Rule Trees

A Execution Mechanism for Time-ordered Forward Deduction Evaluation Strategy of Starlog

Zhonge Xiao
John Cleary
Brian Unger

January 12, 1995

Contents

1 Introduction **2**

2 Basic Execution Control Algorithm **5**

3 R-tree and Its Execution **7**
3.1 R-tree Structure and Properties 7
3.2 R-tree Execution 10

4 Modified R-trees **13**

5 Identify Indirect Primitives **15**

6 Conclusion **16**

7 Acknowledgments **16**

Abstract

This is the second report on "Time-ordered Forward Deduction' algorithm for Starlog. A ***Rule Tree (R-tree)*** *structure is discussed. The R-tree is used to represent a program rule in the dynamic process of program execution. It serves not only as a structured data storage for program rules and their derivable instances, but also as a control structure in program execution which allows the execution to be carried out in more orderly and formal manner.*

1 Introduction

This is the second technical report on a novel evaluation method devised and implemented for Starlog, a temporal pure logic language with time, logical arithmetic and negation. The first report is mainly about the new strategy of evaluation and the principle algorithm. This current report describes the execution mechanism and techniques adopted to implement it. While the report is focused on the basic method of the execution mechanism, it does cover some implementation details to the extent that makes the description understandable.

The basic algorithm of Starlog execution, as described in the first report, is based on two fundamental ideas, *bottom-up*, or *forward deduction* as is called here, and *time-ordering*. With *bottom-up* the deduction process for a program rule can be activated only when there are tuples that have not been used. Tuples are used as input to clause instances which contain at least one atom in their body that calls for the same predicate as that of the input tuple. New clause instances, if derivable, are derived as result of the input of a tuple. The original clauses (rules) are reduced in each step of deduction to more specialized (or further instantiated) instances. Eventually, some of these instances will be reduced to new tuples and others will fail. A question that follows naturally is that in which order these tuples are to be used. With *time-ordering* it requires schedule the use of tuples in their time order so that tuples will also be generated in time order, this is essential to the unique negation evaluation strategy of Starlog. Given this basic algorithm, the execution mechanism can be formed around a simple scheduling strategy which will be discussed in section 2.

The execution mechanism is also organized around a data structure called **Rule Tree**, or in brief **R-tree**. R-tree is a tree structure and is used to keep for each **rule** of a program the records of **original clause** and all clause instances derivable from it during the process of execution except those that have already been reduced to tuples. Here *rules* refer to **original clauses** of a program, not their derivable *clause instances*. R-tree has some desirable properties. Among these properties is that the shape or structure of the R-tree of a rule depends only on the number of **user-defined predicates** that appear in the body of a rule, which has made it possible to implement a simple and uniform execution procedure. With R-tree, it is also possible to eliminate searching for applicable clauses, before a matching clause is found and unification procedure can be applied. There is, however, some limit to the number of user-defined predicates in the same rule body for traversing a R-tree effectively without wasting too much time in visiting empty nodes. Section 3 gives an in-depth coverage on the R-trees.

While the basic execution mechanism adopts a *bottom-up forward deduction* method, the evaluation of negations uses somewhat more *"conventional"* control strategy. That is the interpreter tries to satisfy as goals the user-defined atoms in a negation body from right to left with and only with all existing tuples that could possibly satisfy these goals. The reason that a diversified instead of a uniform approach is adopted is obvious. The evaluation of negations is to prove *refutationally*, by trying to satisfy each goal in its body, that negation fails. In other words, it is a process of satisfying goals just like solve a Prolog query, it is not a process of generating tuples as what Starlog's basic execution aims at, so the most natural and more efficient way of negation evaluation is "*top-down*", starting from goals instead of starting from tuples. It differs, however, from the conventional goal satisfaction in that only tuples are selected for satisfying goals as the purpose of evaluation is to verify the truth or falsity of a negation and the time ordering nature of the algorithm requires that a definite result for up to the **Current Time** be made based on all possible **facts** (**tuples**) up until the Current Time. In some sense this can be viewed as a bottom-up approach.

Another approach for negation evaluation, which has not been implemented yet, is a single uniform execution mechanism for both basic execution and negation evaluation based on *time-ordered forward deduction* algorithm. The algorithm for negation evaluation has to be modified by introducing *negation* *rewrite*. That is for each **negated rule** of a program a additional rule, which defines the *negation predicate* of that negated rule, is created by the interpreter. The new rule, called **negation rule**, takes all the free variables in original negation part as arguments to its head and the negation body as its body. The negation part of the original negated rule will be rewritten as consisting of merely a single atom (**negated atom**) which calls on the negation predicate. The most part of negation evaluation could therefore become the same as the execution of **non-negated rules** and the evaluation of negated rules becomes simply to prove that there is no such **negation tuples** that could possibly satisfy the negated goal. When a instance of the **negated rule** is being evaluated, the variables in its positive part can then be passed to the negation tuples derived from its negation rules and if the unification succeeds then the negation is failed and no tuple could be drawn from this negated rule instance.

