A Schema-Based Approach to Teaching Programming in Lisp and Prolog *

Mária Bieliková and Pavol Návrat
Slovak University of Technology,
Dept. of Computer Science and Engineering,
Ilkovičova 3, 812 19 Bratislava, Slovakia
{bielik,navrat}@elf.stuba.sk
http://www.dcs.elf.stuba.sk/~bielik, http://www.elf.stuba.sk/~navrat

An approach to teaching programming in Lisp and Prolog is presented. Frequently, the languages are used in introductory courses to functional and logic programming, respectively. It is becoming generally accepted that various standardized programming generalizations and abstractions facilitate learning programming. Considering program schemes and programming techniques, both the programming paradigms have perhaps surprisingly much in common. We support this hypothesis by presenting several examples of similar schemes in respective languages. We argue that once a student has learned fundamentals of one programming paradigm, she can learn the other one more easily by formulating the corresponding schemata. The approach is illustrated in the paper using examples from list processing.

Keywords: program schemata, Lisp, Prolog, list, recursion, functional paradigm, logic paradigm

1 Introduction

Producing fairly complex and interesting programs in Lisp and Prolog requires mastering only a few constructs. Syntax of these languages is very simple (moreover, in Prolog it can be made to look superficially very much like a natural language). Therefore, intuitively it seems that Lisp and Prolog should be fairly easy to learn. However, studies have shown that this is not the case [11, 3].

It has been recognized that many logic programs, e.g. list processing programs, are structured similarly, and can be understood as instances of particular program schemata [5]. The similar holds for functional programs, too. Consequently, program schemata are frequently used in intelligent tutoring systems [7].

A schema-based approach facilitates learning of general programming techniques by requiring the student to instantiate templates which are generalizations of a program to solve the assignement. This instructional technique has proven useful when recursion is taught. Recursion is a very difficult concept for most novice programmers

*The work reported here was partially supported by Slovak Science Grant Agency, grant No. 95/5195/605.

[8]. One of the major causes of difficulties is the lack of a structured programming construct for recursion in programming languages like Lisp and Prolog. A schema-based approach enables to alleviate some of these difficulties by providing a novice programmer with a set of standard structures (or program schemata) with which one may build complex and interesting recursive Prolog programs [7].

At the Slovak University of Technology, Prolog and Lisp are the languages used in teaching the standard *Functional and Logic Programming* course. Various pragmatic reasons outweight at least partially the possible weakness of this choice with regard to their known limitations to being the perfectly proper functional or logic language, respectively. Our approach is to teach programming by program schemata construction and explanation. Program schemata construction is helped by presentation of a few examples. After a particular program schema was introduced, students are required to instantiate the schema both as a writing exercise and afterwards as a programming exercise.

Our experience with this kind of presentation of these alternative non-procedural paradigms is very promising. There are at least three benefits of such an approach:

- using program schemata in programming supports understanding of basic programming concepts, both functional and logical ones, similarly as demonstrated by [7],
- comparison of program schemata in both paradigms facilitates learning differences and similarities between functional and logic programming which in turn improves effective programming,
- teaching with schemata provides the tutor with more accurate estimation of the student's problem solving capability.

The paper is organized as follows. In Section 2 an example of moving from functional and logic programs to program schemata is presented. More general program schemata are constructed. In Section 3, there are solved problems by instantiating the already introduced program schemata. Using programs that were just constructed, we show how to produce corresponding schemata by specializing general program schemata. In Section 4, we take the presented program schemata and discuss differences and similarities between functional and logic programming. Next, we give a brief overview of the related work. The paper closes with our conclusions.

2 Program Schemata for List Processing

Many programs share a common underlying structure or a schema [12, 3, 6, 11]. Such classes of programs exemplify more general programming techniques. Adelson [1] has shown that a major difference between novice and expert programmers is in their organization of programming concepts. She found that novices tend to use a syntax-based organization, while expert programmers use a more abstract hierarchical organization of algorithms. A detailed understanding of a programming concept contributes to changing from novice to expert.

