Issues in Informing Science and Information Technology Volume 7, 2010

So Different Though So Similar? – Or Vice Versa? Exploration of the Logic Programming and the Object-Oriented Programming Paradigms

Bruria Haberman
Holon Institute of Technology and Davidson Institute of Science Education, Israel

Noa Ragonis
Beit Berl and Technion, Israel

nthaber@wisemail.weizmann.ac.il **noarag@beitberl.ac.il**

Abstract

Computer science (CS) curricula are composed of various study modules, each of which focuses on particular contents, concepts, principles, and associated problem-solving methods. Developers of CS curricula recommend that students become acquainted with different programming paradigms in order to acquire alternative ways of computational thinking and various approaches for problem solving. In this paper, we illuminate two different though related paradigms: object-oriented programming and logic programming. We present and discuss the findings of a comparative study aimed at revealing similarities and dissimilarities between object oriented programming and logic programming in the context of problem-solving approaches and conclude with guidelines for instructional design of a study track that combines both paradigms.

Keywords: Computer science education, Computer science curriculum, paradigms, problem solving, object-oriented programming, logic programming, Java, Prolog.

Introduction

Computer science (CS) curricula are composed of various study modules. Although the body of knowledge representing CS is actually an integrated whole, students by necessity learn concepts, principles, and associated problem-solving methods in separate courses. According to Atman, Turns, and Mannering (1999), differentiating between distinct learning modules might cause an impoverished understanding of concepts and an inability to use them on demand. Thus, the study modules must be tied together, with an integrative focus on recurrent principles and concepts. The target is to obtain a coherent view of CS, as well as to foster the development of students' problem-solving skills.

Educators debate about the roles of programming paradigms in teaching CS (Gal-Ezer & Harel, 1999; Reinfelds, 1995; Stollin & Hazzan, 2007; White & Sivitanides, 2005). Studies have dealt with questions such as: What skills are required from the learners in each paradigm? Should a learner study more than one paradigm? What paradigm should be taught first? What happens when transferring from one paradigm to another? (Bergin, 2000; Bucci, Heym, Long, & Weide, 2002; Hansen, & Kristensen, 2008; Ragonis, & Ben-Ari, 2005a, 2005b). Like other researchers (Gal-Ezer & Harel, 1999; Reinfelds, 1995), the authors of this paper believe that students should become acquainted with different paradigms in order to acquire alternative ways of computational thinking (Wing, 2006) and various approaches for problem solving.

This paper illuminates two different, though strongly related paradigms: object-oriented programming (OOP) and logic programming (LP). In the next section we present some background on learning different paradigms and our experience in teaching both paradigms. In the third section we present the essence of each of the two paradigms. In the fourth section we compare the paradigms in the context of problem-solving approaches. We conclude with guidelines on the instructional design of a study track that combines both paradigms.

Background

Learning Different Paradigms

Paradigms differ in the way a given problem is analyzed, in the way a solution is designed, and in the way the designed solution is implemented (Detienne, 2001). Studies showed that students learning a new paradigm seem to show both positive and negative transfer effects from their prior learnt paradigms and programming experience. In particular, students may transfer problem-solving techniques from one programming approach to another. Studies investigated the transition between declarative and procedural paradigms and between procedural and object-oriented paradigms. For example, studies showed a negative transfer from procedural programming to LP of some concepts (e.g., recursion) and a positive transfer of the comprehension of computational processes (Carey, & Shepherd, 1988; Haberman, 2004). Other studies indicated significant difficulties regarding the transition from procedural programming to OOP (e.g., Ross, & Zhang, 1997). Such difficulties may be explained by differences in (a) problem-solving approaches anchored in the different paradigms, and (b) implementation methods. These and similar studies expressed the importance of instructors being aware of the characteristics of each paradigm taught to students, specifically the differences and similarities between them.

Authors' Experience in Teaching Logic Programming and Object-Oriented Programming

Teaching Logic Programming – For over 15 years, both authors have been involved in: (a) the development of curricula and suitable learning materials designed for teaching logic programming (LP) in Prolog environment (text books, teachers' guides, lab exercises, and collections of exemplary problems), (b) teacher training, and (c) formative assessment and research (Haberman, & Scherz, 2005; Haberman, Shapiro, & Scherz, 2002; Scherz, Haberman, Ragonis, & Shapiro, 1993).

