Teaching Pure LP with Prolog and a Fair Search Rule[*]

Manuel V. Hermenegildo[1,2,*], Jose F. Morales[1,2] and Pedro Lopez-Garcia[2,3]

[1]Universidad Politécnica de Madrid (UPM), Madrid, Spain
[2]IMDEA Software Institute, Madrid, Spain
[3]Spanish Council for Scientific Research (CSIC), Madrid, Spain

Abstract

Classic Prolog has many features, and even more in its modern incarnations. Many of these features are well aligned with the view of pure Logic Programming as both a specification tool and a programming language. However, some other Prolog aspects depart from this view. Classic examples that have received much attention are assert/retract or the cut. Our focus here is however on the depth-first search rule. While well justified by practical concerns, using only depth-first from the start also introduces early on the need to reason about termination, and also possibly a need to use impure features to make different modes produce answers in finite time. Termination of course has to be faced sooner or later by any programmer, but having to do it right at the start can detract from being able to convey early on the vision of Prolog as a declarative language where one can first concentrate on problem specification and/or knowledge representation and only later worry about efficiency. We review a number of ways in which some of these issues can be tackled when teaching Prolog, while still using throughout a Prolog system. The aim is tutorial, in the hope that these ideas can help other instructors that are teaching or plan to teach Prolog. We believe that some of these considerations may also come in handy when combining Prolog with modern, AI-assisted programming methodologies.

Keywords

Teaching Prolog, Teaching Logic Programming, Logic Programming, Constraint Logic Programming.

1. Introduction

In [1, 3] we presented some thoughts and lessons learned over time about how to teach (Constraint) Logic Programming –(C)LP– in general and Prolog in particular. We argued that teaching (C)LP and Prolog is necessary because it brings in many useful concepts and characteristics that are not present in other programming paradigms such as imperative, object-oriented, or functional programming. We also addressed different aspects of how to teach Prolog, covering from how to show the beauty and usefulness of the language to how to avoid some common pitfalls, misconceptions, and myths about this unique programming paradigm.

Our aim herein is to expand on the fundamental aspect of transmitting to students the beauty and power of Prolog. The (C)LP paradigm is based on logic and includes search as an intrinsic component, as well as the use of unification, and generalized pattern matching. This brings about interesting and distinctive aspects, such as the reversibility of programs and the ability to represent and reason about knowledge. It also makes it possible to formulate runnable specifications and efficient algorithms within the same formalism, making (C)LP not only a singular programming paradigm, but also a modeling and reasoning tool. We would like to explore how to convey these concepts already in the first steps of teaching (C)LP and Prolog, as well as to provide some ideas on how to do this using the same tool –a modern Prolog system– that will be used in the later, more algorithmically-oriented parts of a Prolog course.

Classic Prolog has many features, with even more in its modern incarnations. Many of these features are well aligned with the view of pure Logic Programming as both a specification tool and a

PEG 2024: 2nd Workshop on Prolog Education, October, 2024, Dallas, USA.

[*]Partially funded by MICINN projects PID2019-108528RB-C21 *ProCode*, TED2021-132464B-I00 *PRODIGY*, and by the Tezos foundation. We also thank David S. Warren and Daniela Ferreiro for very useful feedback.

[*]Corresponding author.

manuel.hermenegildo@imdea.org (M. V. Hermenegildo); josef.morales@imdea.org (J. F. Morales); pedro.lopez@csic.es (P. Lopez-Garcia)
0000-0002-7583-323X (M. V. Hermenegildo); 0000-0001-9782-8135 (J. F. Morales); 0000-0002-1092-2071 (P. Lopez-Garcia)

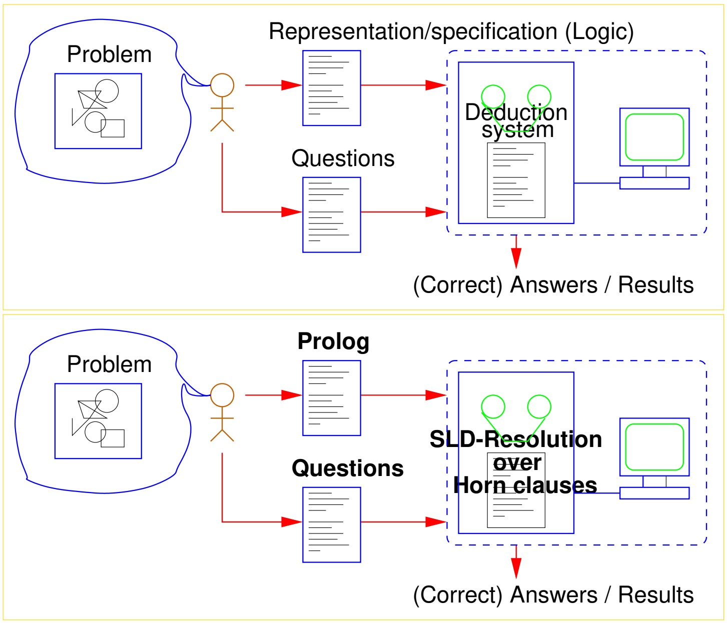

Problem
Representation/specification (Logic)
Deduction system
Questions
(Correct) Answers / Results
Problem
Prolog
SLD-Resolution over Horn clauses
Questions
(Correct) Answers / Results

Figure 1: A motivational view of (C)LP and Prolog: Green's proposal and Prolog as its materialization.

programming language. However, some other Prolog aspects depart from this view. Classic examples that have received much attention are assert/retract and the cut. Our focus here is however on the depth-first search rule: while well-justified by practical concerns, using only depth-first from the start also introduces early on the need to reason about termination, and also possibly a need to use impure features to make different modes produce answers in finite time. Termination of course has to be faced sooner or later by any programmer, but having to do it right at the start can detract from being able to convey early on the vision of Prolog as a declarative language where one can first concentrate on problem specification and/or knowledge representation and only worry later about efficiency.