Our schema-based approach to teaching programming is similar to the work of Gegg-Harrison [6, 7]. The main difference is in the method of instruction. Program schemata are taught just once when the first declarative language is introduced (e.g. Lisp). Afterwards, students modify known program schemata when learning programming in the other declarative language (e.g. Prolog). Similarly to [2], we feel that students should practice as much of the program forming process as possible (i.e., they should learn by doing rather than by simply being told). Tutor should provide a student with a structure of the problem solution (i.e., provide the student with a template to fill in).

We propose more specialized schemata than those presented in [6, 7] for learning Prolog. We believe that a detailed understanding of one problem will help solve other problems, even of possibly fairly different nature.

We have developed sets of program schemata that capture some of the essence of both the functional and logic programming. In this paper we concentrate on problems related to list processing. A list is a basic data type in both Lisp and Prolog. When a list is to be processed, generally a method of processing recursively all the elements of the list is used.

First, several examples of typical list processing problems and their solutions are presented to students. As an example, let us assume the classical task to find a sum of all the elements of an arbitrary list of integers. Standard Lisp and Prolog programs which solve the task are as follows:

```
(defun sum (list)                          sum([ ], 0).
   (cond ((null list) 0)                   sum([H|T], Result) :-
         (t  (+ (first list)                   sum(T, X),
                (sum (rest list))))))          Result is X + H.
```

Both Lisp (on the left side) and Prolog (on the right side) programs recursively traverse the list of elements and construct a result using addition (+). The purpose of the base case is to stop the recursion when the list becomes empty and to return 0 for the sum.

Next, consider the task of finding a product of a list of integers:

```
(defun product (list)                      product([ ], 1).
   (cond ((null list) 1)                   product([H|T], Result) :-
         (t  (* (first list)                   product(T, X),
                (product (rest list))))))      Result is X * H.
```

Generalizing the two examples, the following program schemata for global list processing can be formed:

```
(defun reduceList (list)                   reduceList([ ], Base).
  (cond ((null list) Base)                 reduceList([H|T], Result) :-
        (t (Constructor                        reduceList(T, X),
             (first list)                      Constructor(H,X,Result).
             (reduceList (rest list))))))
```

Note that when describing program schemata, we use a standard Lisp syntax and a slightly extended standard Prolog syntax (predicate symbols can begin with a capital letter). Two types of variables are used: first-order variables (arguments of functions in Lisp and standard Prolog variables) and schema variables (functional symbols in

Lisp and predicate symbols in Prolog both beginning with a capital letter). First-order schema variables permit the generalization of expressions, while second-order schema variables permit the generalization of functions or predicates, respectively. We have found important the syntax of program schemata to be the same or almost the same as the syntax of regular programs.

The program schema `reduceList` contains two second-order variables: `Constructor` defines the process of constructing an output and `Base` defines how to construct the output in case the processing of a list terminates. Apart from the different syntax of Lisp and Prolog and the possibility of nested expressions in Lisp, both program schemata have the same structure. Indeed, mastering one of then is helpful in mastering the other, regardless their order.

Let us turn from recursive list processing to recursive number processing. Consider the task of computing the factorial of some positive integer:

```
(defun factorial (m)
  (cond ((eq m 0) 1)
        (t (* m
              (factorial (- m 1))))))
```

```
factorial(0, 1).
factorial(M, Result) :-
    N is M - 1,
    factorial(N, X),
    Result is M * X.
```