The first author conducted an ongoing qualitative and quantitative study aimed at assessing various aspects of novices' problem-solving strategies and their use of abstract data types (ADTs) in LP (Haberman et al., 2002). The findings indicated that students applied various strategies of using ADTs, some of which diverged from the conceptual model and yielded incorrect products. For example, novices who initially learned and succeeded in using predefined ADT black boxes, had difficulties in transparently using black boxes after becoming acquainted with their implementation (Haberman et al., 2002). The findings also indicated that novices have difficulties in: (a) correctly mapping the relationship between the problem and its abstract model, and (b) differentiating between distinct abstraction levels and (c) linking the solution's different components, in various stages of the problem-solving process (Haberman, & Scherz, 2005).

The second author conducted a study on the teaching of expert systems (ES) to advanced high school students based on LP using Prolog. The study focused on students' conceptions and on stages of abstraction they encountered in learning expert systems (Scherz et al., 1993).

*Teaching Object Oriented P*rogramming - The second author developed learning materials for teaching object oriented programming (OOP) using the *objects-first* approach, and conducted a thorough study on the learning of OOP by novices. The uniqueness of the research was in the breadth and depth of its investigation into the concepts. The study findings were divided into four primary categories: class vs. object, instantiation and constructors, simple vs. composed classes, and program flow. In total, 58 conceptions and difficulties were identified. Most of them appeared with low frequency and characterized a particular period of learning, dissipating as the course progressed. At the end of the course, the students understood the basic principles of OOP. Main conclusions of the study were that the difficulties identified in the research do not imply that teaching OOP is inappropriate for novices. Apparently, the order in which concepts are presented and the kind of examples used are extremely important to assure good conceptualization of the basic principles of OOP (Ragonis, & Ben-Ari, 2002, 2005a, 2005b).

Challenges Addressed In This Paper

Many computer science education research papers address the question of how to teach novices distinct specific paradigms or distinct specific programming languages. However, only few papers present in-depth examinations of teaching a sequence that combines two paradigms. Specifically, we found no studies that compare teaching object-oriented programming to teaching logic programming, or that address the difficulties encountered by learners when transitioning between the two programming paradigms.

In this paper we present and discuss the findings of a comparative study which examined the similarities and dissimilarities of the strongly related paradigms: object-oriented programming and logic programming. Based on our vast experience teaching both paradigms to various populations and studying students' conceptualization of these paradigms, we believe that a combination of the two paradigms can contribute to coherent understanding of core CS concepts. Hence, we examined the similarities and dissimilarities between the paradigms. Such a study is essential in order to develop guidelines for teaching different paradigms in a combined study track.

Although LP and OOP might seem at first glance to be quite dissimilar, we reveal a meaningful similarity in their core principles. Bratko (1990) describes a basic relation between LP and OOP, as follows: “Prolog is a programming language [an implementation of LP] for symbolic, non-numeric computation. It is especially well suited for solving problems that involve objects and relations between objects” (p.4). The reference made in logic programming to objects and relations between objects, which are at the heart of OOP, as well as to other key concepts that are common to both paradigms (such as modularity, abstraction, and information hiding) support our rationale for examining the advantages and challenges of integrating both paradigms into a joint computer science study track.

The study presented here refers to fundamental issues of problem-solving processes. The more advanced issues (e.g., inheritance and polymorphism) require further exploration but can also be integrated into the approach presented here. We believe that by applying our approach, students' integrative view of paradigms can be cultivated, and students may be supported in constructing an assorted and meaningful problem-solving toolbox.

Basic Principles of Logic Programming and Object-Oriented Programming

The Logic Programming Paradigm

Logic programming is a declarative paradigm that is convenient for knowledge representation. “It is based on the belief that instead of the human learning to think in terms of the operations of a computer..., the computer should perform instructions that are easy for humans to provide” (Sterling & Shapiro, 1994, p.3).

(a) *Main characteristics:* Logic programming enables programmers to concentrate on declarative and abstract aspects of problem solving, and frees them from being concerned with the procedural details of the computational process. A logic program is a set of assumptions defining entities and relations between them; it describes the knowledge presented within the problem, but no explicit instructions on how to solve the problem are included.

(b) *Central concepts and principles*: Knowledge representation and formalization using formal logic, abstraction, modularity, and information hiding.

(c) *Model of computation*: The computation of a logic program represents the deduction of consequences of the program's statements. A program is executed by providing it with a query. The proof of a query is performed by a constructive computation mechanism that is based on applying the rule of universal logical modus ponens and on a generalized pattern-matching mechanism called unification.

