate students (at the end of their second year) confuse the concepts of a problem and a solution for it. In terms of the PGK-hierarchy, these students were on level 3 rather

than on level 4. For example, when trying to reduce a certain algorithmic problem to another one, they reduced a problem to a solution, rather than to another problem (e.g. reducing to Bubblesort rather than to sorting, reducing to BFS rather than to the shortest path problem). We also saw that students had difficulties dealing with a solution to a problem as a black box. This again hints that they had not fully reached level 4.

TEACHING ABSTRACTION – RELEVANT WORK

As discussed in the previous section, research indicates that students' abstraction skills are not satisfactory. To address this issue, several CS educators have suggested methods to teach abstraction. Some of these are content-specific (e.g. Haberman, Shapiro, E., & Scherz, 2002) and others are more general. We will elaborate on the general methods (these publications do not include an empirical study component). Kramer (2003) discussed abstraction in the context of software engineering. He argued that abstraction is teachable, but should be taught indirectly, that is, not by lecturing about abstraction, but by doing abstraction. He recommended teaching enough mathematics, and teaching (formal) modeling and analysis, since active learning in these areas necessarily involves abstraction. In the same spirit, Bucci, Long, and Weide (2000) suggested modeling as a tool for teaching abstraction and exemplified their claims using a modeling language in the context of CS1 and CS2. Koppelman and van Dijk (2010) also suggested a strategy for teaching abstraction in an introductory course that essentially calls for exploiting procedural abstraction and the like as much as possible, and acknowledging it.

The component of explicit acknowledgment is probably crucial, as claimed by Hazzan, who has discussed the teaching of abstraction in a few publications (e.g. Hazzan 2008; Hazzan & Tomeyko, 2005). Unlike Kramer, who emphasized indirect teaching, Hazzan argued that on top of active learning, an explicit and reflective meta-component is essential when teaching abstraction. It is not enough for students to perform abstraction and move between levels of abstractions, they must also be aware of it.

Such an explicit meta-component is necessary because abstraction is a soft concept (Corder, 1990); that is, a concept that cannot be characterized by rigid formal rules and therefore cannot be taught through rigorous formalism. Soft concepts are general, and can be applied in different domains with respect to different kinds of problems. This means that coming up with a comprehensive definition that covers the range of varieties of the concept

but is concrete enough to grasp is very difficult. In addition, the concept must be illustrated with specific examples which can blur its generality.

These characteristics of abstraction also follow from Schwill's theory of CS fundamental ideas. A fundamental idea is always a *general* idea, as follows directly from the horizontal criterion of applicability in different domains and with respect to different kinds of problems. Demonstrating abstraction for a specific problem and in a specific context is necessary to explain it, but may not be sufficient to achieve *non-specific transfer* (Schwill, 1994) i.e., to induce significant learning that will enable students to use abstraction in other contexts. This difficulty is common to all CS fundamental ideas, and has been identified and reported in the literature for several other fundamental ideas (or soft concepts) such as recursion [e.g. DiCheva & Close, 1996; Kahney, 1983; Levi, 2001), encapsulation (e.g. Turner, Quintana-Castillo, Pérez_Quiñones, & Edwards, 2008), reduction (Armoni & Gal-Ezer, 2006; Armoni, Gal-ezer & Hazzan, 2006; Armoni 2008) and a programming paradigm (Stolin & Hazzan, 2007). The appropriate, most effective way to teach fundamental ideas, according to Bruner (1960), is in a spiral manner, which touches many contexts, at various levels, and includes a reflective meta-component. In the case of abstraction, this means that abstraction should be practiced across the CS curriculum, starting from CS1 and revisited again and again, in every context that lends itself to its use. Each use of abstraction must be explicitly acknowledged as such.