When comparing the programs for `factorial` with the program schemata `reduceList` in both Prolog and Lisp, one can observe a strong similarity. Differences remain in the stopping condition (number equal to the constant `0` versus list is empty) and in the way of decomposing the input. In case of list processing, input argument (a list) is decomposed to its first element and to the rest of the list (in Lisp program using functions `first` and `rest`; in Prolog program by matching input list to the term `[H|T]`). More general schemata can be constructed by generalizing the stop condition and by permitting an arbitrary destructor to decompose the input. Generalized schemata capture all singly-recursive one input reduction programs:

```
(defun reduce (arg)
  (cond ((BaseCond arg) (Base arg))
        (t (Constructor
             (Destructor1 arg)
             (reduce (Destructor2 arg)))
        )))
```

```
reduce(Arg, Result) :-
    Base(Arg, Result)
reduce(Arg, Result) :-
    Destructor(Arg,H,T),
    reduce(T,X),
    Constructor(H,X,Result).
```

In the Lisp schema `reduce`, input is decomposed into two subexpressions by applying functions denoted as `Desctructor1` and `Destructor2` because the components of input are to be used in two subexpressions. In Prolog, on the other hand, output from a procedure is passed via arguments. Therefore, one procedure `Destructor` with two output arguments (`H, T`) is sufficient.

Note that in the Lisp program schema, two functions are used to express the base case (`BaseCond` and `Base`). They are necessary to capture the base case of the recursion where value `nil` should be returned (e.g. in case of list processing tasks).

Program schemata just introduced can be converted to higher-order programs by adding second-order expressions to the set of arguments. Lisp and other functional languages support higher-order functions. On the other hand, most logic programming languages do not have a full support of higher-order predicates. In [6], for this purpose λProlog is used which extends Prolog by incorporating higher-order Horn clauses, unification and λ-terms. However, higher-order programs are important mainly in automated support of program construction, i.e. program debugging, program transformation and program synthesizing.

For novice programmers, first-order program schemata like `reduceList` are more useful. By inspecting examples and using generalized program schemata like `reduce`, most of the students are able to devise new specialized program schemata. Let us present at least two examples.

3 Program Schemata Specialization

After having been explained general program schemata like `reduce`, students are required to solve specific tasks by instantiating the already introduced templates. Based on programs that were formed, they are required to construct a specialized schema for the class of problems being solved.

Let us illustrate the above approach with an example involving the more general schema `reduce` and more specialized schemata for mapping and filtering. Consider the problem of squaring all of the elements in a list. In order to solve this task, we must first apply a function that squares each element (`square`) as a constructor in the schema `reduce` and then put the squared element in front of the list produced by squaring all the elements in the remainder of the list:

```
(defun squareAll (arg)
  (cond ((null arg) nil)
        (t (cons
             (square (first arg))
             (squareAll (rest arg)))
        )))
```

```
squareAll(Arg, Result) :-
    Arg = [ ], Result = [ ].
squareAll(Arg, Result),
    Arg = [H|T],
    squareAll(T, X),
    square(X,Y), Result=[Y|X].
```

Producing a Lisp program according to the general schema (e.g. `reduce`) is usually more difficult than producing a Prolog program because the functional style of programming often does not make it possible to identify functional symbols precisely as they appear in the general schema. In `reduce`, the second-order schema variable `Constructor` is to be rewritten to functional symbols `cons` and `square`.

Student should use standard transformations to simplify the programs appropriately. One can use a set of techniques proposed in [3]. Generalizing these programs results in program schemata that are more specialized than `reduce`.

Here is the result of this exercise:

```
(defun map (list)
  (cond ((null list) nil)
        (t (cons
             (Transform (first arg))
             (map (rest arg))))))
```

```
map([ ], [ ]).
map([H|T], [Y|X]) :-
    map(T, X),
    Transform(H, Y).
```

Next, let us consider the problem of filtering out all non-integer elements from a list of arbitrary expressions. In this case the constructor has two alternatives: either the processed element is an integer (it is to be included in the resulting list), or it is not an integer (it is not to be included in the resulting list). In Lisp, the constructor is implemented by the function `cond`. The second alternative does not need any explicit precondition, so the predicate constant `t` is used. In Prolog, one should use disjunction (`;`). The second test can be avoided using the cut predicate (`!`).

```
(defun intOnly (arg)
  (cond ((null arg) nil)
        (t (cond
             ((integer arg)
               (cons (first arg)
                     (intOnly (rest arg))))
             (t (intOnly (rest arg)))))))
```

```
intOnly(Arg, Result) :-
    Arg = [ ], Result = [ ].
intOnly(Arg, Result),
    Arg = [H|T],
    intOnly(T, X),
    (integer(H), !, Result=[H|X];
     Result = X ).
```