In the rest of the paper we review a number of ways in which some of these issues can be tackled when teaching Prolog, while still using throughout a Prolog system. The aim is tutorial, in the hope that these ideas can help other instructors that are teaching or plan to teach Prolog.

2. The main message

As we have also argued in [1], perhaps the most important objective during the first steps of teaching Prolog is to succeed in showing the great beauty of the (C)LP paradigm in general and of Prolog in particular. We believe that in order to achieve this, one needs to convey the original inspiration behind the paradigm of being at the same time a specification and knowledge representation language and a practical and highly productive programming language. We find it very useful to explicitly expose students to the ideas put forward by Cordell Green [4]. Green suggested that, provided an effective deduction procedure is available, i.e., a mechanical proof method, inspired at the time by Robinson's recent proposal of the resolution principle [5], a different view of problem solving and computing would be possible (Figure 1, top). In this approach, the automated deduction procedure is programmed once and for all in the computer. Then, for each problem we want to solve, we just need to find a suitable representation for it in logic (i.e., its specification), and, to obtain solutions, we simply ask questions and let the deduction procedure do the rest. Prolog (and (C)LP in general) are the materialization of this idea by Colmerauer and Kowalski [6, 7], where (Figure 1, bottom) the problem representation or specification is written in logic using the Prolog language, and the efficient deduction mechanism is Kowalski and Kuhnen's SLD resolution [8, 9], combined with the practicality brought about by Warren et al.'s Dec-10 Prolog implementation and compilation techniques [10, 11].[1]

[1]Of course, other proof methods such as, e.g., the bottom-up execution of Datalog and ASP systems are also used as a basis for logic programming. Our focus here however is on the Prolog family of (C)LP languages.

```
mother_of(susan, mary).
mother_of(susan, john).
mother_of(mary, michael).

father_of(john, david).

grandmother_of(X,Y) :-
    mother_of(X,Z), mother_of(Z,Y).
grandmother_of(X,Y) :-
    mother_of(X,Z), father_of(Z,Y).
```
run ▶

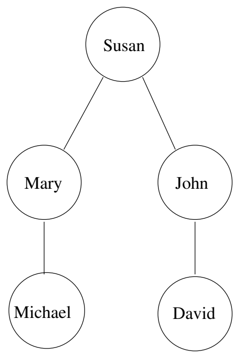


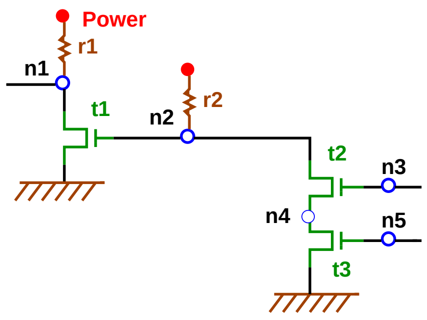
Figure 2: A simple pure logic program for family relationships.

Some "digital circuit theory:"

```
inverter(Input,Output) :-
   transistor(Input,ground,Output), resistor(power,Output).

nand_gate(Input1,Input2,Output) :-
   transistor(Input1,X,Output), transistor(Input2,ground,X),
   resistor(power,Output).

and_gate(Input1,Input2,Output) :-
   nand_gate(Input1,Input2,X), inverter(X, Output).
```



Description of the topology of a circuit:

```
resistor(power,n1).
resistor(power,n2).

transistor(n2,ground,n1).
transistor(n3,n4,n2).
transistor(n5,ground,n4).
```
run ▶

Figure 3: A simple pure logic program for circuit topology.

3. The first simple examples

The vision that with Prolog one can just represent/specify the problem and then obtain answers automatically is relatively easy to convey with the first simple examples such as, e.g., the classical family relationships (Figure 2), circuit topology (Figure 3), simple puzzles, etc.[2] For example, with the family facts and rules of Figure 2 students can get answers to questions in different modes such as `?- mother_of(susan,Y).`, `?- mother_of(X,mary).`, `?- grandmother_of(X,Y).`, etc. Similarly, with the "digital circuit theory" at the top of Figure 3, which describes how inverters, nand-gates, and and-gates are made up of resistors and transistors connected to ground or power, and the concrete circuit at the bottom, described by enumerating the components to which the circuit nodes are attached, students can also get answers to questions in different modes, such as, e.g.:[3]

```
?- and_gate(In1,In2,Out).
```
run ▶

i,e., "is there an and-gate somewhere in this circuit?," and Prolog finds the answer:

```
In1 = n3,
In2 = n5,
Out = n1 ?
```

[2]The level of complexity of these initial examples depends on the background of the students (it is of course not the same to be teaching in schools or to CS students in college). We are using here simple examples that suffice to make the points of discussion.
[3]Clicking on the run ▶ links is perfectly safe!

run ▶
```
mother_of(susan, mary).
mother_of(susan, john).
mother_of(mary, michael).

father_of(john, david).

parent(X,Y) :- mother_of(X,Y).
parent(X,Y) :- father_of(X,Y).

ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).
ancestor(X,Y) :- parent(X,Y).
```

Figure 4: A more involved pure logic program for family relationships.

