Using Rules to Animate Prolog Programs

Nada Sharaf[1], Slim Abdennadher[1], and Thom Frühwirth[2]

[1] The German University in Cairo, Egypt.
{nada.hamed, slim.abdennadher}@guc.edu.eg
[2] Ulm University, Germany.
thom.fruehwirth@uni-ulm.de

Abstract. The paper provides a methodology to visualize the execution of Prolog programs. Program animation is useful in debugging programs. It could also help beginners to Prolog understand how Prolog works. The provided approach uses Constraint Handling Rules (CHR). The aim of the work is to animate the algorithm implemented by the Prolog program.

Keywords: Animation, Prolog Programs, Constraint Handling Rules

1 Animating Prolog Programs through Rules

The aim of the work is to provide a way to animate the execution of Prolog programs. The focus is however not to show the search tree [1,4]. The work presented in [2,3] showed how Constraint Handling Rules (CHR) could be used as the basis of a generic animation engine. The previous work presented a format for annotation rules that could be used for animation. The idea is to associate every method call/constraint with a graphical object/action. Every time a constraint is added/a method is called, the corresonding graphical object is added/updated. This results in a visual animation of the algorithm execution. The idea is to extend the engine to be used with Prolog programs and data structures as well. Each annotation rule has the format:
$interesting\_ev(Arg_0, \ldots, Arg_m) ==> object = Object\_type\#$
$par_1 = ParV_1\# \ldots \#par_m = ParV_m$ where:

- $interesting\_ev(Arg_0, \ldots, Arg_m)$ represents the constraint/call that should affect the visual state of the animation.
- Each annotation rule is associated with a specific graphical *object* that the user chooses (e.g. circle, line, ... etc).
- Each object is associated with a number of parameters.
- Every value for a parameter $ParV_i$ could have one of these possibilities:
  1. A constant value e.g. green, 20, ... etc.
  2. The function $valueOf(Arg_j)$ that returns the value of Arg_i.
  3. The keyword *random* representing a randomly generated number.

Similar to the previous tool, users could specify the annotations using a graphical user interface. In addition, a list visualization utility is now available to users. In this case, a user can associate a list to be visualized through a specific

object for example a *textual* graphical object. In the case where the list has 3 elements, 3 texts are shown. The parameters of the objects are specified by the user similarly. In addition to the possible values shown above, the user can use the two keywords:

1. *index* to represent the index of the element being visualized.
2. *value* to represent the value of the element being visualized.

2 Examples

This section provides an example of an animation produced using the proposed system. Another example is shown in the appendix

2.1 Sudoku

The following program solves the Sudoku problem for a 4×4 board.
The predicate `maplist1(Predicate,ListArgs,InputList,Output)`: holds in case `Output` is a list such that each element at index i is Out_i such that $Predicate(Arg_i, InputList, Out_i)$.
For example: `maplist1(nth1,[1,2],[3,4,5],Output)` is true when `Output` is bound to `[3,4]`.

```
sudo(L):-
    length(L,16),
    assign_check(L,[],Res).

assign_check([],Acc,Acc).
assign_check([H|T],Acc,Res):-
    member(H,[1,2,3,4]), append(Acc,[H],NAcc),
    check_list(NAcc), assign_check(T,NAcc,Res).

check_list(NAcc):-
            check_rows(NAcc,4), check_columns(NAcc,4),
            check_squares(NAcc).
check_squares(List):-
            IndecesSq1=[1,2,5,6],
            IndecesSq2=[3,4,7,8],
            IndecesSq3=[9,10,13,14],
            IndecesSq4=[11,12,15,16],
            maplist1(nth1,IndecesSq1,List,Sq1), all_diff(Sq1),
            maplist1(nth1,IndecesSq2,List,Sq2), all_diff(Sq2),
            maplist1(nth1,IndecesSq3,List,Sq3), all_diff(Sq3),
            maplist1(nth1,IndecesSq4,List,Sq4), all_diff(Sq4).
check_rows(List,Width):-
            list_to_llists(List,Width,LLists),
            maplist(all_diff,LLists).
```

```
check_columns(List,Width):-
        list_to_llists(List,Width,LLists),
        mtranspose(LLists,Transpose),
        maplist(all_diff,Transpose).
```

In this program, two visual factors could be shown to the user. Firstly, the user should see at every step the board. Secondly, since Prolog is based on a generate and test approach, all generated solutions should be shown the user. In other words, whenever the program tries to add a number to the board, the user should visually see the effect on the board. That way, they could see whether the newly generated board satisfies all the needed conditions or not. In this case, two predicates are annotated. The first one is `sudo/2`. It is annotated with three visual objects:

1. A grid showing the empty board. In the shown example, the width of each square is set to `50`.
2. Two lines separating each 4×4 square.