(d) *Problem-solving approach*: Problem solving is based on the identification of entity types and the relations between them. The distinction between explicit knowledge - facts, and implicit knowledge - rules, enables to define chains of logical inference. Problem solving requires generalization (problem prototyping), abstraction, formalization in terms of logic statements, and concretization (linking abstract knowledge to concrete knowledge).

(e) *Programming language features*: The Prolog programming language is an implementation of LP. Prolog program contains logic statements (facts and rules) and not instructions for execution on a computer. Prolog is not statically type-checked, which may contribute to the flexibility of knowledge presentation; for example, a list in Prolog can include elements of different types. A major abstraction of the language is that assignment statements and explicit pointers are not used; instead, a generalized pattern-matching mechanism is used to construct and decompose data structures (Ben-Ari, 1996).

(f) *Advantages for teaching*: The syntax of logic programming resembles natural language and human reasoning, and frees the programmer from dealing with technical formalities common in other languages (Sterling, & Shapiro, 1994). This is why, in many countries, a mother tongue-based Prolog was developed for instructional purposes. LP is friendly and easy to use, and enables to concentrate on problem analysis and modeling. It is suitable for learning and practicing concepts such modularity, abstraction, information hiding, and recursion. It may enhance students' system-level perception in the context of declarative problem solving.

The Object Oriented Programming Paradigm

Object-oriented programming is based on the viewpoint that we live in a world of objects (e.g., cars, peoples, check boxes), so it seems natural to base our programming on objects. Each type of objects (class) can be combined in different projects. OOP enables to represent different relations between classes, and the most significant relations are *uses* and *inheritance*.

(a) *Main characteristics*: A class represents a type of entities and contains their common characteristics - attributes and methods. A class is a pattern for object creation. An object is a specific instance of a class; it has attributes and their values, and can behave according to the methods of its own class – a capsule. Objects interact with other objects.

(b) *Central concepts and principles*: Abstraction, encapsulation, modularity, data hiding, polymorphism, and inheritance.

(c) *Model of computation*: A program is a collection of objects that cooperate with each other in pre-defined ways to accomplish a common task. The objects are the active components. A solution to a problem is a *project* that contains classes, which present entity types and a main program class.

(d) *Problem-solving approach*: Problem solving is based on decomposing the problem by identifying the entity types involved. Each entity is described using attributes and methods that are implemented in a class. A main program class for executing the solution process is developed.

(e) *Programming language features*: Methods contain executive statements which, like in any procedural language, are assigned values according to parameters and can return values. The constructor method is a special method that creates an object. Methods can be static or non-static. OOP languages support the paradigm principles: creating objects, invoking methods on objects, polymorphism, and inheritance.

(f) *Advantages for teaching*: It is natural to teach key OO concepts with relation to real-world entity types. Using suitable environments (like BlueJ) enables to demonstrate those concepts from the beginning and thus simplify the understanding of the abstract concepts. OO methodology is suitable developing complicated systems that suit problem-solving processes in actual systems. The common OO languages (such as Java and C#) include a huge set of built-in classes that enable students to program meaningful and colorful projects shortly after the onset of learning (projects that include graphic or web elements).

The Problem Solving Processes

In this section, we compare the two paradigms with respect to: (a) the main approach to solving a problem, and (b) implementation techniques. We present similar problem-solving stages in both paradigms as well as dissimilar aspects of implementation.

Similar Problem Solving Issues

Despite the differences between the two paradigms, a great deal of similarity exists regarding central and fundamental issues of problem-solving process. Both paradigms suit for system-level problem solving. This means that artifacts developed using these paradigms mostly relate to compound structures, which consist of entities and relationships that are defined between them. The similar central issues refer to the following stages of problem (or system) analysis:

1. Decomposition – Identify the entity types involved and the relations between them. This phase concerns abstraction, and is central to both paradigms.

2. Define the characteristics of each entity type – Identify the attributes of each entity type and the required tasks related to it. In LP: For each entity type, identify the associated predicates: (a) the structure of the basic relation (fact), and (b) the required relations associated with the basic relation (rule heads). In OOP: For each entity type, identify the attributes – data members, and the required operations – the method's head (signature).

