An Alternative Logic Programming Execution Model for Graphical Debugging

Diego Loyola
e-mail: loyola@dlrtcs.da.op.dlr.de

Abstract
The facilities provided by existing debugging tools are insufficient to give a global program execution image and do not enable a detailed program behavior analysis giving access to relevant information in a quick and easy way. In this work a Prolog execution model for debuggers that overcome these problems is presented. The developed model is based on an extended execution tree that represents successful or failed derivations. Each node of the tree has derivation information associated suitable to solve it.
The user can easily find a program mistake making use of the information provided in each node and the facilities of going directly to erroneous nodes skipping the derivation of correct or uninteresting parts.

Keywords
Logic Programming Execution Model, Trace-Debugging, Animation.

Introduction
The current work presents an execution model for the development of system debuggers where the user takes a more active position in the curse of debugging logic programs, and provides a visual feedback of the inspected program.

The computation of a logic program can be seen as a sequence of non-deterministic goal reductions or as a proof tree [1] where nodes are goals that occur in the computation and arcs represent the relation of goal invocations.

```
l1: q(X,Y) ← p(X,Y).
l2: q(X,Y) ← p(X,Z), q(Z,Y).
l3: p(a b) ← □.
l4: p(b,c) ← □.
```

Figure 1. Program example

Figure 2 shows a derivation and Figure 3 a proof tree (indentation represent the arcs) corresponding to the program example in Figure 1.

```
<q(a,Y); q(X0,Y0)←p(X0,Z0),q(Z0,Y0).; {X0→a,Y0→Y}>
<p(a,Z0),q(Z0,Y); p(a,b)←□.; {Z0→b}>
<q(b,Y); q(X2,Y2)←p(X2,Y2).; {X2→b,Y2→Y}>
<p(b,Y); p(b,c)←□.; {Y→c}>
<□, □, { }>
```

Figure 2. Example of a derivation

```
q(a,c)
        p(a,b)
        q(b,c)
                p(b,c)
```

Figure 3. Example of a proof tree

The proof tree is a more intuitive way of describing a computation, but as it can be seen, it is a less complete one. Therefore, an execution model that combines the expressiveness of proof trees and contains all the information of a derivation was developed. The model is based on an extended execution tree where each node has associated derivation information suitable to solve it. Trees corresponding to failed derivations are also obtained in order to analyze the causes that makes a goal not derivable.

Execution Model

The clauses in the program are labeled so that it can be identified which clause is used in each step of the derivation. A numeration on the tree nodes that identifies where a substitution is made and helps to find backtracking points is defined. A sign '+' or '–' is associated to each node indicating a successfully or failed goal derivation.
The behavior of the standard Prolog systems is obtained fixing the computation and search rules such that it is selected the leftmost atom in a goal making a depth–first travel.

The formal definition of the execution model is as follows:

A *labeled clause* is a universally quantified logical sentence of the form

$$L:\ A \leftarrow B_1, \ldots, B_k. \qquad k >= 0.$$

where L is the clause unique identifier, the A and the Bs are logical atoms. A is the clause head and the Bs the body.

A *goal* is a sentence of the form

$$\leftarrow B_1, \ldots, B_k \qquad k >= 0.$$

A *labeled logic program* is a finite set of labeled clauses.

The *extended execution tree* τ of an unique atom goal G from a labeled logic program P, node number N in N^+ and proof of sign S in $\{+, -\}$ is defined as:
(a) Let $l: A \leftarrow \square.$ be a labeled clause in P such that A and G are unifiable via a mgu θ. The extended execution tree of G from P clause l, node number N and sign '+' is

$$\tau_{G\theta l N+} = \langle G\theta, \theta, l, N, +, \{ \langle \text{success}, \phi, l_s, N_s, +, \{\ \} \rangle \} \rangle .$$

where l_s and success do not appear in P, ϕ is the empty substitution and $N_s = N.0$.

(b) Let $l: A \leftarrow B_1, \ldots, B_k.\ k > 0$ be a labeled clause in P such that A and G are unifiable via a mgu θ. The extended execution tree of G from P clause l, node number N and sign S is

$$\tau_{G\alpha lNS} = \langle G\alpha, \delta, l, N, S, \{\tau_{C_1\theta_1 1_1 N_1 S_1}, \ldots, \tau_{C_k\theta_k l_k N_k S_k}\}\rangle .$$

where $C_1 = B_1\theta$, $C_i = B_i\theta\theta_1 \ldots \theta_{i-1}$ for $i = 2,\ldots,k$, $\delta = \theta\theta_1 \ldots \theta_k$, $\alpha = \delta$ restricted to the variables that appear in G, $Ni = N.i$ for $i = 1,\ldots,k$ and S is '+' if S_i is '+' for $i = 1,\ldots,k$ or '−' otherwise.