The issue most relevant to our discussion is that all these queries will return the correct solution and terminate using standard Prolog. I.e., in these examples students can use just the logical reading when writing and reading clauses, and then, assuming that the logic is correct, have confidence that the system will answer correctly any question providing all possible solutions in finite time. However, things can quickly get more complicated.

4. A first source of non-termination

Returning to the family example of Figure 2, consider adding to it the classical ancestor predicate, as in Figure 4. Now for *any* query to `ancestor/2` the program hangs in standard Prolog without giving any answers:

run ▶
```
?- ancestor(X,Y).

<hangs>
```

Of course, we can change the order of the `ancestor/2` clauses and the program will now behave better in standard Prolog. E.g., for:

run ▶
```
?- ancestor(X,Y).
```

We will obtain all the answers:

```
X = susan,
Y = mary ? ;
X = susan,
Y = john ? ;
X = mary,
Y = michael ? ;
X = john,
Y = david ? ;
X = susan,
Y = michael ? ;
X = susan,
Y = david ? ;
<hangs>
```

but the program will still hang at the end. It is an inevitable fact that, sooner or later, students will be confronted with non-termination. It can be hard for them to get around this stumbling block at the early stages of learning the paradigm, and this may lead to disenchantment and/or confusion if no additional tools and understanding of the issues involved are provided.

5. Tabling to the (partial) rescue

Fortunately the technique of *tabling*, pioneered by XSB [12], and now supported by several other modern Prologs [13], including B-Prolog [14], Ciao Prolog [15], SWI Prolog [16], and Yap [17], comes to the rescue for a good number of these cases. Tabling ensures termination of programs for which the "bounded term-size property" holds: programs where all the sizes of subgoals and answers produced during an evaluation are smaller than some fixed number. This is indeed the case for the program of Figure 4, since there are only a finite number of constants in the program and thus any query, subgoal, or answer will be of bounded size. This is also the case for all "Datalog" programs. In order to ensure termination we can declare `ancestor/2` as a *tabled predicate* as follows:[4]

```
:- table ancestor/2.
```

Now we not only get all the answers without having to invert the order of the `ancestor/2` clauses:

```
X = susan,
Y = mary ? ;
X = susan,
Y = john ? ;
X = mary,
Y = michael ? ;
X = john,
Y = david ? ;
X = susan,
Y = michael ? ;
X = susan,
Y = david ? ;
no
```

but the program also ends stating in finite time that there are no more solutions. Note that switching the order of the literals in the recursive clause of `ancestor/2`:

```
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```
run ▶

we obtain all the answers and the program also terminates at the end with standard Prolog. However, this is not always the case, and tabling offers in general a more powerful mechanism than just switching the order of clauses and body literals.

A relevant issue in tabling is which predicates to declare as tabled, which is a subject onto itself, but beyond the scope of our discussion here. However, in the first stages students can simply be told to try declaring different or all predicates as tabled.[5]

Overall, tabling can be very useful in our quest to transmit the beauty of the Prolog language. However, it will obviously not cover the cases which fall outside the bounded term-size property. Unfortunately, this can often happen once nested data structures are introduced and combined with recursion.

6. More complex cases

Consider `natural/1` defining the natural numbers in Peano representation:[6]

[4]In the case of Ciao Prolog, used in our examples, the tabling functionality is loaded with the directive: `:- use_package(tabling).`

[5]In fact, XSB includes an "`:- auto_table.`" declaration for this purpose, which tables automatically enough predicates so that all possible loops have a tabled predicate.

[6]We find Peano arithmetic useful as an instructive, reversible substitute for the non-reversible standard Prolog built-in arithmetic (`is/2`, etc.) in the first parts of the course. An interesting alternative is using constraints. This is well argued in [18] and different constraint systems are available in most modern Prologs. However, using constraints requires the introduction of a 'black box' (the solver, performing constraint propagation, etc.) and we find it convenient, specially at the early stages, that Peano arithmetic does not require anything beyond the standard unification/resolution operational semantics, which has been presented at this stage. In any case, we do mention the existence of both arithmetic constraint domains and `is/2` when first mentioning Peano arithmetic, and sometimes use constraints anyway in early examples. Beyond these considerations, we also find Peano arithmetic motivating and elegant in itself for developing Prolog examples, specially in the first stages of learning to use recursive data types and recursive programs.

```
natural(0).                                                          run ▶
natural(s(X)) :- natural(X).

add(0,Y,Y) :- natural(Y).
add(s(X),Y,s(Z)) :- add(X,Y,Z).

mult(0,Y,0) :- natural(Y).
mult(s(X),Y,Z) :- add(W,Y,Z), mult(X,Y,W).

nat_square(X,Y) :- natural(X), natural(Y), mult(X,X,Y).
```

Figure 5: Defining squares of naturals.

