INFORMATICA, 2008, Vol. 19, No. 1, 17–30

Formal and Practical Aspects of Implementing Abstract Data Types in the Prolog Instruction

Bruria HABERMAN
Department of Computer Science, Holon Institute of Technology and
Department of Science Teaching, The Weizmann Institute of Science
Rehovot 76100, Israel
e-mail: bruria.haberman@weizmann.ac.il

Received: October 2006

Abstract. Abstract data types constitute a central tool in computer science and play an important role in problem solving, knowledge representation, and programming. In this paper, formal and practical aspects of utilizing abstract data types (ADTs) are discussed in the context of logic programming when using the Prolog programming language. The approach is presented in the following stages: (a) First, alternative ways of implementing ADTs in terms of Prolog constructs are presented and partial encapsulation of ADTs in terms of *grey boxes* is demonstrated. (b) Next, complete encapsulation of ADTs in terms of *black boxes* is suggested in a way that strictly reflects the concept's formal computer science definition while taking into consideration the characteristics and constraints of the logic programming paradigm. (c) Finally, implications for instruction are discussed.
Key words: abstract data types, information hiding, black boxes, grey boxes, logic programming.

Introduction

Abstract data types (ADT) constitute a central tool in computer science (CS) and play an important role in problem solving, knowledge representation, and programming (Aho and Ullman, 1992; Dale and Walker, 1996). Abstract data types are mathematical models with associated methods which should be implemented in terms of *black boxes*. ADTs are used according to the information hiding principle (Parnas, 1972) by determining *which* predefined ADT-operations should be invoked disregarding *how* they are implemented within the black box and how they are actually performed. Logic programming (LP) emphasizes the declarative *what instead of how* aspects of programming (Sterling and Shapiro, 1994). The Prolog programming language which is an implementation of LP abstracts the manipulation of compound data structures by hiding procedural aspects and details of their implementation (Ben-Ari, 1996), and thus may be considered as suitable for implementing and utilizing abstract data types (Scherz and Haberman, 1995; Haberman *et al.*, 2002). Surprisingly however, it is the declarative nature of LP that causes difficulties to implement a mechanism that strictly supports a complete encapsulation of ADTs.

In this paper formal and practical aspects of utilizing ADTs in the context of LP, when using the Prolog programming language, are discussed. The approach is presented in the following stages: (a) First, alternative ways of implementing ADTs in terms of Prolog constructs are presented; specifically, partial encapsulation of ADTs is demonstrated in terms of *grey boxes* that reveal the implementation of the ADT's model and hide the implementation of its associated methods; (b) Next, difficulties are discussed concerning conforming full encapsulation of ADTs in such a way that the concept's formal computer science definition is strictly reflected while considering the characteristics and the constraints of the logic programming paradigm and suggesting a solution to deal with this difficulty; actually, a method for complete encapsulation of ADTs in terms of *black boxes* is suggested. (c) Finally, implications for instruction are discussed.

Background

The Notions of Abstract Data Types

Formally, ADT is a mathematical (formal) model with a set of methods – operations and relations that are defined on that model. The ADT concept follows three perspectives: specification, implementation, and use. Specification of an ADT is achieved by using a formal verbal definition of its model and its methods, and may be illustrated by a graphical description. Implementation of an ADT is achieved in terms of the programming language data structures and programming constructs by: (a) organizing the data according to the ADT's mathematical model and (b) formulating a suitable code for each of the specified ADT's methods. The use of ADT is done by invoking its already implemented methods to formulate new code (Aho and Ullman, 1992; Dale and Walker, 1996). The implementation of an ADT should be actually achieved by creating a black box which is a fully implemented component with predictable functionality and pre-defined interface. Every black box has two components: (a) an interface visible to the user, which describes the implemented methods and their meaning and includes assumptions that relate to the way they should be invoked during a programming process; (b) an implementation component that encapsulates the details of how the methods were implemented. The underlying idea of using black boxes, according to the information hiding principle, is that the end-user is only permitted to know what the black box does, and is not allowed to know how the operation is done. Accordingly, the end-user does not need to know how predefined operations are implemented within the black box; therefore the interface does not include any clues about the implementation details, the access to source code is denied, and the use of black boxes is done by transparently invoking the encapsulated predefined operations to define new operations.