Program schemata for filtering can be formed in a way similar to that for mapping. Note that the Prolog schema consists of three clauses. The Lisp schema also differs structurally from the original more general schema `reduce`.

```
(defun filter (list)
  (cond ((null list) nil)
        ((ElemTest (first list)
           (cons (first list)
                 (filter (rest list))))
        (t (filter (rest list)))))
```

```
filter([ ], [ ]).
filter([H|T], [H|X]) :-
     ElemTest(H), !,
     filter(T, X).
filter([H|T], X) :-
     filter(T, X).
```

Next, we discuss a possible approach to teaching programming based on the program schemata introduced above in slightly more detail.

4 Discussion

As we stated in the previous section, Lisp and Prolog program schemata have similar (but not necessarily the same) structure. Specialization in Lisp programs is more difficult due to the feature of function composition. If one abandons the functional style of programming and uses assignments then specialization is easier, but that would be outside the scope, and in fact against our aim of teaching functional programming.

It is to be noted that the presented Prolog program schemata are not aimed to teach backtracking. To teach this important feature of logic programming, we use different methods i.e., tracing of program execution together with their graphical representation.

Moreover, in the presented Prolog schemata we have adopted the following convention: the first argument is input and the second one is output (i.e., mode `p(+,-)`). The main reason for this assumption is to make Prolog schemata as close as possible to Lisp schemata. On the other hand, students are required to modify Prolog programs in order to use them in different ways and encouraged to discover all the possible meaningful combinations for input and output arguments.

For example, let the predicate `Transform` in the schema `map` is defined in such a way that it can be used in two ways: (i) the first argument as input is transformed into the second one (output) and (ii) the second argument as input is transformed into the first one (output). In this case the program `squareAll` can be used for mapping elements of the input list to the list of squared elements and for finding a list of elements such that their squares form the input list:

```
?- squareAll([1, 2, 3], Y).
   Y = [1, 4, 9] ->
?- squareAll(X, [1, 4, 9]).
   X = [1, 2, 3] ->
```

In [3] program schemata are criticized that they describe entire predicates and so a novice never has to address the problems of relating clause ordering to the control flow of the predicate. So there is danger that novices do not understand a schema, but still are about to apply it by instantiating. Our approach tries to avoid this by employing different teaching strategies and also due to the fact that students are required not only to use, but more importantly also to construct and modify program schemata.

5 Related work

In [3], five basic declarative Prolog programming techniques and rules for combination of techniques across entire predicates are described. A recursive clause is seen as a composition of a number of more elementary techniques. Due to the fact that even very simple programs are constructed by applying several techniques, this approach requires a good abstraction ability from students.

Program schemata that capture a large number of standard list processing programs are defined in [6, 7]. This approach is the closest to ours. However, we teach program schemata for both functional and logic programming paradigms and we exploit their similarities and differences. Moreover, our approach to teaching programming is not only to use predefined program schemata but also to guide the students in constructing new schemata and then experimenting and possibly modifying them.

In [11], too, there is defined a number of constructs suitable for building Prolog programs (*program skeletons* and *techniques* which extend skeletons and express steps in program construction). Separation of control flow and technique is the key idea of this approach. A student selects a skeleton and then enhances it, using standard techniques, to solve her particular problem. We feel that to understand skeletons can be rather difficult for novices without having some preceeding experience in building and fully understanding a number of programs.

In [13], a Lisp programming environment specifically designed to fulfil the needs of beginners learning to program in Lisp is described. Defining Lisp functions is simplified by providing templates for functions taught in the Lisp course. There is no generalization of function templates. Templates simply represent actual programs where second-order variables are displayed in parentheses (`<...>`).

