Informatics in Education, 2009, Vol. 8, No. 1, 3–16

Connectivity between Abstraction Layers in Declarative ADT-Based Problem-Solving Processes

Bruria HABERMAN
Department of Computer Science, Holon Institute of Technology
52 Golomb St., P.O.B 305, Holon 58102, Israel
Davidson Institute of Science Education, The Weizmann Institute of Science
Rehovot 76100 Israel
e-mail: bruria.haberman@weizmann.ac.il

Zahava SCHERZ
Department of Science Teaching, The Weizmann Institute of Science
Rehovot 76100 Israel
e-mail: zahava.scherz@weizmann.ac.il

Received: December 2008

Abstract. For over a decade, a declarative approach to problem solving based on the use of abstract data types (ADTs) has been taught to high-school students as part of the logic programming instructional unit. We conducted a study aimed at assessing students' problem-solving processes when utilizing ADTs. The findings indicated that students' strategies that diverged from the conceptual model often cause the students to develop incorrect programs. Specifically, students have difficulties in establishing correct mapping between the problem and its abstract model - the corresponding ADT, and in establishing proper connectivity between layers of abstraction related to different stages of the problem-solving processes (e.g., between distinct programming modules). These difficulties are apparently associated with general difficulties that novices encounter when learning programming, and with the cognitive load encountered when dealing with high levels of abstraction. With the intention to reduce student difficulties, we suggest using an instructional approach designed to gradually educate the students toward attaining proficiency as "problem solvers" through the use of integrative knowledge and autonomous problem-solving techniques. This approach should be further evaluated regarding its feasibility and applicability to reducing students' difficulties in dealing with abstraction processes.

Keywords: logic programming, abstraction, abstract data types, problem solving.

1. Introduction

We developed a two-stage "Logic Programming" (LP) course that was especially designed for high-school students who major in computer science (CS). One main goal of the course was to expose students to different aspects of logic programming and to enhance their problem-solving and design skills in the context of the LP paradigm. The

90-hour basic module covers the following topics: introduction to propositional logic and predicate logic implemented in Prolog, data base programming, compound data structures, recursion, lists, introduction to abstract data types (ADTs), and basic methods of problem solving and knowledge representation. The 60-hour advanced module introduces advanced methods of problem solving and knowledge representation, advanced generic abstract data types, and advanced programming techniques (Haberman *et al.*, 2002). Logic programming enables programmers to focus on the declarative and abstract aspects of problem solving, and usually liberates them from dealing with the procedural details of the computational process (Sterling and Shapiro, 1994). Abstract data types are considered as useful tools for CS problem solving and knowledge representation (Aho and Ullman, 1992). Since in LP the compound data structures are manipulated by hiding the procedural aspects and details of their implementation (Ben-Ari, 1995), it is convenient for implementing and utilizing abstract data types. Hence, it is a suitable programming environment for teaching the use of ADTs (Haberman *et al.*, 2002).

The concept of abstract data types, which is discussed in both modules of the LP course as a recurrent CS concept, is introduced to students as a *mathematical model with a set of operations* (Aho and Ullman, 1992). S*pecification* of an ADT is achieved by formally and verbally defining its use as a model as well as defining its operations. *Implementation* of an ADT in LP is achieved by formulating rules to define general predicates for each of the specified ADT operations. The actual implementation of an ADT is achieved by creating a black box. The *use* of an ADT for problem solving is achieved by defining problem predicates using transparently general predicates that are predefined and encapsulated in the ADT black box (Resnick *et al.*, 2000).

We developed an instructional approach to gradually introduce ADTs as flexible problem-solving and programming tools by using evolving programming boxes (Haberman and Scherz, 2005). We employed our instructional approach to teach problem-solving strategies and knowledge representation methods based on our ADT-based problem-solving conceptual model. We conducted an ongoing study aimed at assessing various aspects of students' use of ADTs in the Prolog environment (Haberman *et al.*, 2002; Haberman and Scherz, 2003; Haberman and Scherz, 2005). In this paper we focused on a particular facet of students' difficulties in utilizing ADTs, specifically related to the connectivity between distinct programming modules including ADT black boxes.