3. Modularity with respect to compound entity types – Each entity type is autonomous and can be used as an independent component. It can, however, be used also in a larger context, in relation to other entity types. In LP: Each entity type can be used as a package (functor) in a larger context, and so, to be operated with other entity types. In OOP: Classes can use other classes or inherit from other classes.

4. Modularity with respect to task definition – Each task is decomposed into subtasks. In LP: Decompose a relation into its most simple sub-relations and define a rule for each. In OOP: Decompose an operation into its most simple sub-tasks and write a different method for each.

Dissimilar Implementation Aspects

Despite the similarity in the problem-solving process that relates to system analysis, significant differences exist between the paradigms regarding implementation aspects:

1. Implementing a task – How is a task implemented to obtain a solution? In LP: A rule is implemented as a declarative logic statement. The components of a rule are logic relations (other rules) that are interrelated by logic operators. Negation and recursion are used (instead of loops). In OOP: A method is implemented as a sequence of operational statements such as assignments, conditional statements, or loop statements, and can use other methods.

2. Implementing a program – What is a program and how are different entities combined? In LP: A program is a set of logic statements, declarative facts and rules. Several programs (data bases) can be uploaded to the activated memory and consulted as a whole. In OOP: A project contains related classes that implement entity types and a program class within a main method to be executed.

3. Representation and creation of concrete entities – How and when are entities created? In LP: A logic program includes a description of concrete entities as part of its logical statements (facts). In OOP: Entities (objects) are created during execution using a constructor method.

4. The meaning of execution: What is "done"? In LP: A logic testing is activated – whether it succeeds or fails. In OOP: A sequence of statements is executed.

Table 1. Representation of *Song* entity in LP and OOP

Song information	song name, song performer, song length in seconds

Define the characteristics of each entity type	**In LP**: the entity characteristics are described as documentation:
	% song (Name, Performer, Length).
	
	In OOP: the entity characteristics are defined as a class:
	class Song {
	   String name;
	   String performer;
	   int length;
	   public Song(name, performer, length)
	}

Represent a specific entity (Object)	**In LP**: a specific entity is represented by a fact – before the execution path: *song (halleluiah, god-Band, 215).* **In OOP**: a specific entity is created using the constructor method – during the execution path: Song song1 = new Song(*"Halleluiah",* *"God-Band", 215*);

5. Program flow – The execution pass: How are tasks activated? In LP: A task is a *goal* (query) that is to be *satisfied*. To satisfy a goal means to demonstrate how the goal *logically follows* from the program's statements. A program is executed by providing it with a query. An answer to a query can be either positive or negative. In OOP: The program flow is the sequential activating of the program statements, starting from the main method. Statements include object creation and method invocation.

For example, Table 1 demonstrates differences in the representation and creation of entities in both paradigms.

Concluding Remarks

Teaching Both Paradigms

Educators argue that in our contemporary world we cannot teach the model of computation according to the traditional sequential, one-dimensional and static approach (Stein, 1996). Both logic programming and object-oriented programming paradigms are suitable for the formalization of compound multi-level systems; however, they differ in the domains the problems belong to. The logic programming paradigm is mostly suitable for the presentation of rule-based problems (e.g., games, consulting, decision support, and expert systems) whereas the object-oriented programming paradigm is mostly suitable for the development of large and compound "real world" software systems.

The educational advantage of logic programming is that the paradigm is suitable for beginners due to its simple syntax, use of natural reasoning, and formalization using natural language. Logic programming is a simple implementation of logics, which is the basis of computer science; hence, learning logic programming may improve students' logical reasoning and establish a good foundation of problem-solving skills. Nevertheless, current software design of real systems commonly uses object-oriented programming. The conceptual problem-solving approach is supported by the object-oriented programming languages, which provide typical embedded structures and reach libraries to support advanced programming. It is of general opinion that CS graduates should experience development of real systems during their studies. Based on the above arguments, we recommend that logic programming be taught before object-oriented programming. It is our opinion that teaching them in parallel, from the beginning, will cause novices a significant amount of confusion. Still, their combination after the main concepts in logic programming are taught, including vast implementation of different tasks, may be considered.

Proper linking between study modules may enhance students' understanding of recurrent concepts and principles. Moreover, it may support students in the construction of a rich and varied problem-solving toolbox, and develop their awareness of the need to choose a suitable paradigm and appropriate tools for solving a given problem. Specifically, learning different paradigms in a suitably combined manner may enhance problem-solving skills.