Consequently, ADTs have the following formally defined characteristics: The specification of an ADT is independent of the implementation environment. The implementation of an ADT is encapsulated in terms of a black box and is performed according to its specification; there may be several alternative implementations of a black box that represent an ADT; the use of a black box is independent of its implementation and is binding to its interface.

The Logic Programming Paradigm

Logic programming (LP) is a declarative programming paradigm; it is derived from an abstract model that is independent of a machine model. "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 and Shapiro, 1994, p. 3). It is based on the observation that formulas in mathematical logic can be interpreted as specification of computations (Ben-Ari, 1996).

A logic program contains a set of assumptions (logical axioms) defining the relations between objects. The program describes the knowledge about the problem that is to be solved – *what* is to be solved. No explicit instructions for the operations are given regarding *how* to solve the problem. Logic programs behave much like executable specifications. Consequently, logic programming enables programmers to concentrate on the declarative and abstract aspects of problem solving, and liberates them from dealing with the procedural details of the computational process. The computation of a logic program represents a deduction of consequences of the program. Actually, a logic program is executed by providing it with a goal statement, which is a logical statement to be proved as a consequence of the set of assumptions in the program. The proof of a goal statement from the program is performed by a constructive computation mechanism, which is based on the applying the rule of universal modus ponens.

The Prolog programming language is an implementation of logic programming ("Prolog was envisaged as a first approximation to logic programming" (Sterling and Shapiro, 1994, p. 147)). In contrast to logic programs that their construction and understanding is independent of any concrete execution mechanism, Prolog programs have precise operational meaning as instructions for execution on a computer. Nevertheless, effective Prolog programming requires an understanding of the theory of logic programming (Sterling and Shapiro, 1994). The classical dialects of Prolog (Edinburgh notation) are not typed, so the programmer is not required to declare the types of predicates' arguments. The fact that the Prolog language is not statically type-checked might cause difficulties in debugging, especially when type errors are encountered (Ben-Ari, 1996); however, it may contribute to the flexibility of knowledge presentation. For example, a list in Prolog can include elements of different types (in contrast to typed languages, e.g., Pascal or C).

A major abstraction of logic programming is that assignment statements and explicit pointers are no longer used; instead, a generalized pattern matching mechanism called unification is used to construct and decompose data structures. Implementations of unification create implicit pointers between components of data structures, but all the programmer sees is abstract data structures such as lists, records and trees (Ben-Ari, 1996). Consequently, as a declarative language, Prolog enables the programmer to program at a high level of abstraction, and thus is very suitable and convenient for knowledge representation and problem solving.

Implementing Abstract Data Types in Prolog

In this section we discuss formal and practical aspects of utilizing abstract data types (ADTs) in the context of logic programming when using the Prolog programming language. According to the "principle of information hiding" an implementation of an ADT in terms of a programming language must support a strict encapsulation of its formalization – both of the ADT's formal model and of the operations and relations defined in that model (Parnas, 1972). Apparently a complete encapsulation of an ADT appears to be a complex assignment in various programming languages and specifically cannot be trivially achieved in logic programming (implemented in Prolog).

Abstract data types can be implemented in alternative ways using variety of Prolog constructs (Scherz and Haberman, 1995). In this paper we discuss alternative ways of implementing ADTs in terms of Prolog constructs using grey boxes and black boxes. In contrast to black boxes that are fully implemented components with predictable functionality and pre-defined interfaces (complete encapsulation), grey boxes reveal parts of their internal workings, not just the relations between the input and output (partial encapsulation). The information can become as detailed as necessary where needed (Buechi and Weck, 1997; Kiczales, 1994; Haberman *et al.*, 2002; Resnick *et al.*, 2000).