2. The ADT Conceptual Model

Abstraction is a major recurring concept in computer science and is considered an important tool in software development. It is the process of generalizing by reducing the information content of a concept, typically in order to retain only information that is relevant for a particular purpose. Abstraction is associated with (1) generalization of specific examples; (2) identification, extraction, and isolation of essential components; and (3) ignoring or holding back irrelevant details. In the context of problem solving, we refer to different abstraction levels that are associated with various stages of problem-solving processes. Studies show that experts and novices differ in their abstracting and generalizing

abilities (Haberman, 2004; Machanick, 1998; Ye and Salvendy, 1996). Gaining expertise in problem solving means being able to identify which abstraction level is suitable for a particular stage, with regard to problem analysis, solution design, and implementation (Haberman and Muller, 2008).

Use of abstract data types in problem solving and knowledge representation is a dominant component of our curriculum (Haberman *et al.*, 2002). Our conceptual model of utilizing ADTs in problem-solving processes and in developing computer programs is compatible with the formal definition of ADT as a formal CS concept (Aho and Ullman, 1992). The model is presented in the context of the LP declarative paradigm, but it may be generalized and adapted to different programming paradigms. The model includes distinct stages that relate to various levels of abstraction in problem-solving processes, and it actually entails transitions between abstraction levels:

(a) *Conceptualization:* (1) Comprehending the given problem; (2) identifying the main ideas, concepts, entities, and the relations among them; and (3) defining the main goals to be solved and queries to be addressed.

(b) *Generalization:* Distinguishing between the general definition of a problem and its concrete specific cases. Choosing problem-predicates that describe the general relations within the problem and the data-predicates that specify concrete cases of the problem.

(c) *Abstraction:* Expressing the concepts and relations in terms of abstract data types; deciding on a suitable ADT that characterizes of the general problem by choosing: (1) an appropriate formal model to describe the collection of objects defined by the problem, and (2) general ADT-predicates that represent operations defined in the formal model that are suitable to represent the relations between the objects, as defined in the problem. In this stage, one ignores the content of the general problem and relates to its abstract form.

(d) *Formalization:* Representing the concepts and the relations that were identified in the problem as a program; describing the general problem in terms of formal terminology by using ADT black boxes that were chosen to describe the problem. At this stage the problem-predicates are defined in the main program by transparently invoking general ADT-predicates (predefined in ADT black boxes).

(e) *Concretization:* Representing the concrete data (input) in terms of data-predicates. This can be done in the main program or in a distinct file. The concrete case of the problem is described by defining the problem-predicates in terms of data-predicates.

(f) *Testing:* Executing and debugging the developed program; assessing the program according to the specified requirements.

The ADT-based problem-solving process involves treating the problem at different levels of abstraction. The first three stages: conceptualization, generalization, and abstraction relate to comprehending and analyzing the problem; the three next stages: formalization, concretization, and testing relate to implementing the results of the analysis, and are aimed at achieving a correct working program that provides a suitable solution for the given problem.

The ADT-based problem-solving conceptual model relates to different phases of a problem: Initially, we relate to the given problem at its concrete level; next, we relate to the general problem, which is a generalization of the given problem, distinguishing between the specific data that related to the given problem and its general characteristics. Next, we map from the general problem to a completely context-free abstract model (ADT) that captures the logical interrelationships among the problem's entities. This stage involves the highest level of abstraction in the problem-solving process. From that point, we return to deal with the lower levels of abstraction related to the problem: first to the general problem to formalize the general features of the problem in terms of programming statements, and then to the concrete problem by linking up the general features with concrete specific data.

The conceptual problem-solving model described above may be used by the students both for solving small-scale problems during the course and for developing projects.

2.1. *Communicating with ADT Black Boxes*

In this section we focus on a particular facet of utilizing ADTs in problem solving and in developing programs (implemented in the Prolog language) – how the developed program should communicate with ADT black boxes. We demonstrate communication methods that are compatible with the ADT-based problem-solving conceptual model. In Section 3 we present students' strategies that diverge from the conceptual model and might cause students to develop incorrect programs.