The initial grid the user visualizes is shown in Figure 2a. The second annotated predicate is `check_list/1`. The annotation associates `check_list(L)` with a list visualization. The user specifies how every element in the list is shown. In this case every element is associated with a textual object where

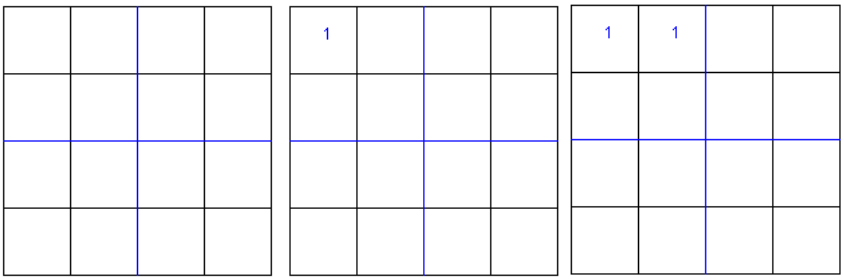
- The x-coordinate of the text is bound to the `(index mod 4)*50+25` of the corresponding element.
- The y-coordinate is bound to the value of `(index/4)*50+25`

3 Conclusions and Future Work

In conclusion, the tool provided a methodology to provide animations for Prolog programs. The animations show the algorithmic parts of the program rather than the search tree and its aspects. In the future, conditional predicate annotation should be offered. In addition, on backtracking more, options should be provided to the user such as removing some visual aspect of the animation.

References

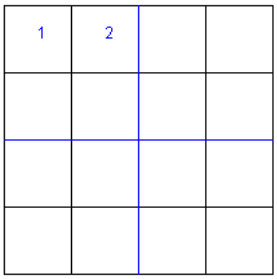
1. Marco Gavanelli. Sldnf-draw: Visualization of prolog operational semantics in latex. *Intelligenza Artificiale*, 11(1):81–92, 2017.
2. Nada Sharaf, Slim Abdennadher, and Thom W. Frühwirth. CHRAnimation: An Animation Tool for Constraint Handling Rules. In *Logic-Based Program Synthesis and Transformation - 24th International Symposium, LOPSTR 2014, Canterbury, UK, September 9-11, 2014.*, volume 8981 of *Lecture Notes in Computer Science*, pages 92–110. Springer, 2014.



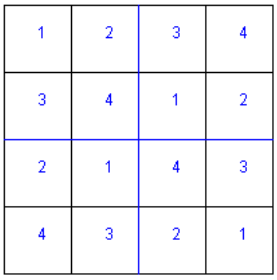
(a) The initial grid

(b) Visualizing the list {1}

(c) Visualizing the list {1,1}



(d) Backtracking and visualizing the list {1, 2}



(e) Visualizing the list {1,2,3,4,3,4,1,2,2,1,4,3,4,3,2,1}

Fig. 1: Animating the Sudoku Generation Prolog Program

3. Nada Sharaf, Slim Abdennadher, and Thom W. Frühwirth. A rule-based approach for animating java algorithms. In Ebad Banissi, Mark W. McK. Bannatyne, Fatma Bouali, Remo Burkhard, John Counsell, Urska Cvek, Martin J. Eppler, Georges G. Grinstein, Weidong Huang, Sebastian Kernbach, Chun-Cheng Lin, Feng Lin, Francis T. Marchese, Chi Man Pun, Muhammad Sarfraz, Marjan Trutschl, Anna Ursyn, Gilles Venturini, Theodor G. Wyeld, and Jian J. Zhang, editors, *20th International Conference Information Visualisation, IV 2016, Lisbon, Portugal, July 19-22, 2016*, pages 141–145. IEEE Computer Society, 2016.
4. Maxim Shishmarev, Christopher Mears, Guido Tack, and Maria Garcia de la Banda. Visual search tree profiling. *Constraints*, 21(1):77–94, 2016.