The novelty of this paper is that the suggested black-box based method reflects the concept's formal CS definition while taking to consideration the characteristics and the constraints of the logic programming paradigm. Without loss of generality we demonstrate and discuss alternative implementations of the *list* ADT using variety of Prolog constructs; similar approach can be applied to implement any abstract data type as well.

Approach A – Partial Encapsulation of ADTs

The *list* abstract data type is an ordered set of elements, with the following methods: membership of an element in a list, place of an element in a list, length of list, first element of list, last element of list, concatenation of two list to get a third list, reverse of a list, etc.

Implementing the list ADT using Lists in Prolog

The list in Prolog is a recursive data structure: either it is empty (notated as [ ]), or it has a Head – the first element of the list, and a Tail – a list of the rest of the list's items (notated as [Head | Tail]).

Fig. 1 illustrates the *list* ADT grey box implemented in terms of the list Prolog data structure. It consists of: (a) an *interface* that describes how the *list* model is implemented, and describes the meaning of each general predicate which represent the methods that are defined on the *list* model, including assumptions about their transparent use, and (b) an *implementation module* that includes the implementation of general predicates.

The implementation is based on the recursive nature of the list data structure, and mostly is done in terms of recursive definitions. For example, the definition of $member(Item, List)$ is based on the following idea: an item is a *member* in a list if

The Interface: % The list is implemented using the list in Prolog data structure % General predicates: % member(Item, List) – Item is a member of the List % place(Place, Item, List) – Item is put in a Place on the List % length(Number, List) – Number describes the length of the List % concatenate(List1, List2, List3) – List3 is the concatenation of List1 with List2
The Implementation module: % member(Item, List) member(X, [X|_]). member(X, [_|Tail]):- member(X, Tail). ⋮ ⋮

Fig. 1. Part of the List ADT grey box implemented in terms of the List Prolog data structure.

it is a Head of the list, or if he is a *member* in the list's Tail. The box in Fig. 1 is considered as a *grey box* because it reveals some implementation details; thus only partial encapsulation is obtained.

The *list* grey box (presented in Fig. 1) can be used to solve a given problem. The list Prolog data structure is used to store concrete data about a specific list and the general list predicates are used to manipulate the list. In the following example, the list Prolog data structure is used to present a list of students' names:

$$students\big([\text{`}Ben\text{'}, \text{`}Dana\text{'}, \text{`}Roy\text{'}, \text{`}Mary\text{'}, \text{`}David\text{'}]\big).$$

A name of one specific student is retrieved from the students' list by transparent invoking of the $member/2$ general predicate:

$$\begin{aligned} student(Student) :- \quad & students(List\_of\_names), \\ & member(Student, List\_of\_names). \end{aligned}$$

This method of using lists as ADTs enables the programmer to store data easily in terms of Prolog lists data structures and to retrieve the stored data in a friendly simple manner. The use of predefined predicates enables to perform list processing in a simple way, without actually dealing with technical details related to list-predicates' recursive definitions. However, this method does not fully support the encapsulation of the list ADT because it does not hide the way that the data is stored, and the programmer must explicitly use the list Prolog structure to store the data.

Implementing the List ADT using Basic Relationships

One method of implementing the *list* ADT is based on the use of the $successor(Item,$ $Suc\_Item, List\_identifier)$ basic relationship. For example, a list of students is presented as follows (l_id presents the identifier of the given list, nil presents a dummy first

The Interface: % The concrete data is presented in terms of the $successor/3$ basic relationship % General predicates: % member(Item, List_id) – Item is a member of the List % place(Place, Item, List_id) – Item is put in a Place on the List % successor(Item, Suc_Item, List_id) – Suc_Item is the successor of Item in the List % length(Number, List_id) – Number describes the length of the List % concatenate(List1_id, List2_id, List3_id) – List3 is the concatenation of List1 with List2
The Implementation module: % member(Item, List_id) member(X, L_id):- successor(Y, X, L_id). % place(Place, Item, List_id) place(1, X, L_id):- successor(nil, X, L_id). place(N, X, L_id):- successor(Y, X, L_id), place(M,Y,L_id), N is M + 1. ⋮ ⋮