A CONCRETE FRAMEWORK FOR TEACHING ABSTRACTION IN CS TO NOVICES IN AN INTRODUCTORY COURSE

Rationale

Abstraction is not just a fundamental idea and a soft concept; it is also a habit of mind (Cuoco, Golenberg & Mark, 1997). In the context of mathematics, Cuoco, Golenberg and Mark called for organizing mathematics curricula around habits of mind, since it is important "to give students the tools they will need in order to use, understand and even make mathematics that does not yet exist" [p. 376]. The framework presented here takes up their recommendation, with the goal of teaching abstraction and developing CS-abstraction abilities. This implies:

1. Being able to differentiate levels of abstraction.
2. Being able to move freely and consciously between levels of abstraction.

3. Being able to decide, in every phase of the problem-solving process, at which level of abstraction it is appropriate to work.
4. Being able to use successive refinements of abstraction.

Since this framework was designed for novices, and was intended to be incorporated into an introductory course, it is built around the PGK-hierarchy. As noted above, this hierarchy is relevant in advanced courses as well, and thus encourages transfer to other contexts.

For the sake of clarity, in the remainder of this paper we rename level 3 of the PGK-hierarchy as the *algorithm* level (instead of the *object* level). This might seem awkward, since this hierarchy is about the concept of an algorithm, and hence all its levels deal with an algorithm. However, the object level grasps what is commonly and popularly referred to as an "algorithm" – a solution which is not presented in a specific programming language. Rephrasing the four sub-skills listed above for the specific case of the PGK-hierarchy, we get:

1. Being able to differentiate the levels of a problem, an algorithm, a program and an execution.
2. Being able to move freely and consciously between the levels of a problem, an algorithm, a program and an execution.
3. Being able to decide, in every phase of the problem-solving process, at which level of abstraction it is appropriate to work: the problem level, the algorithm level, the program level, or the execution level.
4. Being able to use successive refinements of abstraction, for example, starting with a high-level algorithm and then decreasing its level of abstraction while still remaining at the algorithm level (which is independent of any programming language), not going down to the program level.

In Section 2.1 we discussed the findings regarding students' difficulties with abstraction at specific levels of the hierarchy. We will now supply further evidence that points to students' difficulties with abstraction, and can be interpreted in the context of the PGK-hierarchy. This evidence is based on data collected in several other (yet unpublished) studies, with different research scopes and research goals. Therefore, the methodologies that guided the collection of that data (methodologies that were planned with other research goals in mind) are not detailed here and, and for the purpose of this article we will treat this evidence as anecdotal.

Case I: In an advanced phase of a national CS competition, 88 excellent high school CS students were given four algorithmic problems. These problems were difficult and challenging, and as is usually the case with such problems, the real challenge (and therefore, potentially the most time-consuming phase) was to analyze the problems, and gain insights into their key important characteristics. Once deep insight had been achieved a solution could be induced quite directly. The students could choose how they formulated their solutions – in pseudo-code or in a specific programming language. Of the 200 solutions (since the questions were very difficult most students did not answer all four questions) only 16 were formulated verbally in pseudo-code. All the other solutions were formulated in a programming language (mostly Java, but there were also some C# solutions), including fine syntax details and implementation of relevant methods. Only 6 out of the 88 students consistently formulated all their solutions in pseudo-code. Another 4 students gave some solutions in pseudo-code and some in a programming language.

Case II: 538 high school students who had completed a sequence of three introductory CS courses were asked to define the concept of algorithm. Most answers indicated alienation from the concept of a solution that is detached from a specific programming language. For example, "An algorithm is that thing that my teacher makes me write after I finish the program", or "an algorithm is that unnecessary thing that my teacher wants me to do".

Case III: 27 undergraduate CS students at various stages of their undergraduate program were given a list of 20 key CS concepts and were asked to draw a corresponding concept map. The average degree (number of touching links) of the concept of algorithm was 1.37, and the average degree of all concepts over all maps was 2.07. For comparison, the average degree of the concept of program was 3.4. In most concept maps the concept of algorithm was linked to the concepts of program or programming language. Some of the labels on these links indicated a relationship that placed the concept of algorithm on a higher level than that of program; e.g. "A program implements an algorithm", but some labels indicated a relationship that put the concept of program in the center, e.g. "An algorithm helps to create a program", "An algorithm is an initial idea for a program".