The evaluation of negations involves, as discussed in first report, such components as goals satisfaction, paths analysis, critical phase execution and re-evaluation. And It is the negation evaluation mechanism that is mainly responsible for the efficiency of the Starlog interpreter. The techniques adopted for these components in current implementation are beyond the scope of this report.

2 Basic Execution Control Algorithm

The execution control scheme is straightforward. Based on *bottom-up* strategy, it uses strictly time-ordered scheduling of tuple execution and negation evaluation. Two working list structures (**tuple list** and **pending list**) play central roles in this scheduling mechanism.

tuple list a list that keeps all the tuples, initially existing or generated during the execution of a Starlog program, in non-decreasing time order. Each member of the list consists of the **tuple** and its **time tag**. The time tag of a tuple equals to its time stamp value (lower boundary if of interval value). The list is ordered by time tag. The tuple list grows as new tuples with later times are generated, and obsolete tuples with earlier time are collected away from time to time.

pending list a list that keeps all the tuples with negation, in non-decreasing time order (same as tuple list). Each member of the list consists of the **tuple**, its **negative part**, its **time stamp**, and some other useful information for optimized evaluation of negation. Newly generated pending tuples are added to the list and resolved tuples are taken away from the head of the list.

Execution starts with initial tuples in the tuple list. Each step of execution involves only one tuple, called **Current Tuple** or **Input Tuple**. It is used as input to all clauses that contain in their body at least one atom with the same predicate symbol as that of the Current Tuple. The *input* of a tuple for a clause may result in a new clause instance or a new tuple (if there are no more user-defined atoms left in the body of the clause instance). New clause instances will be kept in R-trees as will be discussed in next section, new tuples are put into tuple list in correct time order.

A global variable called **Current Time** (`Tc`) is set to the time value of the Current Tuple or the lower time boundary if the time is of interval value. Before the Current Time advances to a later time value (the time stamp of next un-used tuple in the tuple list, also called **Next Time**), the pending list are checked, if there are any pending tuples that have time values less than Next Time, their negations are evaluated and new tuples may be drawn

from them depending on the result of evaluation. The execution proceeds through both **tuple list** and **pending list** until all the tuples in both lists are exhausted.
Following is the basic control algorithm.

```
/* Tn: next time
/* Tp: time of pending tuple
/* Tc: current time
begin
    while tuple-list or pending-list not empty
       begin
         if there is a un-used tuple in tuple list
            begin
              execution input-tuple;
              while Tp < Tn
                evaluate negation;
            end;
            while Tp < Tn
              evaluate negation;
         set Tc and Tn;
       end;
end;
```

It can be seen that with this control algorithm, the negation part of a pending tuple is evaluated only after all the tuples with earlier time than the time stamp of the pending tuple are generated so that the negation can be evaluated against all possible tuples with earlier time stamp and the correctness of evaluation can be guaranteed for up until `Tc`. It is also true that no more tuples with time stamp earlier than `Tc` could possibly be generated, in other words, all the tuples with time stamp earlier than `Tc` have been generated because all the pending tuples with earlier time stamp than `Tc` have been evaluated and tuples that could be drawn from these pending ones have also been generated.

The links between predicates and rules of a program are established and stored in a table called **Predicate-Reference Table (PRT)** during initial analysis of the program. Each entry in **PRT** consists of a predicate symbol as the key and a list of pointers, which point to those **R-trees** that call on the predicate in their body and the exact index position of the calling atom in the body of the rule. When a tuple is to be executed, a list of relevant R-trees can be obtained from **PRT** and these R-trees are executed one after another. So no searching of any kind is involved in the execution of tuples.

Notice that tuples with time stamp of interval value are ordered by the lower boundary of time interval in both tuple list and pending list. This is necessary as required by the time-ordered forward deduction algorithm. A tuple or fact becomes true starting at lower boundary and lasts until upper boundary, all the consequences which may result from the tuple could have time stamp, according to the causality principle given in first report, no earlier than the lower boundary. The algorithm requires that all tuples that are derivable from the program rules must be generated before the Current Time advances through the time they start to become true. This requirement could be met only if tuples are ordered by their lower time boundary.