When considering combining the teaching of two paradigms and establishing proper linking between recurrent concepts, similarities and differences between the paradigms should be thoroughly examined. In the context of problem solving, the following main issues should be conceptualized and compared: (a) the way a problem is analyzed; (b) the way a problem is decomposed into sub-tasks; (c) the way sub-tasks are implemented in a language that relates to the paradigm; and (d) the way sub-tasks are connected to yield a whole solution. Although most of the fundamental problem-solving issues are significantly similar in both paradigms (stages a-b, see section *Similar problem-solving issues*), the implementation methods are quite different (stages c-d, see section *Dissimilar implementation aspects*). These differences are essential and not syntactical.

Implication for Instructional Design

Instructional design must support linking of distinct study modules, particularly those that relate to different programming paradigms. Proper linkage can be obtained by a fruitful collaboration between instructors, aimed to obtain an integrative and coherent view of computer science. Lack of direct and clear discussion about similarities and dissimilarities in all of the above mentioned aspects will rob students of the opportunity to gain deeper and more meaningful understanding of the valuable concepts or, even worse, cause confusion (as indicated by many research works). Such collaboration should involve:

- Mapping the relations between study modules;
- Identifying recurring concepts and similar principles;
- Investigating similarities and differences;
- Developing meaningful and integrative learning activities (based on the results of such a comparative study).

Specifically, in case of a serial combination of teaching different paradigms (as two distinct study modules, one after the other), when addressing each newly introduced concept in the current course, one must discuss and exemplify (through relevant learning activities) the similarities and dissimilarities of the same and similar concepts taught in the previous course. For example, in the context of teaching logic programming and object-oriented programming, when considering the implementation of an entity, the instructor should remind the students that in logic programming, we create a fact using a predicate for the entity-type, as well as arguments for the specific attributes of a specific entity. In object-oriented programming, on the other hand, we divide the same process into two passes. First, we define a class named on the entity-type with attributes. Then, only in the execution path, we create a specific object with its specific values.

The above-mentioned considerations should be taken into account in the instructional design process as well as in the implementation process.

Summary

One main goal of computer science education is to impart students with abstraction skills in various contexts. Specifically, students should be able to identify main ideas and knowledge structures beyond detailed context data of a given problem (Haberman, & Muller, 2008). Our suggestion to teach logic programming (first) and object-oriented programming in a coherent manner reflects more than "just" a suggestion to teach different paradigms. We suggest developing an instructional design that is based on continuous linking between the paradigms through the comparison of central concepts and problem-solving processes. Such a comparison: (a) enhances students' problem-solving skills that are common to both paradigms; (b) highlights essential differences between the implementation methods in each paradigm; and (c) minimizes misconceptions and mistakes typical of transition between paradigms. To support a structured comparison between concepts, appropriate learning activities should be developed, and class discussion must relate explicitly to similarities and differences between the paradigms.

Based on our vast experience in teaching and studying both paradigms, we believe that the approach presented here will serve as scaffolding for developing students' problem-solving skills, particularly the important ability to classify problems and choose the suitable paradigm for solving a given problem. Nevertheless, further investigation of the approach should be conducted through field implementation and testing.

References

Atman, C., Turns, J., & Mannering, F. (1999). Integrating knowledge across the engineering curriculum. *Proceedings of FIE 1999*, San Juan, PR. Session 13b7, 20-25.

Ben-Ari, M. (1996). *Understanding programming languages*. New York, NY: John Wiley and Sons.

Bergin, J. (2000). *Why procedural is the wrong first paradigm if OOP is the goal*. Retrieved September 2009 from http://csis.pace.edu/~bergin/papers/Whynotproceduralfirst.html

Bratko, I. (1990). *PROLOG programming for artificial intelligence* (2nd ed.). Wokingham, England: Addison Wesley.

Bucci, P., Heym, W., Long, T. J., & Weide, B. W. (2002). Algorithms and object-oriented programming: Bridging the gap. *ACM SIGCSE Bulletin*, *34*(1), 302-306.

Carey, T. T., & Shepherd, M. M. (1988). Towards empirical studies of programming in new paradigms. *Proceedings of the 16th Annual ACM conference on Computer Science*, 72-78.

Détienne, F. (2001). *Software design – Cognitive aspects*. (F. Bott, translator and editor). Heidelberg, UK: Springer.

Gal-Ezer, J., & Harel, D. (1999). Curriculum and course syllabi for a high-school CS program. *Journal of Computer Science Education*, *9*(2), 114-147.