Fig. 2. Part of the List grey box implemented in terms of the $successor/3$ basic relationship.

item):

$$successor(nil, \text{‘}Ben\text{’}, l\_id).$$
$$successor(\text{‘}Ben\text{’}, \text{‘}Dana\text{’}, l\_id).$$
$$successor(\text{‘}Dana\text{’}, \text{‘}Roy\text{’}, l\_id).$$
$$successor(\text{‘}Roy\text{’}, \text{‘}Mary\text{’}, l\_id).$$
$$successor(\text{‘}Mary\text{’}, \text{‘}David\text{’}, l\_id).$$

A name of a specific student can be presented as follows:

$$student(Student, List\_id) : -member(Student, List\_id).$$

Alternative implementation of the *list* ADT is by using the $place(Place, Item, List\_identifier)$ basic relationship that presents the place of an item in a list. In this case, a list of students is presented as follows:

$$place(1, \text{‘}Ben\text{’}, l\_id).$$
$$place(2, \text{‘}Dana\text{’}, l\_id).$$
$$place(3, \text{‘}Roy\text{’}, l\_id).$$
$$place(4, \text{‘}Mary\text{’}, l\_id).$$
$$place(5, \text{‘}David\text{’}, l\_id).$$

```
The Interface:
% The concrete data is presented in terms of the place/3 basic relationship
% General predicates:
% member(Item, List_id) – Item is a member of the List
% place(Place, Item, List_id) – Item is put in a Place on the List
% successor(Item, Suc_Item, List_id) – Suc_Item is the successor of Item in the List
% length(Number, List_id) – Number describes the length of the List
% concatenate(List1_id, List2_id, List3_id) – List3 is the concatenation of List1 with List2
```

```
The Implementation module:
% member(Item, List_id)
    member(X, L_id):- place(_, X, L_id).
    .
    .
    .
    .
```

Fig. 3. Part of the List grey box implemented in terms of the *place/3* basic relationship.

Following this presentation, a name of a specific student can be presented as follows:

$$student(Student, List\_id) : -member(Student, List\_id).$$

The suggested above grey boxes have the following common shortcomings: (1) they don't have identical interfaces, and (2) they differ in the way the data (a given list) is organized. Although these boxes support transparent invoking of general list predicates, the programmer still must know how the ADT's formal model is implemented to present concrete data. This method of explicitly using language constructs to store data is often called by students "casting into a pattern" (Haberman and Scherz, 2005).

To summarize, the approach presented in this section essentially supports **partial encapsulation** of ADTs which is not consistent with the strict essence of the ADT formal concept.

Why a Complete Encapsulation of ADT in Prolog is Problematic?

When the main goal of teaching Prolog is to introduce to students a declarative environment for (partial) implementation of propositional logic and first order logic, only logical predicates should be used while extra-logic predicates for I/O processing and data base updating should not be introduced to the students at all (Haberman *et al.*, 2002; Gal-Ezer and Harel, 1999). According to this approach, students write Prolog programs using an editor, while formulating a set of assumptions in terms of rules and facts. The specific concrete data about the problem is determined before the execution of the program using data-predicates and not interactively during execution of the program (e.g., in C). Actually, there is a resemblance between the student's performance as a "logic program programmer" who develops a Prolog program and ask queries with respect to it, and his performance as a "logician" who formulates a set of axioms with "pen and paper" and checks whether a specific consequence can be proved with respect to a set of axioms, through a logical deductive process.

The restriction of not using extra-logic predicates in the process of programming causes difficulties in the implementation of ADTs in Prolog environment. First, there is a difficulty in implementing the “create” operation that serves for creating a specific data structure according to the specification of the ADT’s formal model. Second, there is a difficulty in encapsulating and hiding the implementation of the ADT’s formal model. Therefore it is hard to accomplish a “full” encapsulation of an ADT in Prolog environment while using only logic predicates. This was the reason for our compromise (for a long period) to present to the students a **partially-hidden** implementation of ADTs (Haberman and Scherz, 2005).