3 R-tree and Its Execution

3.1 R-tree Structure and Properties

The R-tree is used to represent a program rule in the dynamic process of program execution or deduction. A number of clause instances will be derived over the entire process, some are justified, some others are invalidated. To represent original clause and its derivable instances in a structured way rather than simply put them in a plain list, we need a well defined structure to hold them together so that the process of deriving new clause instances with input-tuples could be carried out in more orderly and formal manner. R-tree has been designed with these considerations in mind.

It is appropriate to give the meaning of Starlog execution before presenting the definition of R-tree. Each original program clause can be viewed as consisting of head and body, and the body may contain two types of atoms, **system pre-defined primitives** and **user defined atoms**. System pre-defined primitives have fixed meanings, they are used by user program to specify logical relations or equations among certain terms, usually constants or variables. They are *not executable* in terms of Starlog as there are no corresponding tuples which could be inputs to satisfy these primitive atoms, rather their evaluation is to be handled by special internal procedures of the system. For current implementation, they are passed down to a *Prolog interpreter* developed at this Department, which is capable of doing logical arithmetic and handling interval values. **User defined atoms**, on the other hand, are *executable* in Starlog terms. They can be unified with **input tuples**, the known *facts*, to provide variable bindings, or logical bindings between various terms if the input tuple is a constraint tuple. The result of this input of tuple, if unification succeeds, will be a new clause instance. If the atom being unified is the last user-defined in the body of the clause instance, then a tuple will be generated after successful evaluation of the primitives left in the body. It is not unusual that this evaluation fails because this particular interpretation of the rule is not true, but it may succeed with other input tuples.

A R-tree is the *Starlog representation of a program rule* (or an original clause) and its instances. The root of a R-tree represents a rule, it holds the original clause and possibly some immediate instances of original clause, derived by executing **Indirect primitives** at initialization phase as will be discussed in section 5 later. Each node in the R-Tree represents a class of further derivable clause instances, it holds a changing set of clause instances that have been derived, by the input of certain tuples, from those clauses in its parent node. Figure 1 gives an example of a R-tree for a rule with three user-defined atoms **P, Q** and **R**. A node has as many branches (or son nodes) as the number of user-defined atoms in the clauses of its class except for leaf nodes. Leaf nodes are those nodes that represent the type of clause instances which has only one user-defined atom left *un-executed*. Each branch of a node corresponds to one user-defined atom and is labeled with the name of the predicate which the atom bears. All the clause instances derived from clauses held by a node and with the input of tuples of the same predicate name, say **P**, are stored at (or sunk to) the same son node which is connected by a same-named branch, "**P**". Clause instances derived from leaf nodes are tuples or pending tuples (tuples with negation) and are not kept in R-tree.

From the above description of R-tree definition, we can get the following properties of R-trees.

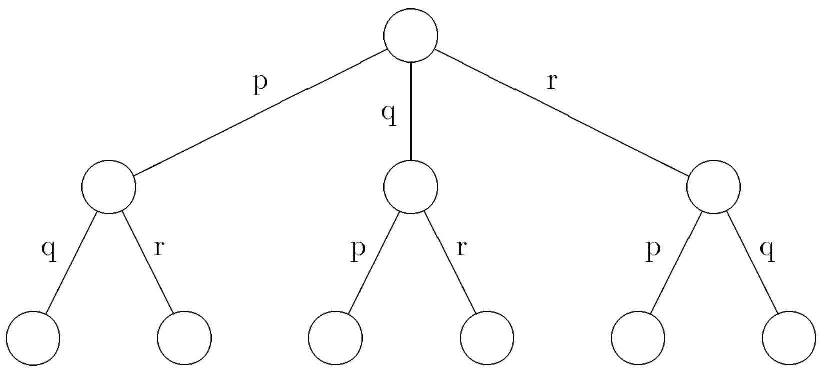

Figure 1: An example R-tree

- The number of branches (or son nodes) a node may have equals to the number of user-defined atoms in the body of clauses it may hold except the leaf nodes. Suppose the number of user-defined atoms in a rule is n, then the root will have n branches, its son nodes will have $n-1$ branches, ..., and the leaf nodes have only one user-defined atom. The R-tree of the rule will have n levels. So the shape or structure of a R-tree solely depends on the number of user-defined atoms in the original clause of a rule and it bears no relation with the number of arguments to its head, whether or not negated, primitives used, etc.

- The clause instances that could be kept at the root or a node all have the same number and same predicate symbols of user-defined atoms. A branch, labeled with say "**p**", leads to a **sub-R-tree** that contains no clause instances with atom name "**p**". So each branch in a R-tree leads to a sub-R-tree reduced by one user-defined atom, which is the sub-R-tree of all the clause instances derived by executing the same atom in clauses kept in its parent node.