```
natural(0).
natural(s(X)) :- natural(X).
```

and `pair/2` defining a pair of natural numbers:

```
pair(X,Y) :- natural(X), natural(Y).
```

A query such as the following:

```
?- pair(X,Y), X=s(0).                                                run ▶
```

will hang in standard Prolog, because of course the search never progresses beyond `X=0`, since there are infinite possible solutions for `Y` that will be explored first. The initially surprised student may eventually realize that in this case it suffices with simply reversing the order of the literals in the query:[7]

```
?- X=s(0), pair(X,Y).
```

and see that now all the answers are enumerated:

```
X = s(0),
Y = 0 ? ;
X = s(0),
Y = s(0) ? ;
X = s(0),
Y = s(s(0)) ? ;
X = s(0),
Y = s(s(s(0))) ?
...
```

However, consider the more complex program of Figure 5, defining `nat_square(X,Y)` that holds if `X` and `Y` are both naturals and `Y` is equal to `X` multiplied by itself. We have also included the auxiliary predicates defining addition and multiplication. In our classes, we develop each of these predicate definitions with the students by reasoning about the (infinite) set of (ground) facts we want to capture, thereby informally introducing the notion of declarative semantics. We also work with the students on generating the definitions by generalization from tables of facts, thinking inductively, and more. See also [19] for an ample discussion of how to build programs inductively.

Standard Prolog execution provides useful answers for, e.g., mode `nat_square(+,-)`:

```
?- nat_square(s(s(0)), X).                                           run ▶
X = s(s(s(s(0)))) ?
```

an even for mode `nat_square(-,+)`:

```
?- nat_square(X,s(s(s(s(0))))).
X = s(s(0)) ?
```

which invariably nicely surprises students, even if in both cases standard Prolog hangs if one asks for more, non-existent, answers. However, the next obvious temptation is to try:

```
?- nat_square(X,Y).
```

but this time standard Prolog only produces the first, trivial answer and then hangs:

[7] At this point it can also be pointed out that there exist other techniques like delays for changing dynamically the goal order.

```
X = 0,
Y = 0 ? ;
<hangs>
```

despite the fact that there are obviously infinitely many additional valid answers.

Unfortunately, these cases of non-termination cannot be avoided with just tabling, since the program is producing continuously growing Peano numbers, i.e., continuously growing terms. Unfortunately, non-termination issues of this sort are often encountered early on by students –basically as soon as recursive data structures are introduced.

7. Fairness to the (partial) rescue

In order to address these more complex cases, we have found it useful to provide students with a means for selectively switching between depth-first and other search rule(s), including in particular at least one that is *fair*. Ciao Prolog, which has incorporated over time a number of features to facilitate teaching Prolog and (C)LP, supports declarations such as:

```
  :- use_package(sr/bfall).
```

which students can use to specify alternative search rules and run some or all predicates using breadth-first, as well as iterative deepening, etc., in addition to depth-first and tabling. Such alternative search rules are in fact relatively straightforward to implement in any Prolog system, for example via meta-interpretation and/or code expansions.[8]

The relevant fact for our discussion is that if students set their predicates to run in, e.g., breadth-first mode, then all examples will "work well" for all possible queries. This, in the sense that all valid solutions will be found, even if possibly inefficiently. For example, our problematic query:

```
?- nat_square(X,Y).                                                          run ▶
```

will now enumerate the infinite set of correct answers:

```
?- nat_square(X,Y).
X = 0,
Y = 0 ? ;
X = s(0),
Y = s(0) ? ;
X = s(s(0)),
Y = s(s(s(s(0)))) ? ;
X = s(s(s(0))),
Y = s(s(s(s(s(s(s(s(s(0))))))))) ? ;
X = s(s(s(s(0)))),
Y = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(0)))))))))))))))) ?
...
```

Another interesting consequence of using a fair search rule, even for the cases where depth-first works, is that the "quality" in the enumeration of solutions also improves with respect to depth-first. For example, the query:

```
?- mult(X,Y,Z).                                                              run ▶
```

does not hang in standard Prolog. However, it produces the following answers:

[8]In our courses we used to teach the first part of the course, centered around pure LP, with a separate system: a metainterpreter-based standalone system, implemented in Prolog, that ran programs with a breadth-first search rule. However, we eventually found it more didactic and convenient to incorporate the idea of supporting alternative search rules within Ciao Prolog, making use of its facilities for defining language extensions.

```
X = 0,
Y = 0,
Z = 0 ? ;

X = 0,
Y = s(0),
Z = 0 ? ;

X = 0,
Y = s(s(0)),
Z = 0 ? ;

X = 0,
Y = s(s(s(0))),
Z = 0 ? ;
```

```
X = 0,
Y = s(s(s(s(0)))),
Z = 0 ? ;

X = 0,
Y = s(s(s(s(s(0))))),
Z = 0 ? ;

X = 0,
Y = s(s(s(s(s(s(0)))))),
Z = 0 ?

X = 0,
Y = s(s(s(s(s(s(s(0))))))),
Z = 0 ?
...
```

It becomes clear that some answers will never be generated by the depth-first search: it will generate infinitely many `mult(0,X,0)` tuples, with `X` being any Peano number, without ever having a chance to generate tuples where the first and third arguments are different from `0`. In contrast, breadth-first will produce *all* solutions:

```
X = 0,
Y = 0,
Z = 0 ? ;

X = 0,
Y = s(0),
Z = 0 ? ;

X = 0,
Y = s(s(0)),
Z = 0 ? ;

X = 0,
Y = s(s(s(0))),
Z = 0 ? ;

X = s(0),
Y = 0,
Z = 0 ? ;

X = 0,
Y = s(s(s(s(0)))),
Z = 0 ? ;
```

```
X = 0,
Y = s(s(s(s(s(0))))),
Z = 0 ? ;

X = s(0),
Y = s(0),
Z = s(0) ? ;