Suggestted Solution for Complete Encapsulation (Approach B)

In order to enable complete encapsulation of ADTs, without violating the declarative nature of the Prolog language, I suggest representing each ADT using an abstract description. The abstract description consists of a string that resembles the conventional mathematical notation used to represent the referred ADT. Accordingly, the *list* ADT should be represented by a collection of items surrounded by “( )” brackets, and the *set* ADT should be represented by a collection of items surrounded by “{ }” brackets (as illustrated in Table 1).

This convention enables the programmer to represent data in Prolog programs in terms of simple facts, without referring to concrete Prolog data structure. In the case that the programmer decides to represent the data differently, it is his responsibility to formulate rules that convert his presentation to the agreed upon one.

In order to support complete encapsulation an additional module for each “ADT black box” was developed – ***The coordinator module***. The coordinator communicates with the program and with the “ADT black box” and actually performs two functions – *create* and *describe* (as illustrated in Fig. 4).

The first function of the coordinator is to convert the abstract presentation of a given data to a compatible one presented in terms of the concrete data structure that implements the ADT’s formal model in the ADT black box. This conversion actually serves as a ***create*** operation that creates a specific instance of the ADT’s formal model. Another

Table 1

Abstract description of ADTs

ADT	Abstract description
List	(item1, item2, item3, . . .)
Set	{ item1, item2, item3, . . . }
Tree	{ edge(node1,node2), edge(node1,node3), edge(node3,node4) }
	{ node1, node2, node3, node4 }

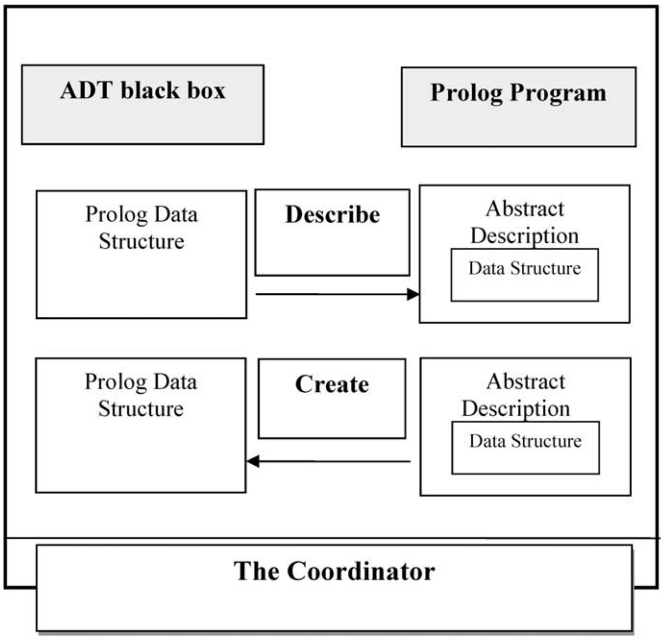

Fig. 4. The Coordinator's function.

function of the coordinator is to perform the opposite conversion – to ***describe*** a Prolog data structure in terms of an abstract presentation.

The following generic template describes the definition of a general predicate using a ***coordinator***:

General predicate :-
Create,
Predicate defined in terms of concrete data structures,
Describe.

Fig. 5 illustrates two alternative coordinators of *list* black boxes that are implemented according to the approach described above. One black box is implemented in terms of the list Prolog data structure (Fig. 5.a) and the successor basic relation implements the other one (Fig. 5.b). The coordinators have two components: (a) an interface, and (b) an implementation component. Note that the implementation of *member/2* predicate requires only the use of the *create* operation to convert the abstract description to a concrete data structure, since no inverse conversion is necessary. Contrarily, the implementation of *concatenate/3* predicate requires the use of both *create* and *describe* operations since it is necessary to convert the abstract description to a concrete data structure and visa versa to enable an abstract description of the result.

This approach supports complete encapsulation; it enables to replace an ADT black box by an alternative one, implemented in a different way, without bothering about the internal implementation.