Communication between the main program and an ADT black box is performed by establishing proper links in the following channels: (a) between specific data and data-predicates, (b) between problem-predicates and data predicates, and (c) between problem predicates and the corresponding ADT-predicates. The communication should be accomplished through the interfaces of the relevant modules/programs/ADT boxes.

2.1.1. *Linking by Casting into a Pattern*

The traditional way of formalizing the ADT formal model (i.e., the entities and the associated basic relationships among them) in a black box can be used to present the specific concrete data (the input) of a given problem. Students frequently described this method as "casting into a pattern" – a terminology that we adopted.

Linking to the List ADT: The linking is done by writing the specific data into a Prolog-style list data structure. For example, a list of students' names in a class will be presented in the following way:

```
% students_in_class (Class, List_of_students)
students_in_class (class_A, ['Abraham', 'Dan', 'Tamar', 'Ben']).
```

Linking to the Tree/Graph ADT: The linking is done by presenting the data in terms of general predicates that describe the nodes and the edges of a tree/graph. The specific data is added to the corresponding predicates' arguments. For example, data about classifying animals will be presented in the following way:

```
% edge (From, To)
edge (animal, mammal). edge (mammal, cat).
edge (mammal, lion).
% node (Node)
node(animal). node(mammal). node(lion). node(cat).
```

2.1.2. *Linking by Conversion*

Presenting specific data in this case is performed in terms of the data-predicates that are associated with the given problem and are syntactically different from the general ADT-predicates. The linkage is performed by defining a suitable rule that converts the problem-associated data presentation to an abstract ADT-based presentation, as illustrated in the following examples.

Linking to the List ADT: Here we demonstrate two alternative methods of list conversion.

(a) Linking by converting a presentation of a single-element to a presentation of a list-of-elements. This can be done by using the *findall/3* general predicate, which adds to a list the values of a specific variable (entity) that satisfies a given goal. For example, given facts related to each of the students in a class: *student(Class, Student_in_class)*, we create a list of all the students that belong to that class:

```
% students_in_class (Class, List_of_students)
students_in_class(Class, List_of_students):-
      class(Class),
      findall(Student, student(Class, Student), List_of_students).
```

(b) Linking by converting a presentation of successor-pairs to a presentation of a list-of-elements. This can be done by a recursive path-based accumulation of the list's elements. For example, given facts about children born in a familly *born_after(Familly, Child, Next_Born_Child)*, we create an ordered-by-birth list of children in the familly, starting with a specific child:

```
% children_in_family (Familly,Child, List_of_children)
children_in_family (Familly,Child, [Child, Next_Born_Child]):-
            born_after(Familly, Child, Next_Born_Child).

children_in_family(Familly,Child, [Child|Rest]):-
      born_after(Familly, Child, Next_Born_Child),
      children_in_family(Familly, Next_Born_Child, Rest).
```

Linking to the tree/graph ADT: The linking is obtained by defining conversion rules aimed at formalizing the general-predicates *node/1* and *edge/2* in terms of the corresponding data-predicates. For example, suppose that the data about the animals' classification will be presented by the data-*predicate as follows*:

```
% includes (Group1, Group1)
includes (animal, mammal). includes (mammal, cat).
includes(mammal, lion).

% animal_type(Animal_Type)
animal_type(animal). animal_type(mammal).
animal_type(lion). animal_type(cat).
```

The converting rules will be defined as follows:

```
node (Node):- animal_type(Node).
edge (Group1, Group2):- includes(Group1, Group2).
```

2.2. *An example – Biblical Genealogy*

The following example shows an ADT-based problem-solving process that is compatible with the previously described conceptual model.

The problem: Given the biblical genealogy, we are interested in retrieving all the male ancestors of a specific person (e.g., Jacob).

The problem-solving process: Here we demonstrate the analysis, reasoning, and decisions that are employed in each stage of the problem-solving process.

(a) *Conceptualization*: The entities identified in the problem are persons. There are *father_of* /,2 and *mother_of* / 2 basic relationships between persons. We are supposed to retrieve the list of all the ancestors of a given person, starting with the first ancestor (the dominating father).