(c) If for all labeled clauses A in P, A and G can not be unified, the extended execution tree of G from P clause l_f, node number N and sign '−' is

$$\tau_{G\phi l N-} = \langle G, \phi, l, N, -, \{\langle fail, \phi, l_f, N_f, -, \{\ \}\rangle\}\rangle .$$

where l_f and fail do not appear in P, ϕ is the empty substitution and $N_f = N.0$.

NOTE: A multiple atom goal $\leftarrow G_1, \ldots, G_k$. can be defined as the labeled clause $l_j: G \leftarrow G_1, \ldots, G_k$. where l_j and G do not appear in P.

In figure 4 the extended execution tree of q(a,Y) for the program in Figure 1 and root node number 1 is shown.

```
<q(a,Y){Y → c}, {X0→a,Y0→Y},l2,1,+,{
        <p(a,Z0){Z0 → b},{ },l3,1.1,+,{
                <success,{ },ls,1.1.0,+,{ }>}>
        <q(b,Y){Y→c},{X2→b,Y2→Y},l1,1.2,+,{
                <p(b,Y){Y→c},{ },l4,1.2.1,+,{
                        <success,{},ls,1.2.1.0,+,{ }>}>}>
```

Figure 4. Example of a extended execution tree

The enriched tree represents successful or failed derivations and has a node for each goal generated by the computation. Additional information is associated to the node: the clause used to solve the goal, variable bindings before and after the derivation and information on whether the node derivation was successful or not.

Debugging

The enriched tree represents successful or failed derivations and has a node for each goal generated by the computation. The following information is associated to the nodes: clause used to solve the goal, variable bindings before and after the derivation (substitution), a numeration and sign of the node derivation. The two debugging alternatives based on the defined execution model were explored: stepwise and graphical debugging.

Stepwise Debugging

The tree is stepwise displayed using menus and the navigation is made for levels, i.e. when a node p is selected, the system shows it and its direct descendents $p_1, \ldots, p_k$. The goals are pressed by '+' or '–' sign if their respective proofs are success or fail. See figure 5. In each moment the user can decide to see the information associated to p (zoom), to go to next level selecting a p_i i = 1...k or to return to the previous level. Finally the user has the possibility of analyzing the next extended execution tree.

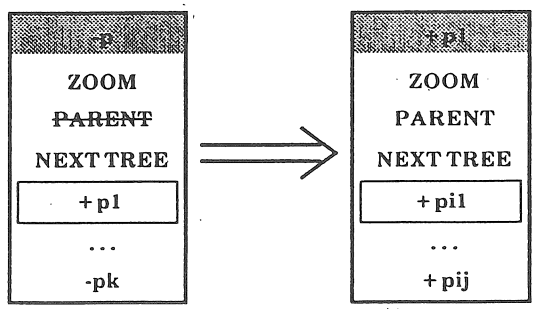


Figure 5. Stepwise Debugging

The developed execution model was implemented using the meta–level reasoning facilities provided for the existing logic languages. A meta–interpreter computes the extended execution tree while solving a goal.

Graphical Debugging

The tree is shown graphically without any additional work from the user. Nodes that can not be proved are marked with a color darker than that of those that could be derived. The user can select any node in order to examine it.

Figure 6 shows a failed derivation of the goal q(a,Y) in the program shown in Figure 1.

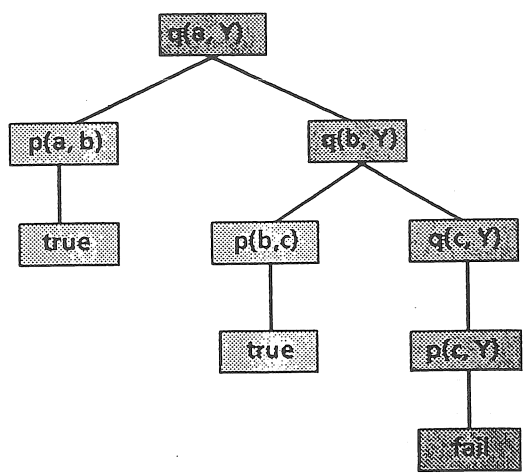


Figure 6. Animation and Graphical Debugging

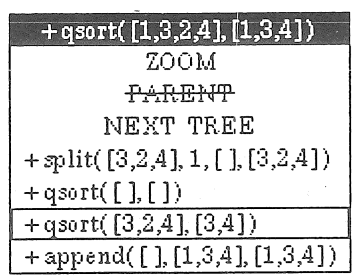
The navigation in the tree is made with commands (key strokes) that change the "*current*" node, moving to nodes in the same level or in different ones. There are scrolling facilities that enables the navigation in "big" trees. In each node the user can ask the system for additional information (zooming).
This graphical debugger is part of a pure Prolog environment based in the developed execution model.