These data hint that students – at various levels and various ages – do not fully perceive level 3 of the PKG-hierarchy, and prefer to restrict their problem solving process to level 2 or lower. Our framework aims at remedying this tendency.

The Context in Which the Framework was Developed

Our framework was developed in a unique context. The CS group in our department initiated a research project (Meerbaum-Salant, Armoni & Ben-Ari, 2010; 2011) that focuses on introducing CS ideas and concepts to young (junior high) students, using the Scratch environment. Scratch (http://scratch.mit.edu), created by the Lifelong Kindergarten Group at the MIT Media Laboratory, is a media-rich visual system for novice programmers (Resnick et al., 2009), and is widely used in outreach projects for young students or for pre-introductory CS courses (CS0).

As part of our research project, corresponding teaching materials were developed. These included a textbook (Armoni & Ben-Ari, 2010) and a set of accompanying projects (available on http://stwww.weizmann.ac.il/g-cs/scratch/scratch_en.html). Our framework has guided the development of the textbook.

This is a challenging setting for introducing CS ideas in general and abstraction in particular. The textbook was developed with a very young audience in mind (grades 7-9, ages 13-15), with hardly any experience in CS, if at all. In addition, the currently available Scratch environment (version 1.4) does not support traditional procedural abstraction tools such as functions.

The Framework

Below we describe the framework itself, and provide several examples taken from the textbook. Note that the framework is general and is not dependent on any language or environment. Thus, while our examples sometimes utilize the Scratch environment, they can be easily adapted to any other context.

The first three guidelines are more general and apply to the other, more specific guidelines

First Guideline: Be Persistent and Precise

As CS experts, CS educators move freely and easily between levels of abstraction. They differentiate the levels, and they can also think simultaneously on more than one level. Thus, it is not uncommon for a CS expert to mention several levels of abstraction together, sometimes even in the same sentence, without a clear distinction between them. When teaching novices, instructors must pay attention to clearly distinguish between levels, and

always work at the appropriate level. For example, when working at Level 3, specific implementation issues should be strictly avoided and should be deferred until the discussion reaches Level 2. This may seem artificial and even burdensome, yet it is a crucial point. Since the natural tendency of students is to work at lower levels and to blur the distinction between the levels, they will naturally and unconsciously follow any implicit guidance that is consistent with this tendency. Only explicit, persistent and strict instruction that contradicts this tendency can in turn overcome it. To make the oscillation between levels natural it is necessary and essential to pass through this artificial and even cumbersome phase.

Second Guideline: Use Linguistic Components as an Aid to Distinguish Between Levels

Words can act as markers that assist students in remembering the level of abstraction they are currently working at. Lower levels of the PGK-hierarchy have specialized words inherently connected to them, and teachers should pay attention not to use these kinds of words when working at higher levels. For example, names of programming language constructs belong to level 2, and at higher levels it is preferable to use other words (e.g., in the context of Scratch, keep the use of "broadcast" to level 2, and at higher levels use other words, such as "inform"). Similarly, names of algorithmic constructs and abstract data types belong to level 3. This guideline will be revisited below, when referring to specific levels.

Third Guideline: Order from High to Low

Throughout our textbook, discussions at higher levels of the hierarchy always precede discussions at lower levels of the hierarchy. For example, as elaborated with respect to the fifth guideline, we first design a high level solution for a problem and only then discuss implementation details. As always, it is important to keep the boundaries clear cut, so that it is clear where a higher-level discussion ends and where a lower-level discussion begins.

Each chapter is concluded by a summary of the new concepts and terms that were introduced. The order guideline applies in this case as well: First, the summary mentions CS concepts, then come instructions in Scratch (that is, implementations of the new CS concepts in Scratch). Technical issues related to the Scratch Environment (for example, how to change the background of the stage) only appear at the end of the summary.

Fourth Guideline: Make Clear Distinction Between Level 4 and Lower Levels

As mentioned above, students – at all levels – have difficulties differentiating the problem level and the solution levels, so it is crucial to make a clear distinction between them. Our textbook is based on active learning and is thus designed around tasks. Every task has a clear title followed by a description of it, and it is very easy to see where the description of a task starts and where it ends. This description should not include any reasoning that already belongs to the solution (whether on a high or low level). It is sometimes desirable to give the students some directions or hints to help them solve a certain task, but these should not be given as part of the description of the task, but after it, with a clear heading.

