JPL : IMPLEMENTATION OF A PROLOG SYSTEM SUPPORTING INCREMENTAL TABULATION

Taher Ali[1], Ziad Najem[2], and Mohd Sapiyan[1]

[1]Department of Computer Science, Gulf University for Science and Technology, Kuwait
ali.t@gust.edu.kw, sapiyan.m@gust.edu.kw
[2]Department of Computer Science, Kuwait University, Kuwait
najem@cs.ku.edu.kw

ABSTRACT

The incremental evaluation of tabled Prolog programs allows to maintain the correctness and completeness of the tabled answers under the dynamic state. This paper presents JPL implementation details. JPL is an approach to support incremental tabulation for logic programs under non-monotonic logic. The main idea is to cache the proof generated by the deductive inference engine rather than the end results. In order to be able to efficiently maintain the proof to be updated, the proof structure is converted into a justification-based truth-maintenance (JTMS) network.

KEYWORDS

Applications of justification-based truth maintenance systems, Belief revision systems, Truth maintenance systems, Justification-based truth maintenance systems, Incremental evaluation of tabled Prolog, Incremental tabulation for Prolog queries, Tabulation for logic programs, Memoing for logic programs.

1. INTRODUCTION

Prolog is a logic programming language associated with artificial intelligence and computational linguistics [1, 2, 3]. Tabled extension for Logic Programming (TLP) [4, 5, 6] evaluation enhances the performance of the Prolog query engine and allows the termination of certain computations that do not terminate under the normal Prolog query evaluation. The incremental evaluation [7, 8] of tabled Prolog programs allows to maintain the correctness (soundness) and completeness of the tabled answers under the dynamic state. This evaluation strategy allows the system to update the tabled answers when the set of facts and/or rules participated in generating the answers of a certain query are either retracted from or asserted to the Prolog program. JPL [7, 9] presented an approach of maintaining one consolidate system to cache the query answers under the non-monotonic logic. It uses the justification-based truth-maintenance system to support the incremental evaluation of tabled Prolog Programs. In this paper we will focus on the system

Jan Zizka et al. (Eds) : CCSIT, SIPP, AISC, CMCA, SEAS, CSITEC, DaKM, PDCTA, NeCoM - 2016
pp. 323–338, 2016. © CS & IT-CSCP 2016 DOI : 10.5121/csit.2016.60127

implementation. The system implementation must take into consideration the following performance factors:

1. Minimizing the overhead of caching the query proof structure.

2. Minimizing the time and space needed to maintain soundness and completeness of the cached proof structure.

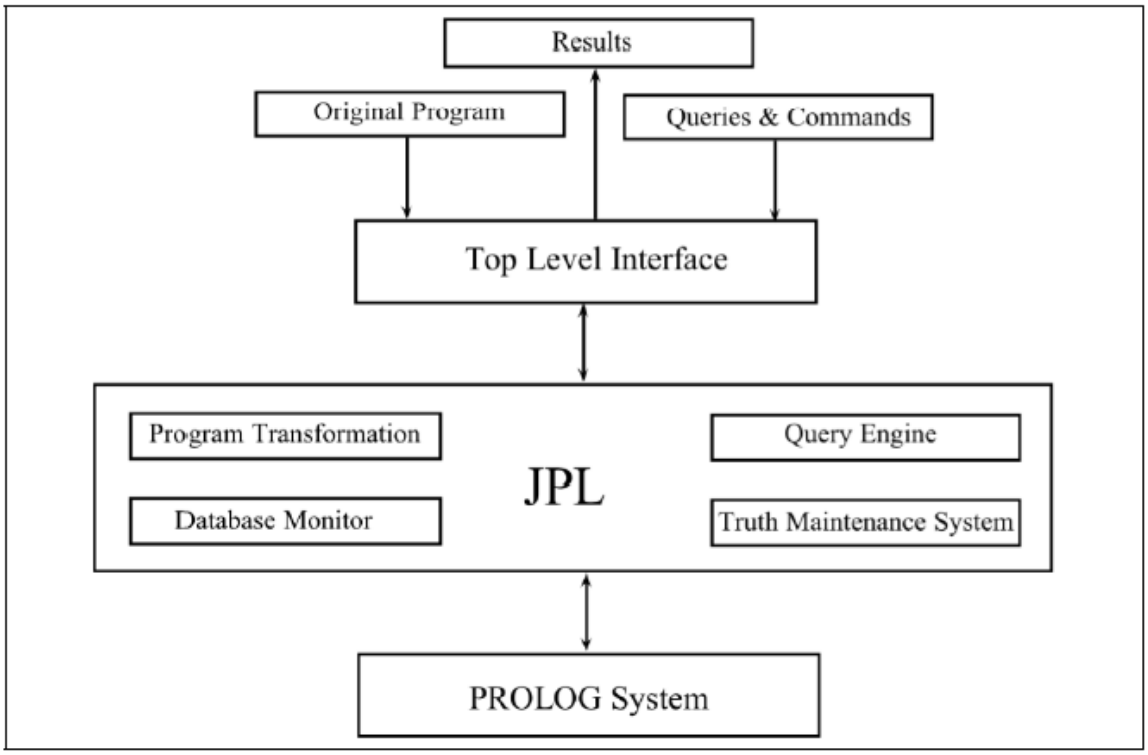

Figure 1: An overview of JPL

Another factor that we have to take into consideration regarding the system implementation is the portability of integrating JPL with more than one PROLOG inference engine.

2. RELATED WORK

Tabling is an implementation technique where answers for subcomputations are stored and then reused when a repeated computation appears. There are two approaches to integrate tabling support into existing PROLOG systems. The first approach is to modify and extend the low-level engine. This is the common approach used by most of systems to support tabled evaluation. The approach modifies and extends the low-level engine [10]. The advantage is the run-time efficiency, however, the drawback is that it is not efficiently portable [11] to other Prolog systems because the engine level modifications are slightly more complex and time consuming. The second approach to incorporate tabled evaluation into existing Prolog systems is to apply the source level transformations to a tabled program, and then use external tabling primitives to provide direct control over the search strategy. This idea was first explored by Fan and Dietrich

[12] and later used by Rocha, Silva and Lopes [13] to implement tabled PROLOG systems. The main advantage of this approach is the portability to apply the approach to different PROLOG systems. The drawback of course is the efficiency, since the implementation is not at a low level.

Algorithm 1 Program Transformation Algorithm

Input: R, original Program rules list
Output: R_t, transferred rules List

```
assign each rule in R a unique id
for each rule Ri in R
        h = head of Ri, b = body of  Ri;
        ht= h with extra unique var argument Jc;
        bt = b+",";
        bt = bt + "append(["+id(Ri)+"],[";
        bt = bt + "["+all non-negated terms in b+"],";
        bt = bt + "["+all negated terms in b+"],["+h+"]";
        bt = bt + "],"+ Jc + "  )";
        Rti = ht + ":-" + bt+".";
```

3. JPL IMPLEMENTATION APPROACH

JPL implementation is based on applying the source level transformations to a tabled program. This will allow to incorporate the idea of incremental tabulation into different PROLOG systems using the same general framework. In order to integrate JPL with PROLOG inference engine we are using INTERPROLOG. INTERPROLOG [14] is an PROLOG-JAVA application programming interface that supports multiple PROLOG systems through the same API. The current version (2.1.2a at the time of implementing this approach) of INTERPROLOG supports three PROLOG implementations (XSB [15], SWI-PROLOG [16] and YAP-PROLOG [17]) on different platforms (Windows, Linux and Mac OS X). It promotes coarse-grained integration between logic and object-oriented layers, by providing the ability to bidirectionally map any class data structure to a PROLOG term. This basic idea of INTERPROLOG is allowing PROLOG to call any JAVA method, and for JAVA to invoke PROLOG goals. This is achieved by using a communication layer to pass object/term data among both processes. INTERPROLOG is used in the implementation of JPL to provide an interface from JPL to all PROLOG inference engines supported by INTERPROLOG.

4. PROGRAM TRANSFORMATION

PROLOG rules are written in a standard form known as Horn clauses. A Horn clause statement has the form:

$$H : -B_1, B_2,...,B_n.$$

where H is a first-order atom and each B_i is a first-order literal. H is called the head of the clause and $B_1, B_2,...,B_n$ is the body of the clause. The semantic of this statement is that when B_i all are true, we can deduce that H is true as well. The above clause can be read "H, if $B_1, B_2,\ldots,Bn$". This basic concept of logic programming is used by JPL to do the program transformation. When a PROLOG programmer executes the command consult/1 to read a PROLOG source file through the JPL's top level interface, JPL executes the program transformation method to convert the

PROLOG predicates (rules) into a format that allows the system later to table the query answers as a JTMS network according to the mechanisms described in the previous chapter.