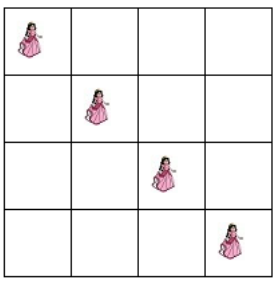
Appendix A N-queens

The n-queens problem aims at placing n queens in a $n \times n$ grid such that no two queens could attack each other. Two queens could attach each other if they are placed in the same row, column or diagonal. A Prolog solution to this problem is shown below:

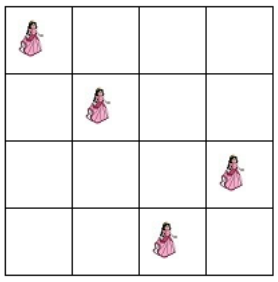
```
formList(S,S,[S]).
formList(S,E,[S|R]):- S<E, S1 is S + 1, formList(S1,E,R).

nqueens(R,N):-
                          length(L,N), formList(1,N,L),
                          formList(1,N,Rows),
                          formList(1,N,Columns),
                          permutation(L,R), \+attack(R,N).
attack(L,N):- check_list_c(L),attack_h(1,L,L).
attack_h(Row,[H|Tail],L):-
                          nth1(Index,L,Col2),
                          Row2 = Index, Row2\=Row,
                          (Index-Col2) =:= (Row-H),!.
attack_h(Row,[H|Tail],L):-
                          nth1(Index,L,Col2),
                          Row2 = Index, Row2\=Row,
                          (Index+Col2) =:= (Row+H),!.
attack_h(Row,[H|Tail],L):-
                          NRow is Row + 1, attack_h(NRow,Tail,L).
```

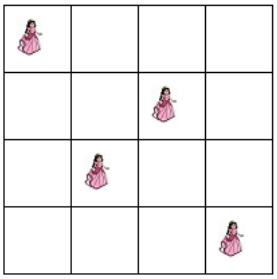
Similarly, the predicate `nqueens/2` is associated with the visual grid. The predicate `permutation/2` is associated with a list visualization. Every element in the list is shown as an image object. The image shows a queen to the user. Users could thus visually see where the queens are placed to determine whether the configuration is a correct one or not.



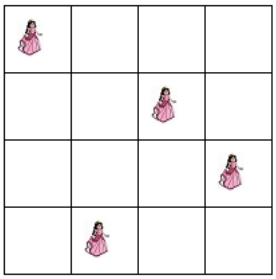
(a) Visualizing the list {1,2,3,4}



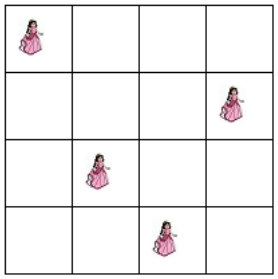
(b) Visualizing the list {1,2,4,3}



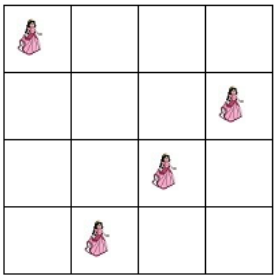
(c) Visualizing the list {1,3,2,4}



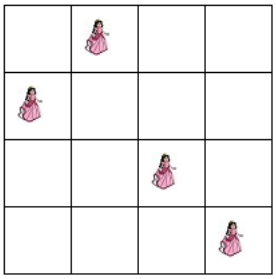
(d) Backtracing and visualizing the list {1, 3,4,2}



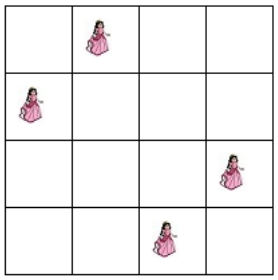
(e) Visualizing the list {1,4,2,3}



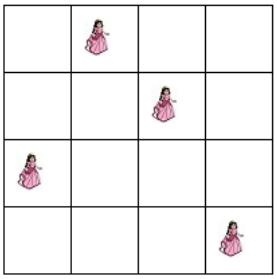
(f) Visualizing the list {1,4,3,2}



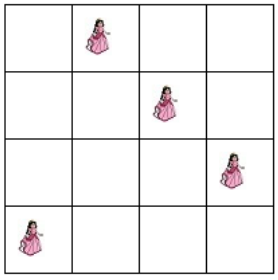
(g) Visualizing the list {2,1,3,4}



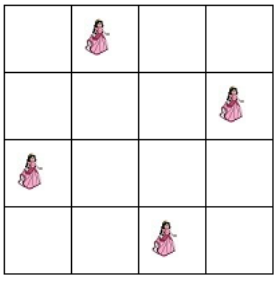
(h) Visualizing the list {2,1,4,3}



(i) Visualizing the list {2,3,1,4}



(j) Visualizing the list {2,3,4,1}



(k) Visualizing the list {2,4,1,3}

Fig. 2: Animating the N-queens Prolog Program