For example, our textbook includes a soccer project. A sub-task of this project was given to the students as an exercise. The main sentence of this exercise could have been phrased as follows: *Expand the script of the referee so that he falls down (changes direction) when he is touched by the ball*. But the words in parentheses are actually part of the solution, hinting that we can make an actor (a sprite) in Scratch look like he is falling if we change his direction. If the instructor feels that such a hint is necessary it should be given after the task description: *Expand the script of the referee so that he falls down when he is touched by the ball. Hint: use direction change to simulate falling.*

Following the second guideline, it is always a good practice to phrase a task, or a problem (level 4), in a natural language, using general words that do not directly correspond to possible solution components (level 3) or to constructs of the programming language at hand (level 2). For example, it is better to phrase a problem in terms of "appearance" or "look" and not in terms of "costume", when Scratch is the programming environment. Or, when assigning a problem dealing with a sequence of numbers do not use the word "array" or the array notation x[i]. The guiding principle is that problems have an independent existence of their own. Specifically, problem descriptions should be readable and understandable, even to those that do not know how to solve them!

Fifth Guideline: Make a Clear Distinction Between the Algorithm Level (3) and the Program Level (2)

It is very important to start the solution process on a high level that does not involve any programming language constructs. In our textbook, due to the relatively young age of the students, we chose not to use the word "algorithm". We use the term "verbal description" to indicate a high level solution, disconnected from a specific programming language. After a task

is assigned, according to the order guideline it is first approached on a high level, using verbal descriptions as solution descriptions. Implementation details are only discussed later when these verbal descriptions are translated into instructions or scripts in Scratch, and corresponding blocks are used. In many cases the translation is left to be done as an exercise.

In our textbook CS concepts are learned when needed to solve a specific task. Thus, often a solution process involves the introduction of a new concept. Again, according to the order guideline this introduction is always done first on a conceptual level, and only later on the Scratch level (Level 2), in which a corresponding Scratch construct, and later a corresponding scratch block are presented. The introduction of each concept is followed by a framed short description of it. Different frames are used for conceptual descriptions (with their title always starting with the words "a new concept:"), and for programming-level descriptions (with their title always starting with words such as "a new instruction in Scratch:" or "a new action in Scratch:"). It is very important not to blur the distinction between the two kinds of descriptions. For example, when introducing the concept of an infinite run, the information *an infinite run can be stopped by clicking the red stop sign* does not belong to the conceptual level, and therefore should not be included in the conceptual frame, but rather in the Scratch frame.

Linguistic aids are very helpful when aiming at differentiating between levels 2 and 3. Scratch's syntax is relatively natural and a script in Scratch reads just like natural language. Nevertheless, just like any other programming language this syntax is made up of rigid and fixed words ("reserved words" is still an appropriate term in the context of some languages though Scratch is not one of them) are used. It is therefore advisable to limit the use of these words to level 2, and use other words and very natural, flexible and varying (but accurate and unambiguous) phrasing for Level 3. Here are two examples (which can be easily adapted to other programming languages):

- While on Level 2 the instruction *forever* is used (with the corresponding block 

), on Level 3 the phrasing *repeat again and again* can be used.
- The use of different, natural phrasing for level 3 can help emphasize the "what" of Level 3 as compared to the "how" of Level 2: a step in the verbal description can be *advance x to the next position in the row*, and a corresponding instruction in Level 2 can be *change x by 50*

Levels are also differentiated by the vocabulary used for meta-discussions. For example, on Level 2 we discuss our solutions using words such as

"instruction" and (on a somewhat lower sub-level of level 2) "block", while in level 3 we use "step" or "sub-task".