(b) *Generalization:* The following basic relationships: *person(X)* (X is the name of a person who belongs to the family), and *father_of(X,Y)* and *mother_of(X,Y)* (X is a parent of Y) are used to present concrete data, and explicitly distinguish between the biblical family and other families; hence, they are classified as data-predicates. In contrast, the definition of *ancestors(X,Y)* (Y is a list of the male ancestors of X) is general and is suitable for any family, and is independent of specific data; hence, ancestor should be classified as a general problem-predicate. The identified predicates should be declared in the interface of the intended program.

(c) *Abstraction:* the graph ADT is an appropriate formal model used to describe the collection of objects (i.e., persons) defined by the problem, and the *path(R,X,Y)* and *root(R)* graph-predicates are suitable for defining the *ancestors(X,Y)* problem-predicate in the following manner: Y is the list of male ancestors of X, if R is a root of the genealogy, and Y is the list of nodes starting from R and ending in X.

(d) *Formalization*: The problem-predicate *ancestors(X,Y)* is defined in terms of the *path(X,Y,Z)* and *root(X)* graph-predicates that are predefined in the graph black box. The general ADT-predicates should be transparently invoked:

$$ancestors(X,Y) : -root(R), path(R,X,Y).$$

(e) *Concretization*: In this stage we present the concrete data in terms of data-predicates *person(X)*, *father_of(X,Y)* and *mother_of(X,Y)* as facts in a Prolog program. The concrete case of the problem is described by defining the problem-predicates in terms of data-predicates. In this example this is done implicitly by linking between the main program and the black box. Here the linkage is done between the *person(X)* and *father_of(X,Y)* data-predicates and the corresponding general graph-predicates *node (Node)* and *edge(From_Node, To_Node)* based on the following assertions: X is a node if it is a person; there is an edge from X to Y if X is the father of Y.

Queries (goals)
?- ancestors('Jacob', List_of_Ancestors).

A program that describes the problem

The Interface

Data Predicates
% person(Person)
% father_of(Father, Child)
% mother_of(Mother, Child)

General Problem Predicates
% ancestors(X,Y)

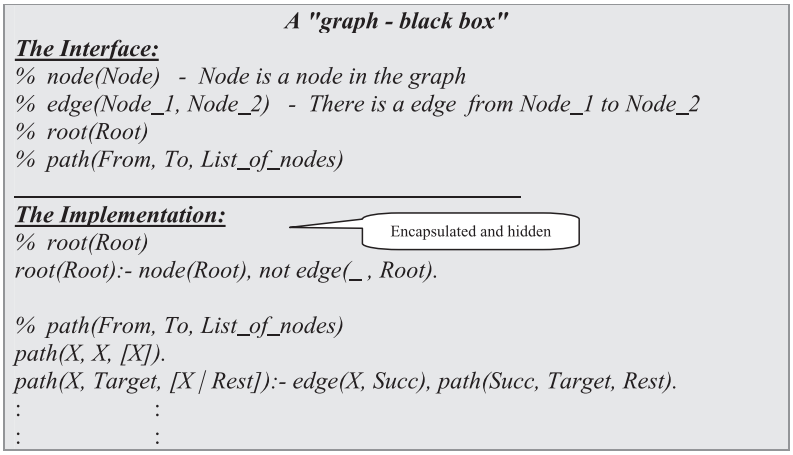
Formalization

Data Predicates
% person(Person)
person('Abraham'). person('Sarah'). person('Issac'). person('Jacob').
% father_of(Father, Child)
father_of('Abraham', 'Issac').
% mother_of(Mother, Child)
mother_of('Sarah', 'Issac').

General Problem Predicates
% ancestors(X,Y)
ancestors(X,Y):- root(R) , path(R,X,Y).

Communicating with the black box

node(Node):- person(Node).
edge(From_Node, To_Node):- father_of(From_Node, To_Node).