Haberman, B. (2004). How learning logic programming affects recursion comprehension. *Computer Science Education*, *14*(1), 37-53.

Haberman, B., & Muller, O. (2008). Teaching abstraction to novices: Pattern-based and ADT-based problem-solving processes. *FIE'08*, T1A-1 – T1A-6.

Haberman, B., & Scherz, Z. (2005). Evolving boxes as flexible tools for teaching high-school students declarative and procedural aspects of logic programming. *Lecture Notes in Computer Science*, 3422, 156-165.

Haberman, B., Shapiro, E., & Scherz, Z. (2002). Are black-boxes transparent?- High school students' strategies of using abstract data types. *Journal of Educational Computing Research*, 411-436.

Hansen, M. R., & Kristensen, J. T. (2008). Experiences with functional programming in an introductory curriculum. *Lecture Notes in Computer Science*, 4821, 30-46. Berlin, Heidelberg: Springer-Verlag

Ragonis, N., & Ben-Ari, M. (2002). Teaching constructors: A difficult multiple choice. Workshop on Pedagogies and Tools for Learning Object-Oriented Concepts. *ECOOP2002*, Ma´laga, Spain. Retrieved September 2009 from http://prog.vub.ac.be/ecoop2002/ws03/acc_papers/Noa_Ragonis.pdf

Ragonis, N., & Ben-Ari, M. (2005a). A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education*, *15*(3), 203–221.

Ragonis, N., & Ben-Ari, M. (2005b). On understanding the static's and dynamics of object-oriented programs. *ACM SIGCSE Bulletin*, 37(1), 226–230.

Reinfelds, J. (1995). A three paradigm first course for CS majors. *Proceedings of SIGCSE 1995*, 223-227.

Ross, J. M., & Zhang, H. (1997). Structured programmers learning object-oriented programming: Cognitive considerations. *ACM SIGCHI Bulletin*, *29*(4), 93-99.

Scherz, Z., Haberman, B., Ragonis, N., & Shapiro, E. (1993). Expert systems by high school students in PROLOG environment. *Proceedings of the 7th International PEG Conference*, Edinburgh, Scotland, July 1993.

Stein, L. A. (1996). Interactive programming: Revolutionizing introductory computer science. *ACM Computing Surveys*, *28*(4), Article No. 103.

Sterling, L., & Shapiro, E. (1994). *The art of Prolog* (2nd ed.). Cambridge, MA: MIT Press.

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

White, G., & Sivitanides, M. (2005). Cognitive differences between procedural programming and object oriented programming. *Information Technology and Management, 6*(4), 333-350.

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

Biography

Bruria Haberman received her Ph.D. degree in Science Teaching from the Weizmann Institute of Science. She is currently a faculty member in the Department of Computer Science in the Holon Institute of Technology, teaching Logic Programming, Data Base Systems and Expert Systems. She is also a member of the computer science team in the Davidson Institute of Science Education in the Weizmann Institute of Science, where she leads the Computer Science, Academia & Industry educational program for talented high school students and their teachers. She is also a leading member of Machshava, the Israeli National Center for computer science teachers. She has developed learning materials for high school level in the areas of logic programming and artificial intelligence, and algorithmic patterns. She has developed academic programs for undergraduate level in computer science. Her primary research interests are declarative knowledge formalization in locic programming, computer science educational research, students' conceptualization of computer science, their problem solving approaches and abstarction skills as well as in-service teacher education.

Noa Ragonis received her Ph.D. degree in science teaching from the Weizmann Institute of Science. She is currently a faculty member in the Department of Computer Science of the School of Education in the Beit Berl College, teaching Algorithems, Object-Oriented Programming, Logic Programming and Learning and Teaching in Online Envirinments. She was the head of the Computer Science department for ten years, and serves now as the head of the curriculum comitty and academic advisor of the School of Education. She is also an adjucent senior lecturer in the Department of Education in Technology and Science in the Technion – Israel Institute of Technology, teaching Methods of Teaching Computer Science, Graph Theory and Computational Models. She has taught computer science in high schools for sixteen years. She has developed learning materials for high school level in the areas of Logic Programming, Expert Systems, Object Oriented programing and computational Models. She has developed academic programs for undergraduate level in computer science education. Her primary research interests are computer science educational research, students' conceptualization of object oriented programming, computer science teachers preparation programs as well as aspects of distance learning.