Sixth Guideline: Use Refinements of Level 3

Since many students tend to avoid Level 3 and jump directly to design in a programming language, it is recommended to enrich the Level 3 discussions, thus enlarging their relative volume. This can be done by using refinements of algorithmic solutions, thus again demonstrating levels of abstraction, this time as sub-levels of Level 3. When the designed solution is complex, we first give a coarse description, with a few sub-tasks obtained by a decomposition of the given problem. Then sub-tasks are refined (sometimes successively) to obtain fine-grained descriptions.

At a high level, sub-tasks might correspond to the general behavior of a certain sprite (Scratch actor) and will later evolve to a set of scripts. On lower levels a sub-task might correspond to a specific behavior of a sprite, and will later evolve to one script. On an even lower level, a sub-task might correspond to a compound step which will later evolve to a few Scratch instructions.

Since the current version of Scratch (1.4) has no built-in procedural abstraction, except for decomposition to Scripts (which are not like procedures since they run concurrently), following this guideline can serve to demonstrate and practice abstraction with a procedural flavor, emphasizing once more the difference between "what" and "how". For example, in one of our projects, there is a verbal description for a restaurant chef that includes the step: *Look for the place of the missing dish in the main course menu*. This step is later refined to two steps which describe how this search is done:

1. *Examine the first dish in the main course menu*
2. *Repeat until the dish you are currently examining is the missing one:*
   2.1. *Examine the next dish in the main course menu*

(Later this description is abstracted to obtain a general search pattern.) As the refinement process proceeds, this description will be changed to a more concrete one that includes variable manipulation (and can be directly translated to a Level-2 solution):

1. *Initialize the position variable to 1*
2. *Run repeatedly until the course in the place that is pointed at by the position variable is the missing course:*
   2.1 *Advance the Position Variable to Point at the Next Place in the Main Course Menu*

The refinement guideline can also be used to introduce and exemplify a major principle of abstraction: the black box principle. Sub-tasks can be refined independently and not necessarily by order of appearance. For example, if a certain solution consists of a sequence of sub-tasks, we might choose to refine a later sub-task before any of the preceding sub-tasks. This means that when refining the later sub-task we do not know yet exactly *how* the preceding sub-tasks are solved, but we can count on their outcomes, knowing *what* they are supposed to be.

Seventh guideline: make a clear distinction between the program level (2) and the execution level (1)

As noted above, research indicates that most students can go beyond Level 1 and do not stay on this very concrete level. That is, the transition from this level to higher ones is not expected to lead to difficulties. Nevertheless, consistent with the general guidelines, there is a clear cut between this level and the next. Actually, (except for the first chapter, which also includes explanations regarding the execution of projects), executions of projects are left for the exercises, and are used to test projects, recognize bugs and consequently improve the projects. This emphasizes that one does not design a solution on this level, and that after working on this level for testing purposes one can climb back up to higher levels, and while working on these higher levels, improve the solution using the results obtained at the lower level.

Eighth Guideline: Be Explicit, also Using Reflection

As noted above, when teaching fundamental ideas, explicit treatment is necessary to achieve non-specific transfer (Schwill 1994, Bruner 1960). This is in line with Hazzan's recommendations (2008) for teaching abstraction. The last chapter of our textbook is a reflective chapter, listing the major CS principles and methods that were introduced, demonstrated and practiced throughout the textbook. Five of the seven items in this list reflect on various facets of abstraction: gradual solution process using decomposition into sub-tasks, verbal descriptions and successive refinement of verbal descriptions, using algorithmic patterns (such as counting, accumulating and searching), and encapsulation of information.

Obviously, in applying the guideline of explicit treatment and reflection, teachers play a major role. Therefore, this issue is emphasized in our courses for in-service teachers on teaching CS concepts with Scratch.

CONCLUDING REMARKS

As a fundamental idea of CS, abstraction is an essential skill for anyone practicing CS. Inherent to being a fundamental idea, learning abstraction is not trivial. And again as a fundamental idea, effective teaching of abstraction must be spiral and explicit. That is, it must start from the very first CS course and be revisited in every course and every context that lends itself to it while acknowledging its use and reflecting on it.

