Learning Logic Programming with ART

Hans van Ditmarsch

When teaching logic, proof visualizations are helpful. Different successful proposals have been made to visually represent logic programming proof concepts such as resolution step (single derivation step), derivation (sequence of resolution steps), refutation tree (represents a set of derivations), and program execution (systematic search in a set of derivations). Such proposals include more or less ‘one-dimensional’ structured sequences of annotated formulas, and ‘two-dimensional’ representations such as refutation trees. We focus on refutation trees and also present some different visualizations. We apologize for taking all logic programming notions and concepts for granted, see [3] for details, of which this submission is an extended abstract.

Nilsson Nilsson introduces the refutation graph in [9]. For an example, see the right tree in Figure 1. A clause is written as a set of positive or negative literals, and not as a disjunction. The empty clause is denoted by NIL. A resolution step is visualized by two arcs linking a lower node with two higher level nodes. The initial clauses are on the upper level of the tree, and the resolvents at (in general, arbitrarily) lower levels. This visualization can be used for arbitrary resolution procedures.

Apt Apt [1, p.850] introduced the SLD tree, or refutation tree, as a concise representation of all SLD-resolution derivations given a goal, a logic program and a computation rule. For an example, see the left tree in Figure 1. The top node is the initial goal. The branches are labelled with the numbers (i, ii, iii) of the selected program clauses—the corresponding program is given in the caption of the figure—and with the substitutions necessary for unification. The left-to-right order of the daughters of a node has no meaning (as in Nilsson)—as the tree is relative to a set of clauses, not a sequence of clauses. Similar examples of refutation trees are found in [7, pp.55-66], [6], and [11, pp.110–113].

Byrd Box The program execution trace is a common way to represent procedural information, originating in a proposal by L. Byrd, see [10]. ‘Box’
Figure 1: Three ways of computing an answer to the query $\leftarrow P(x)$ given the logic program consisting of clauses $P(x) \leftarrow Q(x)$ (i), $Q(a) \leftarrow$ (ii), and $Q(b) \leftarrow$ (iii).

suggests how the Prolog interpreter handles calls (CALL) to satisfy predicates successfully (EXIT) or unsuccessfully (FAIL), including the backtracking mechanism (RETRY) which corresponds to backtracking in a refutation tree. For the program in Figure 1, we would obtain the result: CALL $P(x)$, CALL $Q(x)$, EXIT $Q(a)$, EXIT $P(a)$. In case the call for $Q(x)$ needs to be reconsidered, we would see RETRY $Q(x)$, after which there is another EXIT $Q(b)$. Mulholland studied the variety and use of trace facilities [8].

Eisenstadt Visualizing logic program execution has been an ongoing research effort at the British Open University since the 1980s. This resulted in the Transparent Prolog Machine (TPM) [4], in which Prolog program execution can be visualized at various levels of detail. The most elaborate detail is called an AORTA (AND-OR tree augmented) diagram. The middle ‘trees’ in Figure 1 gives an example. ‘Tick’ means ‘succeed’, ‘tick-cross’ means ‘fail on retry’. An AORTA diagram corresponds to a subtree of an SLD tree resulting from depth-first expansion of the root up to completion of a success branch (refutation). It combines features of the refutation tree with those of the Byrd Box.

ART Slightly different from Apt’s SLD tree, we now let the left-to-right order of the branches in a refutation tree correspond to the order in which matching clauses are found in a logic program, seen as a sequence of program lines, as in Prolog. With that restriction, the form of the tree is completely determined (we cannot swap branches anymore). The process can also be reversed: given a refutation tree where leaf nodes are labelled with either SUCCEED (for the empty clause) or FAIL, and all other nodes and all links are not labeled, i.e., an abstract refutation tree or ART, one can construct a program, a computation rule and a goal with this shape as its refutation tree. Thus we can introduce in a visual way basic logic
programming concepts such as resolution strategies and finite failure.

For an example, consider the logic program specifying addition that consists of clauses $Plus(0, x, x)$ (i) and $Plus(Sx, y, Sz) \leftarrow Plus(x, y, z)$ (ii), the goal (query) $\leftarrow Plus(x, y, SS0)$ corresponding to the addition $x + y = 2$, and the computation rule ‘select first subgoal’. Its refutation tree is depicted in Figure 2.

The abstract refutation tree, or ART, for this tree is depicted in the middle of Figure 2. We construct this ART by replacing leaf node labels ‘empty clause’ (for ‘successful refutation’) by SUCCEED, other leaves (not in this Figure) by FAIL, and omitting all other node and branch labels. So ‘abstract’ means ‘containing less information’. A branch terminating in SUCCEED corresponds to a refutation. A branch terminating in FAIL and an infinite branch correspond to failed derivations.

Many refutation trees have the same ART. We now reverse this procedure, and construct a program and goal and a computation rule for a given ART. For finite trees this turns out to be rather trivial, though quite instructive. A simple procedure is given in [3]. The program resulting from that procedure when applied to the ART in Figure 2 is: $p_1 \leftarrow p_2$, $p_2 \leftarrow p_3$, $p_3 \leftarrow$. The clause order in this program corresponds to the arc labels (i, ii, ...) in the right refutation tree in Figure 2. For trees with infinite branches the situation is more complex. Additionally, one can think of ways to represent loops in ART.

**ART in teaching**  The British Open University course *Intensive Prolog*
[5] uses AORTA diagrams as a didactic tool. We co-authored the adaptation of that course for the Open University of the Netherlands [13]. Additionally, we devised the ART while teaching logic programming in the period 1994-1999, at the University of Groningen, for both Cognitive Science and Computer Science audiences. This resulted in a study guide [2] additional to The Art of Prolog [11], developed within the framework of a major Dutch project for courseware development [12]. Plans to develop software for building ART have not yet been realized—we used it on the blackboard, and interactive with students in class. We think that this was already quite helpful for students to understand the refutation proof procedure.

References