JPL applies program transformation only on clauses that are selected by the PROLOG programmer to be as incremental tabled predicates. Algorithm 1 illustrates the program transformation process. The process starts with assigning each rule in the original program a unique ID.

```
%Original Tabled Predicates
connected(X,Y) :- edge(X,Y). %r1
connected(X,Y) :- edge(X,Z), connected(Z,Y). %r2

%Transformed Predicates by JPL
connected(X,Y,J0) :- edge(X,Y), append([r1],[[edge(X,Y)],[],connected(X,Y)],J0).
connected(X,Y,J0) :
-edge(X,Z), connected(Z,Y,J1), append([r2],[[edge(X,Z),connected(Z,Y)],[],connected(X,Y)],J0).
```

Figure 2: Original tabled and transformed Predicates by JPL for the translative closure program of the directed edge relationship.

For example, The first rule in the translative closure program of Figure 2 is uniquely identified as r1, while the other rule is defined as r2. The next step in the algorithm is to convert each rule in the original program into a format that allows the query engine later to link the facts which are participated as antecedents to produce the consequence (answer) of the query. This is achieved by applying two major changes in each program rule:

1. Add an extra unique var argument to the rule head, this var points to a list which contains all the necessary information needed by JPL to build the JTMS network of any query answer generated from this rule. The list is to be composed of the following items:

   (a) The rule ID.
   (b) List of facts (antecedents), coming from the rule body, that must be true (IN) to support the rule consequence.
   (c) List of facts (antecedents), that must be false (OUT) to support the rule consequence. This case is used when the body contains negated terms.
   (d) The consequence of the rule, which is basically the head of the rule.

2. Add an extra term to the rule body that generates the above list and stores it in the extra var added to rule head.

Once the rule is in its final format using the above conversion method, a TMS node is created for the rule and it is linked to the TMS node of the original rule head. This will help the query engine later to identify all rules related to a certain query entered by the user. Figure 2 shows how a *connected/2* tabled predicate that defines the translative closure program of the directed edge relationship is transformed to support JPL's tabulation strategy. Each rule head, in Figure 2, is converted from *connected(X,Y)* to *connected(X,Y,J0)* by adding an extra var argument J0. In the same manner each rule body is changed by appending the required code that originates the list in the var J0. For example the second rule body is changed from :

edge(X,Z), connected(Z,Y)

to:

edge(X,Z); connected(Z,Y,J1); append([r2]; [[edge(X,Z), connected(Z,Y)]; []; connected(X,Y)]; J0):

Both of these two rules are linked to the TMS node of the original rules head, i.e. *connected(X,Y).*

5. QUERY ENGINE

Algorithm 2 Query Engine

Input: *Q*, a PROLOG query.
Output: List of query answers.

```
queryTmsNode = tmsNode assocaited with Q
if ( queryTmsNode != full ){
        tn = getGrandParent(Query);
        gt = getRules(tn);
        while (gt != null ){
                justifications=executePrologQuery(gt.element.rhs);
                installJustifications(justifications);
        }
        justifications=executePrologQuery(Query);
        installJustifications(justifications);
    }
}
showQuerySolutions(queryTmsNode);
```

Query engine of JPL responds to end user queries through the system top interface. The necessary methods needed to implement the logic of the query engine are part of the JTMS class. There are two scenarios for the query execution module that are described in Algorithm 2. The first scenario checks if the TMS node associated with the user query already exists in the JTMS network and is marked as fully proved. When this condition is true the query engine simply shows all the valid(IN) answers of the query from the JTMS network linked to the query's TMS node. The second scenario deals with the case when the query is executed for the first time. JPL deals with this case as a three step process:

1. The system locates the general TMS node associated with the query. The desired TMS node is used to locate the set of PROLOG rules associated with the query. If the set is not empty, the query engine requests the PROLOG abstract engine attached to it to execute the right hand side of each rule. The answers (justifications) coming from the PROLOG abstract engine execution are installed as justifications in the JTMS network.

2. The query engine tries to find more solutions for the query that might come from the database of facts which were not discovered in the first step. If such answers exist, then justifications related to these answers are also installed in the JTMS network.

3. After the previous two steps, the query answers are ready. The query engine simply shows all the answers of the query from the JTMS network linked to the query's TMS node.