- The branches of the root or a node are arranged from left to right in the same order as their corresponding atoms in the clause body.

- Two user-defined atoms of the same predicate symbol are considered as two separate atoms, each has its only branch. They can be distinguished by their positions in the original clause and their respective branches are ordered accordingly. This is true only if the two atoms have different arguments (at least one argument is different), otherwise they are merely the same atom playing different roles in different parts of the rule.

3.2 R-tree Execution

The R-trees not only represent rules and their instance clauses of a program, but also serve as a *control structure in tuple executions.*

According to the execution schedule algorithm described in section 2, A tuple that have earliest time stamp among all existing unused tuples is selected from the Tuple List for execution. This tuple will be used as an input tuple to all relevant R-trees which can be obtained as a list from **PRT** (section 2).

A input tuple arrives at the root of a R-tree and traverses the R-tree making derivation where applicable. Starting at the root, the execution of a R-tree involves two recursively carried out consecutive operations as described below.

- DERIVATION upon arrival of a input tuple at a node, if there are any clauses stored at that node, derivation with the input tuple will be carried out for each of the clauses. Any resulting new clause instances will be stored, if not a leaf node, at the son node which bears the same predicate name as that of the input tuple and is in the same order of position as the atom being executed (or being unified with the input tuple). This son node is called the "**home node**" of the atom and the rest son nodes are said to be "**foreign nodes**" of the atom. Derivation at a leaf node will generate a new tuple, or pending tuple, provided the atom as a goal can be satisfied with the input tuple.

- BRANCHING having done the derivations at a node, if current node being executed is not a leaf node, then move down to each of the foreign nodes in turn from left to right. Each foreign node is regarded as a sub-R-tree and same execution procedure with the same input tuple will be applied until it reaches a leaf node and no branching is possible.

One important feature of this execution mechanism is that **negated clauses** are executed exactly the same way as **non-negated clauses**. Negation evaluation is separated from tuple derivation. The difference is that tuples generated from negated clauses will be put into pending list waiting to be evaluated when the Current Time is about to advance over their time stamp, while tuples generated from non-negated clauses are definite.

The following is the algorithm of R-tree execution with an input tuple:

```
procedure exec-R-trees
begin
  while more R-trees?
    begin
      exec-R-tree;
    end;
end;

procedure exec-R-tree
begin
  while more clauses at the node?
    begin
      derive new clause instances;
    end;
  if not leaf node
    begin
      deposit new clause instances down to home node;
      branching;
    end;
  else new tuples generated
    begin
      /* call primitive evaluator */
      evaluate primitives of new tuples;
      if tuples negated
         put new tuples into pending list;
      else
         put new tuples into tuple list;
    end;
end;

procedure branching
begin
  for each foreign node
    begin
      exec-R-tree;
    end;
end;
```

4 Modified R-trees

The R-tree structure described above works well if the number (N) of user defined atoms in the body of a rule does not exceeds, say, 3 or 4. When a rule contains 5 user-defined atoms, the number of leaf nodes will be 120 and total number of nodes will reach 205, when $N = 10$ there will be 3628800 leaf nodes. This is totally out of question. The number of nodes at each level of the R-tree equals to permutation A_n^i, i is the number of level (starting with root of level 0). The total descendant nodes of a R-tree equals to:

$$A_n^1 + A_n^2 + ... + A_n^{n-1}$$

It is obvious that there are plenty of redundancy. It is not necessary to have atom groups of same combination but different permutations to be assigned with different nodes. In Figure 1 clauses derivable by first *input* **p** then *input* **q** and clauses derivable by first *input* **q** then **p** are kept in different nodes. This is not necessary because the order of unification of atoms does not matter. Each atom group of same combination needs only one node (and its sub-Rtree) and the node is assigned to the left-most among branches of the same atoms combination. The number of node in a R-tree can thus be reduced significantly by merge redundant nodes. The number of nodes at each level will be the combination C_n^i instead of permutation and the total number of descendant nodes becomes:

$$C_n^1 + C_n^2 + ... + C_n^{n-1}$$

Figure 2 shows the modified R-tree structure. For a R-tree with 10 user-defined atoms there are only 1022 compared to more than three million. However, this is still a large number of nodes, what's more is that eliminating redundancy in R-tree does not reduce the excessive time spent traversing R-tree. The system-overhead in executing such R-trees is not tolerable. A mechanism must be found to reduce the execution time for large R-trees as well. next section will discuss this topic.