The framework described in this paper can be used to teach abstraction in an introductory course. The pivot of this framework is the PGK-hierarchy which includes four abstraction levels of the concept of an algorithm. In a nutshell, throughout the teaching process, an explicit multi-level use of abstraction is necessary, with a systematic distinction between the four levels, using linguistic aids to emphasize the differentiation and ordering references to these levels from high to low.

It is important to note that the syllabus of the course is not altered. Rather, the key is to exploit or create opportunities in the regular teaching process of the course that naturally lend themselves to using and discussing abstraction.

We are currently planning a study that will examine the effectiveness of this framework in the context of an introductory CS course for junior high school students, using the Scratch environment.

Acknowledgements

Many thanks to Moti Ben-Ari for his very helpful comments and fruitful discussions. Many thanks to Esther Singer, for the linguistic editing.

References

Aharoni. D. (2000). Cogito, ergo sum! Cognitive processes of students dealing with data structures. In *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education*, Austin, TX, March 2000, H. Walker and S. Haller, Eds. ACM press, New-York, NY, 26-30.

Aho, A. V. and Ullman, J. D. (1992). *Foundations of Computer Science.* Computer Science Press, New York, NY.

Armoni, M. (2008). A (mostly) quantitative analysis of reductive solutions to algorithmic problems. *Journal on Educational Resources in Computing 8(4)*, 11:1-30.

Armoni, M. and Ben-Ari, M. (2010). *Computer Science Concepts in Scratch*. Weizmann Institute of Science, Rehovot, Israel (in Hebrew).

Armoni, M. and Gal-Ezer, J. (2006). Reduction – an abstract thinking pattern: the case of the computational models course. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, Houston, TX, March 2006, D. Baldwin, P. Tymann, S. Haller, and I. Russell, Eds. ACM press, New-York, NY, 389-393.

Armoni, M., Gal-Ezer, J., and Hazzan, O. (2006). Reductive thinking in computer science. *Computer Science Education 16(4)*, 281-301.

Bennedsen, J. and Caspersen, M. E. (2008). Abstraction ability as an indicator of success for learning computer science? In *Proceedings of the 4th International Workshop on Computing Education Research* (ICER), Sydney, Australia, September (2008), M. E. Caspersen, R. Lister, and M. Clancy, Eds. ACM press, New-York, NY, 15-25.

Bennedsen, J. and Caspersen, M. E. (2006). Abstraction ability as an indicator of success for learning object oriented programming? *Inroads – The ACM SIGCSE Bulletin 38(2)*, 39-43.

Bruner, J. S. (1960). *The Process of Education*. Harvard University Press, Cambridge, MA.

Bucci, P., Long, J. L., and Weide, B. W. (2001). Do we really teach abstraction? In *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education*, Charlotte, NC, February 2001, H. Walker, R. McCauley, J. Gersting, and I. Russell, Eds. ACM press, New-York, NY, 26-30.

Cuoco, A., Goldenberg, E. P., and Mark, J. (1996). Habits of mind: an organizing principle for mathematics curricula. *Journal of Mathematical Behavior 15(4)*, 375-402.

Corder, C. (1990). *Teaching Hard Teaching Soft: A Structured Approach to Planning and Running Effective Training Courses*. Gower, Aldershot, UK.

Denning, P. J., Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., and Young, P. R. (1989). Computing as a discipline. *Communications of the ACM 32(1)*, 9-23.

Dicheva, D. and Close, J. (1996). Mental models of recursion. *Journal of Educational Computing Research 14(1)*, 1-23.

Dijkstra, E. W. (1975). About robustness and the like, EWD 452. *The Archive of Dijkstra's Manuscripts*. Available in http://www.cs.utexas.edu/users/EWD/, retrieved May, 26, 2012.

Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM 15(10)*, 859-866.

Haberman, B. (2004). High-school students' attitudes regarding procedural abstraction. *Education and Information Technologies 9(2)*, 131-145.

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)*, Caparica, Portugal, June 2005, J. Cunha, W. Fleischman, V. Proulx, and J. Lourenço, Eds. ACM press, New-York, NY, 74-78.

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.