(a) The formal model is implemented in terms of lists in Prolog

Interface:

```
% create(Abstract_Description, Data_Structure)
% convert from ( _,_,_,. . .) to [ _,_,_,. . . . . .]

% describe(Data_Structure , Abstract_Description)
% convert from [ _,_,_,. . .] to ( _,_,_,. . . . . .)

% member(Item, Abstract_description_of_List)

% concatenate (A_L1,A_L2,A_L3)
% A_L1 and A_L2 are given lists presented using abstract description,
% A_L3 is the result of the concatenation of A_L1 and A_L2, and should be
% presented in terms of abstract description
```

The Implementation Module:

```
% create(Abstract_Description, Data_Structure)
create( (X), [X]) :- X /= (_ , _).
create( (X,Y), [X | T]) :- create(Y,T).

% describe(Data_Structure , Abstract_Description)
describe( [X], (X)) :- X /= [_ | _].
describe( [ X | Y], (X ,T)) :- describe(Y,T).

% member(Item, Abstract_description_of_List)
member(Item, Abstract_description_of_List):-
        create(Abstract_description_of_List, List_Prolog_Data_Structure),
        member1(Item, List_Prolog_Data_Structure).

% concatenate (A_L1,A_L2,A_L3)
        concatenate (A_L1,A_L2,A_L3):-
        create(A_L1, C_L1),
        create(A_L2, C_L2),
        concatenate1 (C_L1,C_L2,C_L3),
        describe(C_L3,A_L3).
                .
                .
                .
                .
```

(b) The formal model is implemented in terms of *successor/3* (Fig. 5)

Interface:

```
% create(Abstract_Description, Data_Structure)
% convert from ( _,_,_,. . .) to succsessor(Item,Suc_Item,List_Index)

% describe(Data_Structure , Abstract_Description)
% convert from succsessor(Item, Suc_Item, List_index) to ( _,_,_,. . . . . .)

% member(Item, Abstract_description_of_List)

% concatenate (A_L1,A_L2,A_L3)
% A_L1 and A_L2 are given lists presented using abstract description,
% A_L3 is the result of the concatenation of A_L1 and A_L2, and should be
% presented in terms of abstract description
```

To be continued

Fig. 5. The coordinators of two alternative “list ADT black boxes”.

Continuation of Fig. 5

The Implementation Module:

```
% create(Abstract_Description, Data_Structure)
create(X, N):- define_list_index(N), create1(X,N).
define_list_index(1):- not succsessor(_,_,_).
define_list_index(N):- succsessor(_,_,M),
        not (succsessor(_,_,T), T > M ), N is M + 1.
create1( (X,Y), N ) :- Y /=(_ ,_),assert(succsessor(X,Y,N) ).
create1( (X,Y), [X | T]) :- Y = (A,B),
        assert(succsessor(X,A,N) ), create1(Y,N).

% describe(Data_Structure, Abstract_Description)
describe( List_index, (Item, Suc_Item)) :-
  succsessor(Item, Suc_Item, List_index),
  not succsessor(_, Item, List_index),
  not succsessor(Suc_Item, _, List_index).
describe( List_index, (Item,T)) :-
    succsessor(Item, _, List_index),
    describe(List_index,T).

% member(Item, Abstract_description_of_List)
member(Item, Abstract_description_of_List):-
  create(Abstract_description_of_List, List_index),
    member1(Item, List_index).

% concatenate (A_L1,A_L2,A_L3)
concatenate (A_L1,A_L2,A_L3):-
  create(A_L1, L1_index), create(A_L2, L2_index),
  concatenate1(L1_index,L2_index,L3_index),
  describe(L3_index,A_L3).
              .
              .
              .
              .
```

Fig. 5. The coordinators of two alternative "list ADT black boxes".

Conclusions and Implication for Instruction

The alternative approaches for implementing ADTs presented in this paper can be utilized for goal-oriented instruction of abstract data types to students who study logic programming in the Prolog environment.