6 Conclusion

In the paper we illustrated the approach to teaching programming in Lisp and Prolog by exploiting similarity of functional and logic programming. In fact, we have developed a set of recursive program schemata which cover three important aspects of recursion: (i) base case (single, multiple), (ii) method for recursive step execution (tail, fat recursion), (iii) reduction to a simpler task (monotonic, nonmonotonic recursion; single, multiple recursion; change in one or several arguments). In the paper, we presented examples of recursive programs with (i) a single base case, (ii) tail recursion and (iii) change in just one argument (monotonic, single recursive case).

Our approach involves the whole spectrum of types of learning as treated in the classical artificial intelligence literature. A student starts with learning by being told and shown examples together with a program schema which generalizes these examples. Next, the student continues learning by seeing examples and solving tasks by instantiating the program schemata. Then, the student is supposed to devise new program schemata which express common features of the sample programs by either generalizing them or by specializing or extending the already known schemata. She learns the subject by discovery. All these levels are combined with programming exercises.

Program schemata (generalizations), programming techniques or plans (abstractions) all represent various kinds of programming (meta-)knowledge. We find it important to make the knowledge explicit for the learner [10]. This aspect as discussed in this paper is mainly related to programming-in-small. Second important aspect is connected with understanding similarities and differencies between various programming paradigms [9].

References

1. B. Adelson. Problem solving and the development of abstract categories in programming languages. *Memory & Cognition*, 9:422–433, 1981.
2. J.R. Anderson and B.J. Reiser. The LISP tutor. *Byte*, 10(4):159–175, 1985.
3. A. Bowles and P. Brna. Programming plans and programming techniques. In P. Brna, S. Ohlsson, and H. Pain, editors, *Proceedings of World Conf. on Artificial Intelligence in Education*, pages 378–385, Edinburgh, Scotland, 1993. AACE Press.
4. A.T. Corbett and J.R. Anderson. Knowledge decomposition and subgoal reification in the ACT programming tutor. In *Proceedings of World Conf. on Artificial Intelligence in Education*, pages 469–476, Washington, D.C., 1995. AACE Press.
5. N.E. Fuchs and M.P.J. Fromherz. Schema-based transformations of logic programs. In T.P. Clement and K.K. Lau, editors, *Proceedings of Logic Program Synthesis and Transformation*, pages 1–14. Springer Verlag, 1992.
6. T.S. Gegg-Harrison. Representing logic program schemata in λProlog. In L. Sterling, editor, *Proceedings of the 12th Int. Conf. on Logic Programming*, pages 467–481, Kanagawa, Japan, 1995. MIT Press.
7. T.S. Gegg-Harrison. Extensible logic program schemata. In J. Gaallagher, editor, *Proceedings of the 6th Int. Conf. on Logic Program Synthesis and Transformation*, Stockholn, Sweden, 1996. Springer-Verlag.
8. S.M. Haynes. Explaining recursion to the unsophisticated. *SIGCSE Bulletin*, 27(3):3–6, 1995.
9. P. Návrat. A closer look at programming expertise: Critical survey of some methodological issues. *Information and Software Technology*, 38(1):37–46, 1996.
10. P. Návrat and V. Rozinajová. Making programming knowledge explicit. *Computers and Education*, 21(4):281–299, 1993.
11. L.S. Sterling and M. Kirschenbaum. Applying techniques to skeletons. In J.M. Jacquet, editor, *Constructing Logic Programs*, pages 127–140. John Wiley, 1993.
12. T. Van Le. *Techniques of Prolog Programming.* John Wiley & Sons, Inc., 1993.
13. G. Weber and A. Möllenberg. ELM programming environment: A tutoring system for LISP beginners. In K.F. Wender and F. Schmalhofer, editors, *Cognition and Computer Programming*, pages 373–408. Albex, 1995.