X = 0,
Y = s(s(s(s(s(s(0)))))),
Z = 0 ? ;

X = s(s(0)),
Y = 0,
Z = 0 ? ;

X = 0,
Y = s(s(s(s(s(s(s(0))))))),
Z = 0 ? ;

X = s(0),
Y = s(s(0)),
Z = s(s(0)) ? ;
...
```

which, in addition, are clearly more informative, satisfying, and useful for subsequent predicates as the solutions obtained using the depth-first search.

In summary, we argue that the use of a fair search rule helps students visualize the true potential of the (C)LP paradigm and Prolog and that it is possible indeed to solve problems by simply thinking logically and/or inductively.

It is interesting to note that this fairness in enumeration also comes in handy in several other areas, such as for example for type definitions. Fig. 6 shows the definition of two types (they are marked as *regular types* by the `regtype` directive): `color/1`, defined as the set of values {`red`, `green`, `blue`}, and `colorlist/1`, representing the infinite set of lists whose elements are of type `color`.[9] In Ciao Prolog marking predicates as types (or in general as properties) allows them to be used in assertions. However, they remain regular predicates, and can be called as any other, used as run-time tests (dynamic checking), "run backwards" to generate examples or test cases, etc. See, e.g., [20] for a recent more complete presentation of these aspects.

For example, calling:

```
?- colorlist(L).
```
run ▶

[9]Fig. 7 shows the same properties of Fig. 6 but written instead using Ciao Prolog's functional notation. The two definitions are equivalent, and thus the answers obtained, functional syntax being just syntactic sugar.

```
:- use_package([assertions,regtypes]).                                   run ▶
% :- use_package(sr/bfall).

:- regtype color/1.
color(red).
color(green).
color(blue).

:- regtype colorlist/1.
colorlist([]).
colorlist([H|T]) :- color(H), colorlist(T).
```

Figure 6: Defining some basic types.

```
:- use_package([fsyntax,assertions,regtypes,sr/bfall]).                   run ▶
:- regtype color/1. color := red | green | blue.
:- regtype colorlist/1. colorlist := [] | [~color|~colorlist].
```

Figure 7: Defining some basic types using functional notation (`fsyntax`).

returns, in depth first:

```
L = [] ? ;
L = [red] ? ;
L = [red,red] ? ;
L = [red,red,red] ? ;
L = [red,red,red,red] ?
...
```

These are obviously correct answers, but not very satisfying for type exploration. However, if we select breadth-first execution we get a much more useful fair generation:

```
L = [] ? ;
L = [red] ? ;
L = [green] ? ;
L = [blue] ? ;
L = [red,red] ? ;
L = [red,green] ? ;
L = [red,blue] ? ;
L = [green,red] ? ;
L = [green,green] ? ;
L = [green,blue] ? ;
L = [blue,red] ? ;
L = [blue,green] ? ;
L = [blue,blue] ? ;
L = [red,red,red] ?
...
```

8. Going from executable specifications to efficient algorithms

With all these tools, the student can also be shown with examples (and through benchmarking them) how in Prolog it is possible to go from executable specifications to efficient algorithms gradually, as needed. One of the examples we use is the modulo operation `mod(X,Y,Z)`, where `Z` is the remainder from dividing `X` by `Y`. A mathematical definition or specification for this operation is:

$$mod(X,Y,Z) \iff \exists Q s.t.\ X = Y * Q + Z \wedge Z < Y$$

This can be expressed *directly* in Prolog as:

```
mod(X,Y,Z) :- less(Z, Y), mult(Y,Q,W), add(W,Z,X).                       run ▶
```

This version is clearly correct (since it is directly the specification) and, using, e.g., breadth-first search, works in multiple directions, always finding all solutions:

```
?- op(500,fy,s).
yes
?- mod(X,Y, s 0).
X = s 0,
Y = s s 0 ? ;
X = s 0,
Y = s s s 0 ? ;
X = s s s 0,
Y = s s 0 ? ;
X = s 0,
Y = s s s s 0 ? ;
...
```

but it can of course be quite inefficient. However, also in Prolog, we can write another version such as this one:

run ▶
```
mod(X,Y,X) :- less(X, Y).
mod(X,Y,Z) :- add(X1,Y,X), mod(X1,Y,Z).
```

which is much more efficient –one can time some queries or reason about the size of the proof trees to show this. Also, this version works well with the default depth-first search rule for several modes. However, the enumeration of solutions is again biased and thus less elegant and useful than the one obtained using breadth-first search.

We do not mean to imply however that the use of a fair search rule or pure programs are a requirement for traveling the path from executable specifications to efficient algorithms. Figure 8 shows an additional example, where we have a specification and two implementations for computing the maximum of a list of numbers: `max/2` on the left is a direct transliteration of the mathematical specification and `max2/2` on the right is a much more efficient, algorithmic implementation. Here both programs can only be used in `max(+,-)` mode because of the use of non-reversible standard Prolog built-ins, but are still quite suitable as an example of writing both the specification and an algorithm in Prolog.

Regarding the examples to show, if only small and simple ones are chosen, sophisticated Prolog implementations running on fast modern computers can also create the misconception that Prolog provides solutions effortlessly. Thus, it is important to eventually also tackle larger and more complex problems and stress that, as a Turing complete language, also Prolog programmers eventually need to care about algorithmic time and memory complexity of both their programs and the libraries/features they utilize, which of course includes, at the limit, termination. Here it is also important to explain at an early stage how to control search, via clause order and literal order, as well as pruning in later stages.

Figure 8: Specification for maximum and two implementations.

run ▶

"Max is the maximum element of a set if there is no element in the set that is larger than it."

$$max(L, Max) \leftarrow Max \in L \ \wedge \ \nexists E \mid E \in L \ \wedge \ E > Max$$

