CREATIVE MATH. 12 (2003), 00-00

THE USE OF PROLOG FOR THE STUDY OF ALGEBRAIC STRUCTURES AND COMPLEX OPERATIONS

DAN POPA

Abstract. Should mathematicians learn the Prolog language? Due to its ability of modeling algebraic structures having complex operations and its backtracking algorithm able to scan a search space looking for solutions, Prolog becomes a great tool to study complex algebraic structures. The model of interactions between compilers when computer scientists are attempting to bootstrap a system or to create a new language or a new compiler is such an algebraic structure. This paper tries to answer "yes" at the question in the first line.

1. Introduction

The goal of this paper is to introduce a new methodology of work, intended to be available for the mathematicians which are interested in complex algebraic structures, meaning algebraic structures with complex operations. I refer to such operations of which flow of calculus is escaping from the chains of our intuition. In such an algebra, even to verify a calculus might be a terrible work. The methodology, as it is, is directly usable for works on finite sets but it can can also give us the base of the proofs by inductions which have to be made in order to "conquer" infinite sets having the same cardinal as the natural numbers set. The following results are selected from the algebraic part of a research concerning compilers, interactions of the translators and the extensibility of the programming languages. I wish to send all my gratitude to Professor D.Todoroi from ASEM (Republic of Moldova) for the subject of research which was the starting point of this work.

2. Requested Items

In order to practice this kind of research you will need:

- the object of the study, I mean an algebraic structure having one ore more complex operations. For example, I needed to study the compilers interactions.
- a computer (Actually, last year, I used an old Intergraph TD3 Workstation having an 150Mhz Pentium Processor, 64Mb RAM and a 1GB SCSI disk drive). Any modern PC can be used, but a better one will be more practical,

Received by the editors: 07.06.2003.

because of the increased speed. It will make symbolic calculation instead of you.
- a Prolog Interpreter or a Prolog Compiler. I used Turbo Prolog 2.0 produced years ago by Borland. Any other implementation of the Prolog Language may be used, but the syntax of an other Prolog dialect may be just a bit different.
- a manual of the Prolog language. From it you will be able to learn how can you translate an algebraic operation as a Prolog clause. Some particularity aspects concerning the version of the Prolog language you are planning to use can be find in such a manual. You may or you may not need it. The translation of algebraic operations as Prolog clauses looks very natural to me.
- a result obtained by calculations in that algebra - it will be the goal, the thing to be proved.
- an example of an algebraic operation translated in Prolog. This paper contains such an example. Let's see it:

3. Algebraic operations studied in this paper

Two different interactions of compilers had been studied. Both of them can produce a compiler by combining somehow a pair of other compilers. The algebraic operations were called "/" respectively "o". In this study a compiler is written as an implication L_i -> $L_j(L_k)$. Practically, for the algebraist, it consists in an ordered set of three values (i, j, k) which means, in fact, an ordered set of three languages, (L_i, L_j, L_k) selected from a large, indexed set or an indexed hierarchy of extensible languages. Semantically speaking, the ordered set (L_i, L_j, L_k) represents a translator from the L_i language to the L_j language written in the L_k language. The translator is a program, so it had been written in a language, too.
From the point of view of a mathematician, we have the definitions:
$L_i -> L_j(L_k)\ /\ L_j -> L_m(L_k) => L_i => L_m(L_k)$
$L_i -> L_j(L_k) \circ L_k -> L_m(L_n) => L_i -> L_j(L_m)$
where the "=>" sign shoul be read as "equal by definition" or "from that composition we get..." (Let's remark that somebody which is familiar with the compilers interactions will recognize the first operation as the cascaded link of two compilers and the second as the compilation of one compiler with an other.)

4. How can algebraic operations be translated in Prolog?

Let's take an example. The second operation, noted as "$\circ$" is defined as
$L_i -> L_j(L_k) \circ L_k -> L_m(L_n) => L_i -> L_j(L_m)$. In fact it means that
$X -> Y(Z) \circ Z -> T(M) => X -> Y(T)$ for every set of for languages X, Y, Z, T.
Using Prolog, the implication can be written as a clause of a predicate. As

name for the predicate I have used “compile”. An other predicate, “existaL” is expressing the fact that a language L is available in the modeled universe. When using a hierarchy of languages, “existaL” means “L belongs to the set of languages of the hierarchy”. Practically it is used by Prolog as a “come back and try again” point for the backtracking algorithm of the Prolog interpreter. Let’s see how the rule was translated as sample of code:

```
compile(X,Y,T) if existaL(X),
                  existaL(Y),
                  existaL(Z),
                  existaL(T),
                  existaL(M),
                  compile(X,Y,Z),
                  compile(Z,T,M).
```

Let’s remark the fact that both operands become predicates in the clause (last two lines) and the conclusion is written as the head of the clause, followed by “if” which can be seen as a translation of “=>”.