3. Students' Difficulties

Students attending the LP course were introduced to the ADT-based problem-solving conceptual model (described in Section 2). We conducted an ongoing study aimed at assessing various aspects of students' use of ADTs in the Prolog environment (Haberman *et al.*, 2002; Haberman and Scherz, 2003; Haberman and Scherz, 2005). We found that students adopted various strategies for using ADTs, some of which were compatible with the ADT-based problem-solving conceptual model. Other students improvised alternative strategies, which indicated that their conception of ADT did not match the formal CS definition. Nevertheless, the use of ADTs for problem solving and knowledge representation helped many students develop correct programs regardless of the strategies they used (Haberman *et al.*, 2002).

Here we discuss students' difficulties related to incorrect linking between the product's components, in various stages of the problem-solving process, which might cause the students to develop incorrect/non-working programs.

Our study revealed that most novices prefer *linking by casting to a pattern*, but a few students managed to successfully perform *linking by conversion*.

Often, even though students correctly identify the appropriate ADT for a given problem, they fail to correctly use the corresponding ADT-black box; more specifically, they fail to establish proper links between various components at the abstract level (the space problem and the corresponding ADT operations), or at the programming level (i.e., connectivity between specific data, data-predicates, problem-predicates, and the corresponding ADT-predicates). Specifically, we identified the following students' difficulties (and linked the difficulties to corresponding stages of the ADT conceptual model):

1. **Incomplete abstraction.** This refers to missing mapping between problem predicates and the corresponding ADT-predicates (a stage related to the conceptual model: (c)).
2. **Missing linkage to an ADT-black box**. Students avoided linkage between the data-predicate and the corresponding ADT-predicates. Interviews with students re-

vealed that they sometimes misleadingly assume that somehow the connection between the predicates automatically occurs owing to loading both files of the main program and the ADT-black box (a stage related to the conceptual model: (d)).

3. **Linking to an incorrect ADT-black box.** Sometimes the students use general-predicates of a specific ADT-black box, but they perform linking to another black box. For example, students use set-predicates, but perform linking to a list-black box (stages related to the conceptual model: (c), (d)).
4. **Incorrect linking**. Sometimes, even though the students refer to the suitable ADT-black box and try to perform a link to that black box, they fail to do it correctly. They often believe that the casting of specific data (input) into data-predicates guarantees accessibility to the data when posing a query, and therefore they might skip linking to the black-box, or to the problem-predicates. Another incorrect linking occurs when students perform seemingly (but incorrect) casting directly to the invoked ADT-predicate using a functional-based syntax (stages related to the conceptual model: (d), (e)).
5. **Difficulties owing to dealing simultaneously with several levels of abstraction**. Generally, students experience difficulties in moving between abstraction levels during a problem-solving process. Sometimes they avoid mapping from the problem directly to a generic ADT, and they define a mediator-problem-based ADT aimed at describing the general problem (difficulties of this type relate to stages (b), (c), (d), and (e) of the conceptual model).

For example, correct formalization of the problem-predicate *number_of_ students _in_class/2* according to this approach will consist of a three-level linking:

```
Linking by casting into a pattern - to the List ADT
% students_in_class (Class, List_of_students)
students_in_class (class_A, ['Abraham', 'Dan', 'Roy', 'Ben']).
Linking between a problem-predicate, a data-predicate, and a
mediator-problem-based ADT-predicate
% number_of_ students_in_class (Class, Num_of_students)
number_of_students_in_class (Class, Num_of_students):-
        students_in_class (Class, List_of_students),
        num_of_students_in_a_list(Num_of_students, List_of_students).
Linking between a mediator-problem-based ADT-predicate and
a generic ADT-predicate
% num_of_ students_in_a_list (Num_of_students, List_of_students)
num_of_students_in_list (Num_of_students, List_of_students):-
        num_of_items_in_a_list(Num_of_students, List_of_students).
```

For some students this might cause problems in linking between the specific data and the problem-predicates. For example, students correctly perform the linkage only when posing queries regarding the program, and avoid performing links in the program. This results in a program in which the generality of the problem's solution is usually reduced.