```
max(L,Max) :-
     member(Max,L),
     \+ (member(E,L), E>Max).
```

```
?- max([3,5,2,8,1],Max).
Max = 8
```

```
max2([H|T],Max) :-
     max_(T,H,Max).

max_([],Max,Max).
max_([H|T],TMax,Max) :-
        H > TMax,
        max_(T,H,Max).
max_([H|T],TMax,Max) :-
        H =< TMax,
        max_(T,TMax,Max).
```

```
?- max2([3,5,2,8,1],Max).
Max = 8
```

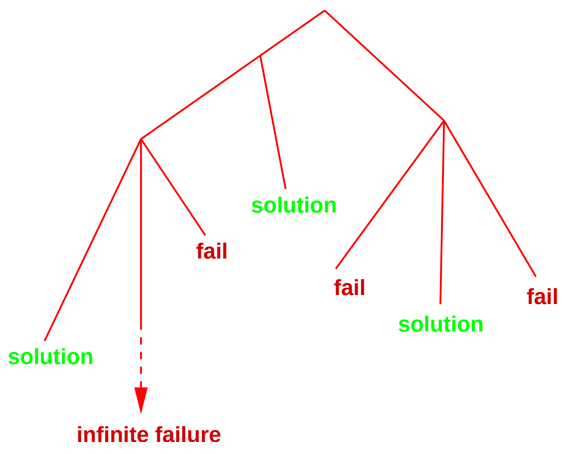


Figure 9: Possible cases in the search tree.

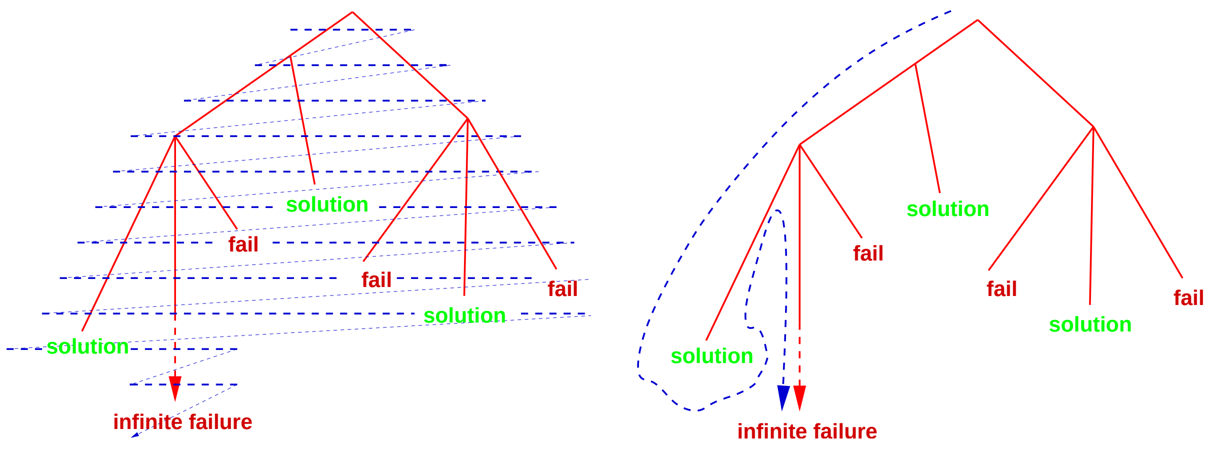


Figure 10: Breadth-first (left) and depth-first (right) exploration of the search tree.

9. Explaining what is really going on

After students have written just a few examples using both depth-first and a fair search rule, they will have observed that the fair search rule does provide all the expected answers, but that execution sometimes hangs after that. It is important thus to explain better what is really happening with their programs, which also implies coming to terms with the realities of non-termination in Turing complete programming languages.

To this end, we have found it very useful to use depictions such as those in Figures 9 and-10 to introduce students in a graphical way to the basic theoretical results at play, i.e., the soundness and (refutation-)completeness of the SLD(NF)-resolution proof procedure used by Prolog.

The idea is to convey that the search tree has in general the shape of Figure 9, i.e., that all solutions and some failures are at finite depth, but that there may be branches leading to failure that are infinite. While techniques such as tabling, as we have seen before, can help detect and avoid traversing some of these infinite failure branches when the bounded term-size property holds, it is not possible to always do so. At this point, it can be interesting to recall or introduce them to the notions of *undecidability* and the *halting problem*, and relate these concepts to the graphical depiction of the search tree. The level of discussion will depend of course on the students' level, but it can always be done informally and adapted to their background. At the same time one should underline that this is of course not a particular problem of Prolog or Logic Programming, but rather the essence of computability, and no language (Prolog, logic, nor any other Turing-complete programming language) or proof system can provide a magic formula that guarantees termination in all cases.

This schematic search tree depiction makes it then easy to explain why breadth-first (or iterative

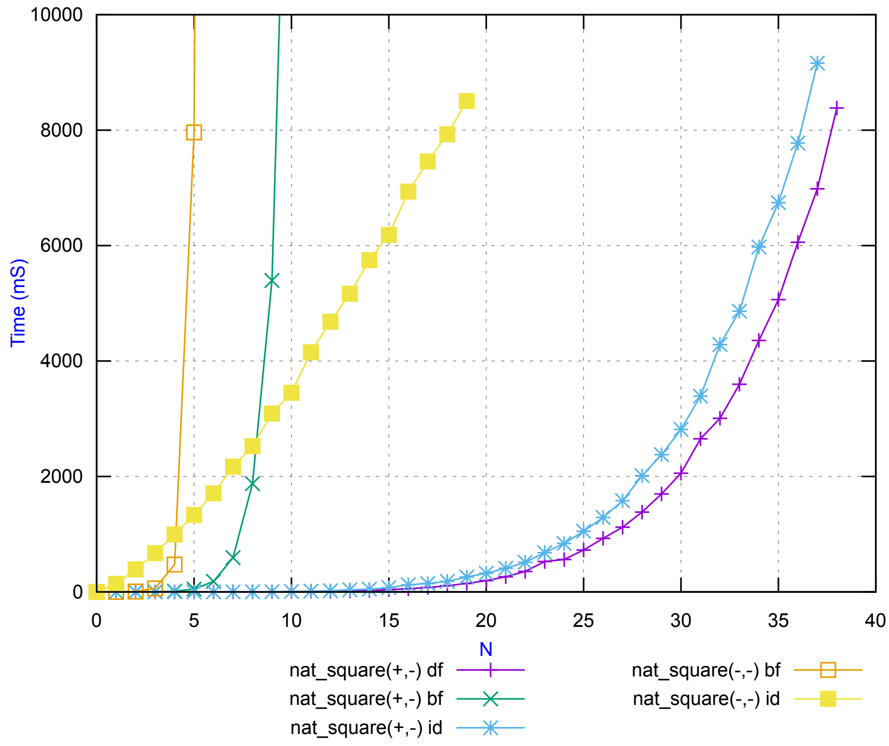


Figure 11: Comparing search rules.

deepening, or any other fair search rule) is guaranteed to produce all solutions if they exist (Figure 10, left). If there is a finite number of solutions, such fair search rules find them in finite time, even if perhaps they do not terminate in some cases after producing all the answers. And it is also easy to explain why depth-first search (Figure 10, right) may sometimes hang before producing some of the answers: those that are in the tree to the right of the leftmost infinite branch.

This is also a good opportunity for introducing and explaining graphically other alternative fair search rules such as iterative deepening and compare them with the students to depth-first search. One can then discuss the advantages and disadvantages of these search rules in terms of time, memory, etc. E.g., that the price to pay for breadth-first execution's advantages is larger memory consumption and execution time, while depth-first can be implemented very efficiently with a stack, with iterative deepening representing an interesting middle ground. This can be shown very practically by benchmarking actual examples, motivating the practical choices made for Prolog, which bring in great efficiency at the (arguably reasonable) price of having to think about the order of query goals, clauses, and body literals.