5. What kind of problems can be solved by a program?

Depending of the goal of the Prolog program, the Prolog interpretor or the machine code generated by the Prolog Compiler, when running, will try to discover the chains of clauses which stars from the hypothesis (the existed language and algebraic values) and lead to the desired conclusion. The conclusion itself may have variables as arguments. In this case, the resulted output will contain the list of all substitutions which can make the goal true, in that world created by hypothesis.

In fact, during our research, the Prolog programs which have been written was able to answer questions like this:

- How could the author of an other proof have reached a specific result? What calculus leads to his or her conclusion ?
- Can we get a specific value as a result of a calculus, which such algebraic operations and in specific hypothesis?
- If a new operation (or operator) is introduced in the problem’s universe, can we get now a specific result, involving all the operations?
- Did an other sequence of operations leading to the same value exist ? How the list of all possible sequences looks like?
- Being given a specific theorem involving calculations in such a structure, is this theorem valid for a set of finite cases? (Let’s remark that missing of solution in only one case may make the generalized theorem false. But, for a mathematician, to build an example of failure for a supposed valid theorem may sometimes be difficult).

6. Tips and tricks

The chains of rules like: A if B, B if A, ... A if B ... should be avoided. For example, we are able to do this by defining an order for the arguments of the predicates (not for the studied algebraic structure). Let's take an example.

When using a hierarchy of languages, the new developed compilers for the new extended languages of the hierarchy will be placed on the upper levels. We will be able to write clauses with inequalities, like this one:

```
compile(X,Y,T) if existaL(X),
                  existaL(Y),
                  existaL(Z),
                  existaL(T),
                  existaL(M), X > Y, Z > T,
                  compile(X,Y,Z),
                  compile(Z,T,M).
```

In such case, the language will be represented by their numbers in the hierarchy, as arguments. And the inequalities will prevent the cyclic reasoning.
In fact more other constrains can be introduced in the Prolog clauses, in order to reduce the search space and to avoid cyclic reasoning. For example, during our research we had noticed that the second compiler should run on a real machine. It means that Lm is placed in the $M = -1$ position of the hierarchy. The clause were consequentially rewritten:

```
compile(X,Y,T) if existaL(X),
                  existaL(Y),
                  existaL(Z),
                  existaL(T),
                  X > Y, Z > T,
                  compile(X,Y,Z),
                  compile(Z,T, -1),
                  write(“L”, X , ”-> L” , Y , “( written in L”, Z, ”) ”),
                  write(“L”, Z , “-> L” , T , “( written in L-1) ” ),
                  write(“ by translating the compiler => ” ),
                  write(“L”, X, “-> L”, Y, ” ( written in L” , T, ”) ”),
                  readln(_).
```

Notice the presence of the I/O predicates “write” and “readln”. Their side effects are similar with the effects of the same words of Pascal like languages. When the program is running, an output will be displayed and a key should be pressed in order to continue.

Running such a program...

When a program like above is running, all the sequences of algebraic operations will be displayed. The successful search message which began by “Solution:..”

is displayed by a "write" predicate included by the "goal" of the Prolog program.

The final output gave to me by the computer was the following, where a solution consists in a chain of operations.

```
$L_1$->$L_0$ (written in $L_0$) $L_0$->L-1(written in L-1) translating the compiler
=> $L_1$-> $L_0$ (written in L-1)
Solution:  $L_1$->$L_0$ (written in L-1)
$L_1$->$L_0$ (written in $L_0$) $L_0$->L-1 (written in L-1) translating the compiler
=>$L_1$->$L_0$ (written in L-1)
$L_2$->$L_0$ (written in $L_1$) $L_1$-> $L_0$ (written in L-1) translating the compiler
=> $L_2$-> $L_0$ (written in $L_0$)
$L_2$->$L_0$ (written in $L_0$) $L_0$->L-1 (written in L-1) translating the compiler
=>$L_2$->$L_0$ (written in L-1)
Solution:  $L_2$->$L_0$ (written in L-1)
$L_1$->$L_0$ (written in $L_0$) $L_0$->L-1 (written in L-1) translating the compiler
=>$L_1$->$L_0$ (written in L-1)
$L_2$->$L_0$ (written in $L_1$) $L1$-> $L_0$ (written in L-1) translating the compiler
=>$L_2$->$L_0$ (written in $L_0$)
$L_2$->$L_0$ (written in $L_0$) $L_0$-> L-1 (written in L-1) translating the compiler
=>$L_2$->$L_0$ (written in L-1)
$L_3$->$L_0$ (written in $L_2$) $L_2$-> $L_0$ (written in L-1) translating the compiler
=>$L_3$->$L_0$ (written in $L_0$)
$L_3$->$L_0$ (written in $L_0$) $L_0$-> L-1 (written in L-1) translating the compiler
=>$L_3$->$L_0$ (written in L-1)
Solution:  $L_3$->$L_0$ (written in L-1)
...
```