5.1 Installing the Justifications

As seen in Section 5, the PROLOG abstract engine, attached to JPL, executes PROLOG queries requested by the query engine. The results of these queries must be installed as justifications in the JTMS network associated with the query. Figure 3 represents the list of answers (b) returned by the PROLOG abstract engine for the query connected(X,Y) with respect to the translative closure PROLOG program (a). The list of results are in a format that allows the system to convert them easily into justifications. The list item contains the rule participated as antecedent in generating the deduction. The set of facts participated as antecedents in generating the deduction.

$connected(X,Y) :- edge(X,Y).$ %r1
$connected(X,Y) :- edge(X,Z), connected(Z,Y).$ %r2
$edge(a,b).\ edge(a,c).\ edge(b,d).\ edge(c,d).\ edge(d,e).$ %f1..f5

(a)

$[r2, [edge(a,b), connected(b,d)], [], connected(a,d)]$
$[r2, [edge(a,c), connected(c,d)], [], connected(a,d)]$
$[r2, [edge(a,b), connected(b,e)], [], connected(a,e)]$
$[r2, [edge(a,c), connected(c,e)], [], connected(a,e)]$
$[r1, [edge(a,b)], [], connected(a,b)]$
$[r1, [edge(a,c)], [], connected(a,c)]$
$[r2, [edge(b,d), connected(d,e)], [], connected(b,e)]$
$[r1, [edge(b,d)], [], connected(b,d)]$
$[r2, [edge(c,d), connected(d,e)], [], connected(c,e)]$
$[r1, [edge(c,d)], [], connected(c,d)], [r1, [edge(d,e)], [], connected(d,e)]$

(b)

Figure 3 : List of answers(b) returned by the PROLOG abstract engine for the query connected(X,Y) with respect to the translative closure PROLOG program(a).

The answer itself which represents the consequence of the deduction. Below are the details of each list item:

1. The rule ID that is behind the generation of this result.

2. Set of PROLOG items that must be IN to support the consequence of the result. This case is related to non-negated subgoals in the right hand side of the rule.

3. Set of PROLOG items that must be OUT to support the consequence of the result. This case is related to negated subgoals in the right hand side of the rule.

4. The consequence of this result.

These four items are one to one mappings to the data attributes for the Justification class. The class contains five attributes. The first attribute is used as a flag to indicate whether the

justification is active or not. The rest of the attributes are used to store the above information for the justification.

Going back to the example of Figure 3(b), the first answer:

[r2; [edge(a,b); connected(b,d)]; []; connected(a,d)]

for the query *connected(X,Y)* states the following:

1. The rule *r2* is in the antecedent list of the answer *connected(a,d).*
2. The atoms *edge(a,b)* and *connected(b,d)* must be IN to support the consequence of the the answer *connected(a,d).*
3. None of the atoms must be OUT to support the consequence of the answer since there are no negated subgoals in the program of Figure 3(a).
4. The consequence of this answer is the Prolog atom *connected(a,d*).

For each of the answers of Figure 3(b), JPL installs a justification for it and makes the justification active. When the data related to any justification is changed, the system maintains the consistency of the justification according to the changes in the set of rules/-facts related to the justification. An important point that must be highlighted is that each justification element is stored as a TMS node. The details of the TMS node data structure is given in next section.

6. TMS NODES

A TMS node is the basic data structure used in JPL to cache the proof structure of a PROLOG query. The set of TMS nodes linked together through justifications represents the JTMS network for the query. The TMS node is a complicated data structure, it must store all the information that allows the system to maintain the consistency and completeness of the cached proof structure for a previously proven query. The following subsections describe the main most important attributes and operations for this crucial data structure.

6.1 Attributes

Label

The label attribute stores the current status of the node. At any time, the TMS node is labeled one of the following:

1. IN or OUT

One of these two labels are used to represent the status of the TMS node when the TMS node that is pointing to a PROLOG predicate, i.e. a fact or a rule. When the system believes that the PROLOG predicate is true or active, then the label of the TMS node is IN. The TMS node label is OUT when the predicate is false or inactive.

2. *Full, Partial or New*

One of these three labels are used when the TMS node is pointing to a PROLOG query. The label Full indicates that the query associated with the TMS node is fully proved and when the user re-evaluates the query no additional work would be required from the PROLOG inference engine attached to JPL in generating the answers of the query. The label Partial means that the query attached to the TMS node is partially proved. If a partially proven query is going to be evaluated then the system must complete the query evaluation before returning its answers. The New label is used when the query is proved for the first time and a new TMS node is created for the query.