As an example, Figure 11 shows some results comparing depth-first (df), breadth-first (bf), and iterative deepening (id) for queries to the `nat_square/2` predicate of Figure 5 for two modes: `nat_square(+N,-N2)` and `nat_square(-N,-N2)`, i.e., providing a Peano number `N` and calculating its Peano square, and generating all the pairs of Peano numbers and their squares. For the first mode the graph provides execution times in mS for calls with increasing numbers of `N`, starting from `0`, i.e., `?- nat_square(0,R).`, `?- nat_square(s(0),R).`, etc. For the second mode, the numbers on the y axis are the timings for each one of the answers to `?- nat_square(N,R).`, i.e., the time to get the first values of N and R, then for the second, etc. In this case no curve is given for depth-first, since as we saw it hangs after the first solution.

It is important to note that these numbers are for sub-optimal implementations of the fair rules and that they are for a single problem, so obviously no general conclusions can be drawn and the trade-offs between these search rules are well studied anyway. Here the results are only intended as an illustrative example.

In any case, breadth first search is slower than iterative deepening in most, but not all, cases, and both are slower than depth-first, but obviously only for the cases in which depth-first works. It is also interesting to see that depth-first and iterative deepening are relatively close, as expected.

The important point however is that the fair rules are usable for both (and all) modes, at least for small queries, and can thus be really useful as default rules in the first steps in a teaching environment, which is our concern herein.

10. Final remarks

Our starting point has been the observation that having to deal with the complexities of termination using Prolog's depth-first rule during the first stages of teaching Prolog often detracts from the objective of conveying to students the beauty and usefulness of the language. Motivated by this observation, we have reviewed some ways of addressing these issues in practice, while still using a Prolog system for all stages, such as the use of tabling and fair search rules. If time permits, it is important to also discuss how the constraint systems built into most current Prolog systems (Q, R, fd, ...), can bring about many other improvements to search (e.g., constrain and generate vs. generate and test).

More generally, teaching Prolog and (C)LP is an extremely rich subject and there are of course many other important aspects that we have not been able to address here, including of course all of the traditional subjects of a Prolog course beyond the first steps. For a more complete picture, we have covered previously some of these other issues in [1, 3]; here we have expanded on one of the points we raised there. Much of our experience over the years is materialized in a) the courses that we have developed, for which, as pointed out before, all materials such as slides, Active Logic Documents [3], examples, etc. are publicly available,[10] and b) the many teaching-oriented special features that we have incorporated over time in our own Ciao Prolog system.[11] Here we have touched upon some of these features: being able to choose different search rules and use of the playground from documents for the examples, but there are many others.

We hope that at least some of the ideas presented and these materials and systems are helpful and inspiring to both Prolog instructors that are teaching or plan to teach Prolog and students.

References

[1] M. Hermenegildo, J. Morales, P. Lopez-Garcia, Some Thoughts on How to Teach Prolog, in: D. S. Warren, V. Dahl, T. Eiter, M. Hermenegildo, R. Kowalski, F. Rossi (Eds.), Prolog - The Next 50 Years, number 13900 in LNCS, Springer, 2023, pp. 107–123. URL: http://cliplab.org/papers/TeachingProlog-PrologBook.pdf.