The results previously described indicate that students have difficulties in establishing correct mapping between the problem and its abstract (context-free) model–the corresponding ADT, and in establishing proper communication between specific corresponding programming modules. These difficulties are apparently associated with difficulties often encountered by a novice in learning to program in Prolog (Resnick *et al.*, 2000; Scherz and Haberman, 2005), and with the cognitive load required to write a program (Pain and Bundy, 1985) especially when dealing with high levels of abstractions (Haberman and Scherz, 2005).

4. The Instructional Approach

Studies show that experts and novices differ in their abstracting, and generalizing abilities (Haberman, 2004; Machanick, 1998; Ye and Salvendy, 1996). Gaining expertise in problem solving means being able, for example, to identify which abstraction level is suitable for a particular stage, with regard to problem analysis, solution design, and implementation. Hence, it is important that instructional design be oriented towards constructing an integrative, coherent perception of abstraction. Students need to develop proficiency in choosing the most suitable abstraction tool for solving a given problem and, furthermore, in identifying which tasks should be performed when dealing with different abstraction layers during various stages of solving a problem (Haberman and Muller, 2008). The teaching-learning process should be designed to gradually educate the students toward attaining proficiency as "problem solvers" through the use of integrated knowledge and autonomous problem-solving strategies.

In this section we briefly describe an instructional approach that we developed for the purpose of gradually introducing ADTs as flexible problem-solving and programming tools using evolving programming boxes (a detailed discussion of the instructional approach and its implications appears in (Haberman and Scherz, 2005).

We recommend that the ADT concept be gradually presented in the following consecutive stages (Haberman and Scherz, 2005):

Stage 1 – Acquaintance with given specifications of ADTs: Initially students become acquainted with the specification of abstract data types (e.g., lists, sets, trees, and graphs). Suitable examples of concrete problems should be used to illustrate the presented ADTs.

Stage 2 – Use of ADTs to solve a given problem: Next, students should practice how to choose ADTs to solve a given problem. For example, students should be able to determine that the *tree*-ADT is the most suitable one to present the family parenthood relationships between the females (or males), whereas the *graph*-ADT should be used to present relationships between all the family members (without referring to a specific gender).

Stage 3 – Use of ADT black boxes in programming: At this stage students should practice using predefined ADT black boxes to write computer programs that solve given problems. Specifically, students are taught to define problem predicates by transparently

invoking predefined general ADT-predicates. We emphasize the following aspects: (a) use of a black box is independent of its implementation and therefore it does not require becoming acquainted with the implementation details; (b) use of a black box binds to its interface.

Stage 4 – Specification of new ADTs: At this stage the student plays the role of a consumer who specifies and orders a new ADT black box from his teacher. The teacher implements the required ADT according to the student's specifications in terms of a black box, which is then used by the student to write his program.

Stage 5 – Acquaintance with the implementation of predefined ADT boxes: After students became familiar with the specifications and the use of ADTs, we suggest that they gradually learn how to implement an ADT according to its specifications. Initially, students become acquainted with the implementation of familiar ADTs. At this point the black boxes that have been transparently used in the previous stage become unfolded, i.e., the code within the black box is no longer hidden. Actually, at this point the black box becomes visible, however, only as *read-only* boxes, and the students perform operations such as reading the code, running the code, and following up its execution in order to understand "how it works".

Stage 6 – Manipulation of predefined ADT boxes: At this stage the *read-only* boxes becomes "more" accessible in the sense that their code can also be modified. Here students learn advanced programming techniques and efficiency aspects, and practice code debugging, code modification, and writing new code from scratch.

Stage 7 – Implementation of new ADTs: After becoming acquainted with the implementation of predefined ADT boxes, the students learn how to implement new ADT boxes according to a defined specification. At this stage they eventually become independent of the teacher in terms of supplying built-in programming tools.

Stage 8 – Knowledge integration and autonomous problem solving: At this stage students make a significant step toward attaining proficiency, and they practice solving advanced and complex problems. The students employ ADTs to solve a given problem in the following process: They try to determine familiar ADTs suited for the given problem and use the relevant predefined ADT black boxes. When the predefined ADTs do not suit their needs, they specify new ADTs from scratch or modify the specification of other ADTs, implement them in terms of black boxes, and then use them to develop their programs. Moreover, the students start acting like autonomous developers, by reusing their own tools, but on the other hand, they experience sharing tools with peers and reuse others' tools.