ADTs may be introduced to students in various levels of abstraction to achieve various instructional goals: (a) as a formal computer science concept, (b) as a problem-solving tool, and (c) as means to teach compound programming language data structures. For example, when the main goal is to emphasize the formal CS definition of ADT, the instructional approach should offer a method that supports the concept's strict formal definition, meaning full encapsulation of implementation details (both of the ADT's formal model and its methods.) Approach B is suitable for that purpose since it uses abstract descriptions of ADTs and "upgraded black boxes" (with coordinator modules). On the other hand, when the main goal is teaching compound programming constructs and data structures, approach A is recommended since it supports partial encapsulation- reveals the implementation of the ADT's formal model, but hides the implementation of its methods. Actually, approach A enables the students practicing the use of predefined methods to process concrete data structures, before learning how to implement these methods (Haberman and Scherz, 2005). Both approaches could be utilized to organize instruction around practicing problem-solving techniques.

To summarize, the paper illustrates that an instructional approach should be developed according to pedagogical assumptions and to subject matter considerations as well. The educators should carefully define the instructional goals and decide which characteristics and properties of the concepts should be emphasized and used in order to achieve the goals. If the main goal is to present a formal definition of a concept, the instructional approach should offer a method that supports the concept's strict formal definition. If the main goal is to use the formal concept as a tool to achieve pedagogical goals like enhancing problem solving skills, the instructional approach could be based on relevant properties of the concept.

References

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

Ben-Ari, M. (1996). *Understanding Programming Languages*. John Wiley.

Buechi, M., and W. Weck (1997). A plea for Grey-Box components. In *Workshop on Foundations of Object-Oriented Programming*. Zürich, September 1997.
Available: `http://www.cs.iastate.edu/~leavens/FoCBS/buechi.html`

Dale, N., and H.M. Walker (1996). *Abstract Data Types – Specifications, Implementations, and Applications*. D.C. Heath and Company.

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

Haberman, B., E. Shapiro and Z. Scherz (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 Z. Scherz (2005). Evolving boxes as flexible tools for teaching high-school students declarative and procedural aspects of logic programming. *Lecture Notes in Computer Science*. Springer-Verlag GmbH. pp. 156–165.

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

Parnas, D.L. (1972). On the criteria to be used in decomposing systems into modules. *Communication of the ACM*, **15**(12), 1053–1058.

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

Scherz, Z., and B. Haberman (1995). Logic programming based curriculum for high school students: The use of abstract data types. *SIGCSE Bulletin*, **27**(1), 331–335.

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

B. Haberman received her PhD degree in science teaching from the Weizmann Institute of Science in 1999. She is currently an instructor in the Department of Computer Science in the Holon Institute of Technology. She is also a member of the computer science team in the Department of Science Teaching in the Weizmann Institute of Science, and a leading member of Machshava – the Israeli 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, and algorithmic patterns. She has developed academic programs for undergraduate level in computer science. Her primary research interests are computer science educational research, students' conceptualization of computer science, as well as in-service teacher education and distance learning.

Formalūs ir praktiniai abstrakčių duomenų tipų naudojimo PROLOG komandose aspektai

Bruria HABERMAN

Abstraktūs duomenų tipai yra viena pagrindinių programų inžinerijos priemonių, svarbi sprendimų priėmimo, žinių atvaizdavimo ir taikomojo programavimo uždaviniuose. Šiame straipsnyje nagrinėjami formalūs ir praktiniai abstrakčių duomenų tipų (ADT) vartojimo aspektai loginiame programavime PROLOG programavimo kalba. Pateikiama metodologija susideda iš tokių žingsnių: a) pateikiami alternatyvūs ADT naudojimo būdai PROLOG konstruktų atveju, dalinis ADT įkapsuliavimas taikant ”pilkosios dėžės” terminiją; b) pasiūlytas pilnas ADT įkapsuliavimas taikant ”juodosios dėžės” terminiją, įvertinant loginio programavimo paradigmos charakteristikas ir apribojimus; c) pateikiamos ir aptariamos išvados.