[2] M. R. Genesereth, Prolog as a Knowledge Representation Language – the Nature and Importance of Prolog, in: D. S. Warren, V. Dahl, T. Eiter, M. V. Hermenegildo, R. A. Kowalski, F. Rossi (Eds.), Prolog: The Next 50 Years, volume 13900 of *Lecture Notes in Computer Science*, Springer, 2023, pp. 38–47. URL: https://doi.org/10.1007/978-3-031-35254-6_3. doi:10.1007/978-3-031-35254-6_3.

[3] J. Morales, S. Abreu, D. Ferreiro, M. Hermenegildo, Teaching Prolog with Active Logic Documents, in: D. S. Warren, V. Dahl, T. Eiter, M. Hermenegildo, R. Kowalski, F. Rossi (Eds.), Prolog - The Next 50 Years, number 13900 in LNCS, Springer, 2023, pp. 171–183. URL: http://cliplab.org/papers/ActiveLogicDocuments-PrologBook.pdf.

[4] C. Green, The Application of Theorem Proving to Question-Answering Systems, Ph.D. thesis, Stanford University, 1969.

[5] J. A. Robinson, A Machine Oriented Logic Based on the Resolution Principle, Journal of the ACM 12 (1965) 23–41.

[6] A. Colmerauer, The Birth of Prolog, in: Second History of Programming Languages Conference, ACM SIGPLAN Notices, 1993, pp. 37–52.

[7] R. A. Kowalski, The Early Years of Logic Programming, Communications of the ACM 31 (1988) 38–43.

[10]Our teaching materials can be found at: https://cliplab.org/logalg

[11]See http://ciao-lang.org, http://ciao-lang.org/playground, etc.

[8] R. Kowalski, D. Kuehner, Linear resolution with selection function, Artificial Intelligence 2 (1971) 227–260.

[9] R. A. Kowalski, Predicate Logic as a Programming Language, in: Proceedings IFIPS, 1974, pp. 569–574.

[10] D. Warren, Applied Logic—Its Use and Implementation as Programming Tool, Ph.D. thesis, University of Edinburgh, 1977. Also available as SRI Technical Note 290.

[11] L. Pereira, F. Pereira, D. Warren, User's Guide to DECsystem-10 Prolog, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1978.

[12] T. Swift, D. S. Warren, XSB: Extending Prolog with Tabled Logic Programming, Theory and Practice of Logic Programming 12 (2012) 157–187. doi:10.1017/S1471068411000500.

[13] P. Körner, M. Leuschel, J. Barbosa, V. Santos-Costa, V. Dahl, M. V. Hermenegildo, J. F. Morales, J. Wielemaker, D. Diaz, S. Abreu, G. Ciatto, Fifty Years of Prolog and Beyond, Theory and Practice of Logic Programming, 20th Anniversary Special Issue 22 (2022) 776–858. URL: https://arxiv.org/abs/2201.10816. doi:10.1017/S1471068422000102.

[14] N. Zhou, The Language Features and Architecture of B-Prolog, Theory and Practice of Logic Programming (2012) 189–218.

[15] M. V. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, E. Mera, J. Morales, G. Puebla, An Overview of Ciao and its Design Philosophy, Theory and Practice of Logic Programming 12 (2012) 219–252. URL: http://arxiv.org/abs/1102.5497. doi:10.1017/S1471068411000457.

[16] J. Wielemaker, T. Schrijvers, M. Triska, T. Lager, SWI-Prolog, Theory and Practice of Logic Programming 12 (2012) 67–96. doi:10.1017/S1471068411000494.

[17] V. Santos Costa, R. Rocha, L. Damas, The YAP Prolog system, Theory and Practice of Logic Programming 12 (2012) 5–34. doi:10.1017/S1471068411000512.

[18] M. Triska, U. Neumerkel, J. Wielemaker, Better termination for prolog with constraints, CoRR abs/0903.2168 (2009). URL: http://arxiv.org/abs/0903.2168. arXiv:0903.2168.

[19] D. S. Warren, Writing Correct Prolog Programs, in: D. S. Warren, V. Dahl, T. Eiter, M. Hermenegildo, R. Kowalski, F. Rossi (Eds.), Prolog - The Next 50 Years, number 13900 in LNCS, Springer, 2023.

[20] M. Hermenegildo, J. Morales, P. Lopez-Garcia, M. Carro, Types, modes and so much more – the Prolog way, in: D. S. Warren, V. Dahl, T. Eiter, M. Hermenegildo, R. Kowalski, F. Rossi (Eds.), Prolog - The Next 50 Years, number 13900 in LNCS, Springer, 2023, pp. 23–37. URL: http://cliplab.org/papers/AssertionsAndOther-PrologBook.pdf.