Such instructional approaches that gradually introduce abstract data types as flexible problem-solving tools may reduce students' difficulties in their transition between abstraction layers when solving problems, and in constructing appropriate linking between programming modules.

5. Concluding Remarks

Students need to develop proficiency in choosing the most suitable abstraction tool for solving a given problem and, furthermore, in identifying which tasks should be performed when dealing with different abstraction layers during various stages of solving a problem. Adequate learning of abstraction by novices may enhance their problem-solving abilities (Haberman and Muller, 2008). For example, reflection on the process, estimation of the product's quality, and reusability in future tasks need to be part of the routine of learning when utilizing abstraction tools (Hazzan and Kramer, 2006).

In this paper we demonstrated how evolving ADT boxes can be employed to teach an ADT-based problem-solving approach in the logic programming paradigm. We believe that the suggested instructional approach can be adopted to enhance problem-solving techniques in any programming paradigm, and that it can also be used to guide the students toward achieving proficiency in programming based on abstraction and reuse of code.

We recommend that the instructional approach be employed while the students are provided with an appropriate learning environment that promotes learning processes. Examples that provide scaffolding should be used to present the activities associated with each stage of the model. Moreover, appropriate exercises as well as activities that support the approach should be developed to motivate students to use black boxes transparently, reuse code provided by others, modify code, and use appropriate ADTs to solve given problems. In order to reduce students' difficulties, teachers should be aware of specific difficulties that occur at each stage of the learning process; specifically, they should identify the "weak links" and the "missing links" in the students' problem-solving strategies, such as the ones presented here. In addition, learning and instructional activities should be organized in such a manner as to minimize the cognitive load imposed upon the students when they are required to develop a program. One way that this can be achieved is to coach the students to organize their programs hierarchically and modularly (Scherz and Haberman, 2005; Sterling and Shapiro, 1994). In order to foster integrative knowledge, we recommend that students continue, at each stage of learning, to practice and meaningfully utilize the tools and the methods that they have previously acquired (Haberman and Scherz, 2005). Moreover, encouraging students to reflect on problem-solving processes, such as moving between abstraction layers, may be highly beneficial (Haberman and Muller, 2008).

The instructional approach presented here should be further evaluated regarding its feasibility and applicability to reducing students' difficulties when dealing with different abstraction layers during problem-solving processes.

References

Aho, A.V. and Ullman, J.D. (1992). *Foundations of Computer Science*. W.H. Freeman and Company.

Ben-Ari, M. (1995). *Understanding Programming Languages*. John Wiley.

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), 411–236.

Haberman, B. and Scherz, Z. (2003). Abstract data types as tools for project development – High school students' views. *Journal of Computer Science Education*, online, January 2003. Available: `http://iste.org/sigcs/community/jcseonline/`

Haberman, B. (2004). High-school students' attitudes regarding procedural abstraction. *Education and Information Technologies*, Special issue devoted to recent research projects of secondary informatics education, **9**(2), 131–145.

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

Haberman, B. and Muller, O. (2008). Teaching abstraction to novices in the course of pattern-based and ADT-based problem solving processes. In *Proc. of 2008 Frontiers in Education Conf.*, Saratoga Springs, New York, USA, T1A [1-6].

Hazzan, O. and Kramer, J. (2006). Abstraction in computer science & software engineering: A pedagogical perspective. `http://edu.technion.ac.il/Faculty/OritH/HomePage/FrontierColumns/OritHazzan_SystemDesigFrontier_Column5.pdf` [accessed October 15, 2007].

Kiczales, G. (1994). Why are black boxes so hard to reuse? Invited talk. In *OOPSLA'94*. Available: `http://www.parc.xerox.com/spl/projects/oi/towards-talk/transcript.html`

Machanick, P. (1998). The abstraction-first approach to data abstraction and algorithms. *Computers & Education*, **31**, 135–150.