Hartmanis, J. (1994). Turing Award lecture on computational complexity and the nature of computer science. *Communications of the ACM 37(10)*, 37-43.

Hazzan, O. (2008). Reflections on teaching abstraction and other soft ideas. *Inroads – The ACM SIGCSE Bulletin 40(2)*, 40-43.

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.

Hazzan, O. and Tomayko, J. E. (2005). Reflection and abstraction in learning software engineering's human aspects. *Computer 38(6)*, 39-45.

Khaney, H. (1983). What do novice programmers know about recursion? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Boston, MA, December 1983, R. N. Smith, R. W. Pew, and A. Junda, Eds. ACM press, New-York, NY, 235-239.

Koppelman, H. and Van Dijk, B. (2010). Teaching abstraction in introductory courses. In *Proceedings of the 15th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, Ankara, Turkey, June 2010, R. Ayfer, J., Impagliazzo, and C. Laxer, Eds. ACM press, New-York, NY, 174-178.

Kramer, J. (2003). Abstraction – is it teachable? The devil is in the detail (Keynote address). In *Proceedings of the 16th Conference on Software Engineering Education and Training (CSEET)*, Madrid, Spain, March 2003, P. Knoke, A.M. Moreno, and M. Ryan, Eds. IEEE Press, Piscataway, NJ, 32.

Levi, D. (2001). Insights and conflicts in discussing recursion: a case study. *Computer Science Education 11(4)*, 305-322.

Meerbaum-Salant, O., Armoni, M., and Ben-Ari, M. (2011). Habits of programming in Scratch. In *Proceedings of the 16th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education* (*ITiCSE*), Darmstadt, Germany, June 2011, G. Rößling, T. Naps, and C. Spannagel, Eds. ACM press, New-York, NY, 168-172.

Meerbaum-Salant, O., Armoni, M., and Ben-Ari, M. (2010). Learning computer science concepts with Scratch. In *Proceedings of the 6th International Workshop on Computing Education Research* (*ICER*), Aarhus, Denmark, August 2010, M. E. Caspersen, M. Clancy, and K. Sanders, Eds. ACM press, New-York, NY, 69-76.

Or-Bach, R. and Lavy, I. (2004). Cognitive activities of abstraction in object orientation: an empirical study. *Inroads – The ACM SIGCSE Bulletin 36(2)*, 82-86.

Perrenet, J., Groote, J. F., and Kaasenbrood, 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)*, Caparica, Portugal, June 2005, J. Cunha, W. Fleischman, V. Proulx, and J. Lourenço, Eds. ACM press, New-York, NY, 64-68.

Perrenet, J. and Kaasenbrood, E. (2006). Levels of abstraction in students' understanding of the concept of algorithm: the qualitative perspective. In *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, Bologna, Italy, June 2006, R. Davoli, M. Goldweber, and P. Salomoni, Eds. ACM press, New-York, NY, 270-274.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., and Kafai, Y. (2009). Scratch: Programming for all. *Communications of the ACM 52(11)*, 60-67.

Sanders, K. and Thomas. L. (2007). Checklists for grading object-oriented CS1 programs: concepts and misconceptions. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, Dundee, Scotland, June 2007, J. Hughes, R. Peiris, and P. Tymann, Eds. ACM press, New-York, NY, 166-170.

Schwill, A. (1994). Fundamental ideas of computer science. *Bulletin of European Association for Theoretical Computer Science 53*, 274-295.

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.

Stolin, Y. and Hazzan, O. (2007). Students' understanding of computer science soft ideas: the case of programming paradigm. *Inroads – The ACM SIGCSE Bulletin 39(2)*, 65-69.

Turner, S. A., Quintana-Castillo, R., Pérez_Quiñones, M. A., and Edwards, S. H. (2008). Misunderstandings about object-oriented design: experiences using code reviews. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, Portland, OR, March 2008, J. D. Dougherty, S. Rodger, S. Fitzgerald and M. Guzdial, Eds. ACM press, New-York, NY, 97-101.

Wing, J. M. (2006). Computational Thinking. *Communications of the ACM 49(3)*, 33-35.