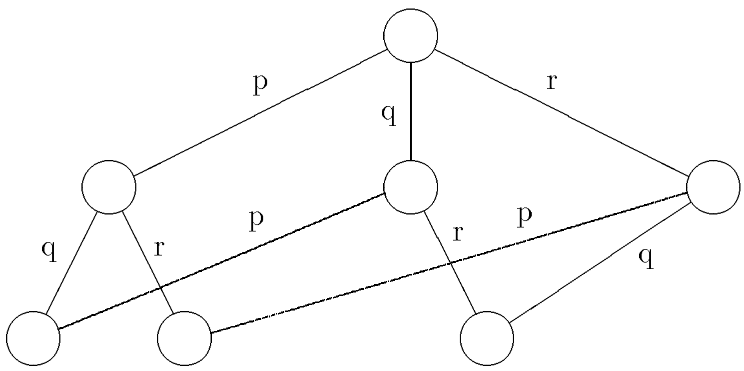

Figure 2: Modified R-tree of Figure 1

5 Identify Indirect Primitives

Eliminating redundancy in Rtree does not at all reduce the excessive time spent traversing R-tree. In this section we will examine another notion which can help bring the size of R-trees under control. It is quite common that a rule may contain some **user-defined atoms** which are themselves defined solely by primitives. These atoms may be considered as *Indirect Primitives* (abbreviated as **IP**s). Those rules that define IPs will never generate tuples as far as Starlog execution is concerned, in fact these rules are constrained tuples. It doesn't make much sense to treat IP atoms as any other user-defined atoms because one more user-define atom in the body of a rule simply means one more level and much more nodes in the R-tree for that rule. These IPs are static in the sense that for each IP there are fixed limited number of definitions, i.e constrained tuples. They can be used, at the initialization phase rather than in execution phase of a program, if the number of their definitions are not very large, as input tuple to derive limited number of clause instances for those rules that "*use*" them. This will reduce the number of user-defined atoms for relevant rules and will prevent the R-trees from growing too big. Further more, those rules that contains only *System-defined Primitives* and/or *Indirect Primitives* may also be treated as IPs and the size of R-trees can be brought down even further.

¿From the above discussion, **Indirect Primitives** can be defined inductively as:

- *rules that contain only primitive atoms in their body;*
- *rules that are defined solely by system primitives and/or indirect primitives.*

Identify IPs and do not count them as user-defined atoms (that is no branches will be assigned to these atoms) and yet treat them in a similar way as other user-defined atoms in terms of derivation by input tuple. This could reduce the system-overhead and still maintain the simple uniform execution mechanism. One could expect that most logic programs are of recursive nature, the number of "*real*" user-defined atoms in the body fo a rule would usually rather small so that R-tree mechanism is very efficient and extremely simple.

Indirect Primitives cab be identified at initialization phase of Starlog program execution by two steps:

- *find those **predicates** that are solely defined by system primitives. These are the original IPs.*
- *recursively scan R-trees to find other rules that are defined by system primitives and/or indirect primitives only. These are the secondary IPs.*

IPs are executed at initialization phase and the resulting rule instances are to replace the original rule kept at the root of its R-tree.

6 Conclusion

The R-tree structure allows for efficient program representation and execution provided that the number of "*true*" *user-defined atoms* in a rule is small. "*Large*" rules can always be broken down to smaller rules to bring the size of rules under control.

7 Acknowledgments

This work was completed with support from the National Science and Engineering Research Council of Canada. We would also like to thank Alan Dewar for his help.

References

[1] Cleary, J.G. "*Logical Arithmetic*" 1987.
Future Computing Systems 2, no.2, pp. 125-149.

[2] Cleary, J.G. "*Colliding Pucks Solved in a Temporal Logic*" 1990.
Proceedings Distributed Simulation Conference. San Diego, California, January.

[3] Klocksin, W.F. and Mellish, C.S. *Programming in Prolog.* 1981.
Springer-Verlag.

[4] Dybvig, R.K. *The SCHEME Programming Language.* 1987.
Prentice-Hall.

[5] Kaushik, V.N. *Starlog: from Semantics to Interpretation.* 1991.
MSc. Thesis, University of Calgary.

[6] Lloyd, J.W. *Foundations of Logic Programming.* 1987.
Springer-Verlag.

[7] Naqvi, S. and Krishnamurthy, R.
"*Database Updates in Logic Programming*" 1988.
ACM Symposium on PODS, pp. 261-262.

[8] Warren, D.S. "*Memoing for Logic Programs*" 1992.
Communications of the ACM, March 1992/Vol.35, No.3, pp. 94-111.