You may see how the computer proving the fact that values like L_n-> L_0 (L-1), where n is 1,2,3, may be obtained in our hypothesis, i.e. by combining the given translators. In fact it has just rebuild the calculus from a given theorem. In this case there is a lemma from the paper of Diana Micuşa, called the ND Lemma or LD Lemma (depending of the language in which the paper is written). In fact the proof should be completed by human mind, because the computer can only verify a finite number of cases. The step of going from n to $n + 1$ has to be made by the mathematician himself /herself, with the help of this examples. This is an easier task, because the computer may give us any examples we wish.

Warning: The search space usually grows up exponentially with the dimension of the looked for calculus. In our study, the value 12 of n is needed a bit more than an hour.

Remark: Using the computer we are thrown from an extreme to an other. At the start point we do not now how to combine the values, using operations to perform calculation and get the result but, after the use of the computer, we can get more examples than we need!

A complete program

We have picked one example (the 17th), from our research. Here it is:

```
domains
predicates
   existaL(integer)
   compile(integer,integer,integer)
   compile2(integer,integer,integer)
   rulez(integer)
clauses
   existaL(-1).
   existaL(0).
   existaL(1).
   existaL(2).
   existaL(3).
   existaL(4).
   existaL(5).
   existaL(6).
   existaL(7).
   existaL(8).
   existaL(9).
   existaL(10).
   existaL(11).
   existaL(12).

   rulez(-1).
   rulez(X) if compile2(X,-1,-1),
              write("Programs written in L can run", X),
              readln(_).
compile(0,-1,-1).
compile(X,0,Y) if existaL(X),
              existaL(Y),Y>=0,
              Y=X-1.
compile(X,Y,T) if existaL(X),
               existaL(Y),
               existaL(Z),
               existaL(T),
               X>Y,
               Z>T,
compile(X,Y,Z) ,
compile(Z,T,-1),
              write("L", X, "-> L" , Y , " (written in L", Z, ") "),
             write("L", Z, "-> L" , T , " (written in L-1) " ),
             write(" translating the compiler =>" ),
             write("L", X, "-> L" , Y , "( written in L", T, ")" ),
        readln(_).

compile2(0,-1,-1).
compile2(X,0,Y) if existaL(X),
               existaL(Y),
               Y>=0,
               Y=X-1.
compile2(X,Y,T) if existaL(X),
               existaL(Y),
               existaL(Z),
               existaL(T),
               X>Y,
               Z>T,
               compile2(X,Y,Z) ,
               compile2(Z,T,-1),
               readln(_).
goal
compile(X,0,-1),
write(" Solution:  L", X, "-> L0","(written in L-1)"),readln(_), fail
```

Let's remark the "goal" finished by the special predicate "fail" which forces the Prolog system to begin backtracking, searching for an other solution.

7. Conclusion

The research of complex algebraic structures was enriched with an unexpected tool: the Prolog language. To study such structures, especially to check calculus and to search for solutions may be made faster using a computer. A part of the results of our research (the methodology shown here being excluded) was presented as a paper called "Results concerning interactions between compilers and interpreters" in section 14, "Cybernetics and IT Technologies" of the "International Meeting of Young Researchers", 18-19 of April 2003 in Chişinău, Republic of Moldova.

Acknowledgment. Due to professor Henz Martin and his course CS3212 from the School of Computers in Singapore, available as Web resource, I could freely access a better made set of informations concerning J. Earley's theory and notations. Thank you very much!

References

[1] Earley J., Howard S., *A Formalism for Translator Interactions*, Communications of the ACM, Volume 13, Number 10, October 1970, p.607-617.

[2] Martin H., *CS3212*, School of Computers, Singapore, WWW resources.

[3] Micuşa D., Pereteatcu S., Todoroi D., Zabolotnii P., *The Extensible Programming Method as a tool of Object-Oriented Programming*, Proc. of the 9^{th} Romanian Symposium on Computer Science: ROSYCS'93, "Al.I.Cuza" Univ. of Iasi, Romnia, 1993, pp. 359-380.

[4] Micuşa D., *Draft of the Ph D Thesis* (probably finished now, when you read this text).

[5] Serbanati L.D., *Programing Languages and Compilers*, Editura Academiei RSR, 1987.

[6] Todoroi D., *Computer Science - The Adaptable Programming (The Basic Conception)* Ed. ASEM, Editor: I.V. Pottosin, Kishinev 1992.

Dan Popa
Department of Mathematics and Physics
University of Bacău
Spiru Haret 8, 600114 Bacău, Romania
E-mail:popavdan@yahoo.com