Debugging example
This navigation facilities and the information associated to each node allows the user to have a clear idea of the actual program behavior, and to identify and reach directly the suspicious nodes. In the following example the develped tools [3] are used to find the problem in the buggy quicksort shown in Figure 7

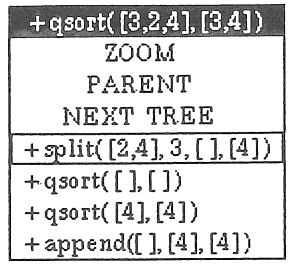
```
qsort( [X | Xs], Result)  ←
          split(Xs, X, Lo, Hi),
          qsort(Lo, ResultLo),
          qsort(Hi, ResultHi),
          append(ResultLo, [X | ResultHi], Result).
qsort( [ ], [ ]).
split( [X | Xs], Crit, Lo, [X | Hi])  ←
          X >= Crit,
          split(Xs, Crit, Lo, Hi).
split( [X | Xs], Crit, Lo, Hi)  ←
          X < Crit,
          split(Xs, Crit, Lo, Hi).
split( [ ], Crit, [ ], [ ]).
append( [X | Xs], Ys, [X | Zs])  ←
          append(Xs, Ys, Zs).
append( [ ], Ys,Ys).
```

Figure 7. Buggy quicksort

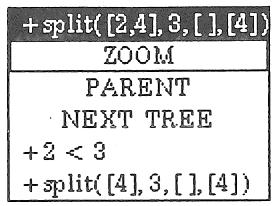
With the goal qsort([1,3,2,4],X) the system displays the following menu



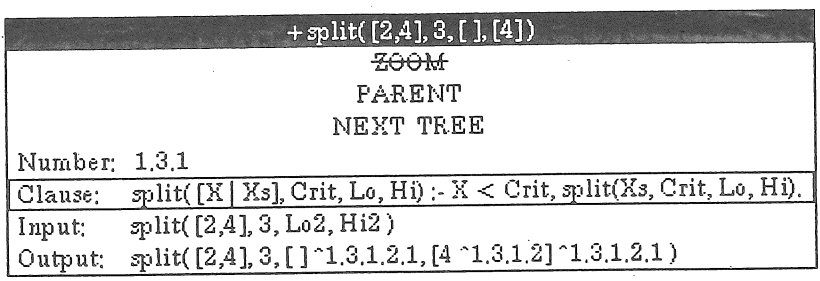
The root goal is solved with an erroneous result X=[1,3,4], the element "2" is lost. The nodes used in its derivation +split([3,2,4],1,[ ],[3,2,4]) and +qsort([ ],[ ]) are correct. The second element in +qsort([3,2,4],[3,4]) is lost. Selecting this qsort node we obtain the derivation



The element is lost in +split([2,4],3,[ ],[4]). Its derivation is



The nodes +2<3 and +split([4],3,[ ],[4]) are correct ... the problem must be in +split([2,4],3,[ ],[4]) self. Zooming this node



The mistake is detected: the element "X" in the clause used to solve the predicate split is not added to "Lo" as it should be.
In this example, the user has had to direct the debugger only three times to get the error. More user interactions and time are needed in order to find the bug using the traditional debugging methods.

Conclusions

The proposed execution model enables a debugging methodology that solves the main problems found in the traditional trace packages (for example the procedure box model) and it is more intuitive that the TPM model [2].

A detailed analysis of Prolog program behavior is provided using the extended execution tree. The developed tool leaves the notion of desirability of program behavior up to the user during debugging. He/she decides which are the "suspected" nodes and can directly reach them.

Stepwise and graphical debugging alternatives were developed. The information provided in each node and a flexible navigation through the extended tree enables the user to go directly to erroneous nodes skipping the derivation of correct or uninteresting parts, and to drive the debugging process only a few times as was verify in different test sessions.
The main facilities provided by the developed tool are the possibility of:

- Giving a visual feedback of program execution.
- Having a flexible navigation through the tree.
- Knowing wether the node derivation was successful or not.
- Identifying the clause used to solve a goal.
- Knowing variable binding before and after the derivation.
- Identifying the node where a variable is instantiated.

The visualization of the execution tree can also be useful for educational purposes and as a tool for Prolog programs debugging.

Future work will be concern with the graphical representation of "big" trees (giving a global view while enabling to concentrate the attention in a specific subtree) and an automatically empirical analysis of the extended execution tree looking for 'bug patterns' that identify the possible cause of the program failure.

Bibliography:

[1] JW. Lloyd (ed), Computational logic, Brussels; spring–verlag; 1990
[2] H. Eisenstadt, The Transparent PROLOG Machine (TPM): an execution model and graphical debugger for logic programming, Journal of Logic Programming, Vol. 5 No. 4 pages 277–342, 1989
[3] D. Loyola, Extending Execution Trees for Debugging and Animation in Logic Programming, Lecture Notes in Computer Science, No. 528, 1991