Newell, A. and Simon, H.A. (1972). *Human Problem-Solving*. Prentice-Hall, New York.

Pain, H. and Bundy, A. (1985), What stories should we tell novice Prolog programmers. In R. Lawley (Ed.), *The Artificial Intelligence Programming Environments Book*, John Wiley.

Resnick, M., Berg, R. and Eisenberg, M. (2000), Beyond black boxes: bringing transparency and aesthetics back to scientific investigation. *Journal of the Learning Sciences*, **9**(1), 7–30.

Scherz, Z., Goldberg, D. and Fund, Z. (1990). Cognitive implications of learning Prolog – mistakes and misconceptions. *Journal of Educational Computing Research*, **6**(1), 89–110.

Scherz, Z. and Haberman, B. (2005). Mini-projects development in computer science – Students' use of organization tools. *Informatics in Education*, **4**, 307–319.

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

Ye, N., and Salvendy, G. (1996). Expert-novice knowledge of computer programming at different levels of abstraction. *Ergonomics*, **39**(3), 461–481.

B. Haberman received her PhD degree in science teaching from the Weizmann Institute of Science in 1999. She is currently a senior teacher in the Department of Computer Science in the Holon Institute of Technology. She leads the "Computer Science, Academia and Industry" educational program for talented high school students at the Davidson Institute of Science Education at the Weizmann Institute of Science, and is a leading member of Machshava – the Israel National Center for high school computer science teachers. She has developed learning materials for high school level in the areas of logic programming and artificial intelligence, abstract data types and algorithmic patterns. She has developed academic programs for undergraduate level in computer science. Her research has focused on computer science educational research, problem solving, students' conceptualization of computer science, as well as on in-service teacher education.

Z. Scherz has a MSc in biophysics, and a PhD in science education. Her postdoctoral research was performed at the University of Washington's College of Education. In 1984, she joined the staff of the Department of Science Teaching at the Weizmann Institute of Science, where she has led the logic programming in education group, and currently heads the chemistry and the scientific communication teams at the junior high level. She has written many learning materials for the junior high and high school levels in the areas of logic programming, artificial intelligence, science and technology and high order skills. Her research has focused on student conceptualization of computer science and scientific principles, on their learning of high order skills, as well as on the professional development of leading teachers.

Abstrakcijos sluoksnių sąryšiai sprendžiant abstrakčiųjų duomenų tipų uždavinius deklaratyviais metodais

Bruria HABERMAN, Zahava SCHERZ

Jau daugiau nei dešimtmetį aukštųjų mokyklų studentai yra mokomi deklaratyvių uždavinių sprendimo metodų, remiantis abstrakčiaisiais duomenų tipais (ADT). Toks mokymas buvo įtrauktas į loginio programavimo kurso turinį. Buvo atliktas tyrimas, kuriuo siekiama įvertinti, kaip studentai sprendžia uždavinius, naudodami ADT. Tyrimo rezultatai rodo, kad studentai, kurių pasirinktos strategijos buvo nutolusios nuo koncepcinio modelio, dažnai sukurdavo neteisingas programas. Daugiausiai sunkumų studentai turėjo susiedami uždavinį su jo abstrakčiu modeliu ir nustatydami ryšį tarp abstrakčių sluoksnių, susijusių su uždavinio sprendimo etapais (pvz., tarp atskirų programavimo modulių). Šie sunkumai yra akivaizdžiai susiję su bendro pobūdžio sunkumus, su kuriais susiduria naujokai, besimokantys programavimo, ir su pažintiniu krūviu, kylančiu dirbant su aukšto lygio abstrakcijomis. Siekiant sumažinti studentams iškylančius sunkumus, rekomenduojama naudoti mokymo metodas, skirtus palaipsniui mokyti studentus siekti aukštesnės uždavinių sprendimo kvalifikacijos, naudojantis integruotomis žiniomis ir autonominiais uždavinių sprendimo būdais. Šis požiūris turėtų būti toliau analizuojamas, atsižvelgus į jo pagrįstumą ir taikymą siekiant sumažinti studentams iškylančius sunkumus, susijusius su abstrakcijos procesais.