Type

This attribute stores the type of the TMS node. JPL uses three types of TMS nodes. The type of the TMS node is either a PROLOG fact, rule or a query.

Node

The Node attribute points to the actual PROLOG term.

Support

This object represents the set of justifications that support the PROLOG fact attached to this TMS node.

Algorithm 3 Set label operation's algorithm for the TMS node

Input: *newLabel*, the new label
Output: *void*

```
if (this.label != newLabel) {
        boolean active = false;
        if ( this.type == Fact && newLabel == OUT ){
                active = this.findAnotherSupport();
                if (!active)      {
                        this.label = x;
                        if (this.type == 'Fact' || this.type == 'Rule')
                                this.propagateInnessOutness(myJtms);
                }
        }
}
```

In-List

The set of justifications where the PROLOG predicate, associated with this TMS node, is participating as an IN(*true*) antecedent in the justification.

Out-List

The set of justifications where the PROLOG atom, associated with the this TMS node, is participating as an OUT(*false*) antecedent in the justification.

Related Queries

This set points to the list of other cached queries, in the system, which are effected whenever there is a change in the proof structure of the PROLOG query attached to this TMS node.

Rules

If the TMS node is associated with a PROLOG query where the query term matches the head of certain rules in the PROLOG program, then this attribute points to the query related rules from the program. Note that some of the above attributes might have a Null value depending on the type of the TMS node. For example, if the TMS node is associated with a PROLOG query, then the values of support, in-list and out-list is Null since they are used with TMS nodes that point to a PROLOG fact.

6.2 Operations

Set Label

The "Set Label" is the most critical operation of JPL. The operation is maintaining the consistency and the completeness for the queries cached proof structure. Algorithm 3 describes the set label process for a TMS node. The algorithm starts with checking if the new label is different from the current label of the TMS node. This check is required to avoid any redundant computations when there is no change in the label. The next step in the algorithm is to check if the label of the TMS node is being changed from IN to OUT and if that particular node is pointing to a PROLOG fact. If this is true, then the system tries to find an active justification that can support this fact, and hence, avoid changing its label from IN to OUT. If no such active justification is found, or the TMS node type is not a fact, or the new label is being changed from OUT to IN, then the system changes the label of the TMS node to its new value. Once the label value is changed, JPL propagates the effect of this change throughout the JTMS network by calling the propagateInnessOutness method.

Algorithm 4 Propagating the change of a TMS node label throughout the JTMS network.

Input: ***this***, the node with changed label.
Output: ***void***

```
for each justification in this.inList {
        if (this.label == IN)
                try to make the justification active;
        else
                make the justfication inactive;
}
for each justification in this.outList {
        if (this.label == OUT)
                try to make the justification active;
        else
                make the justfication inactive;
}
```

Propagate Inness/Outness of Existing Facts/Rules

The objective of this operation is to maintain the consistency of the JTMS network whenever there is a change in the label of a TMS node. The algorithm for propagating the change of a TMS node label throughout the JTMS network is explicated in Algorithm 4. The logic of the algorithm is straightforward. For each justification in the in-list of the current TMS node, the system tries to set the justification active when the label of TMS node is IN; otherwise the justification is set as inactive. Also, for each justification in the out-list of the current TMS node, the system tries to set the justification active when the label of the TMS node is OUT, otherwise the justification is set as inactive.

- Activating a justification

  To activate a justification, the system must ensure that all antecedents of the In-List are labeled as IN, and all antecedents of the Out-List are labeled as OUT. If these two conditions are satisfied, then the justification's consequent TMS node label is changed to IN, and the justification is marked as active.

- Inactivating a justification

  To mark a justification as inactive, the system changes the justification's consequent TMS node label to OUT. Then the justification is marked as inactive.

Propagate the Asserting of a New Fact

This method is called when a new PROLOG fact is asserted through the JPL interface. Figure 5 outlines the algorithm for propagating the effect of inserting a new PROLOG fact. The system locates the TMS query nodes that might get affected from the insertion of the new fact. For each such query, if the query has been fully proved previously, the algorithm locates the set of rules attached to the query's TMS node. Every rule is unified with the new fact, then the system executes the partial PROLOG query (which is coming from the right hand side of the rule) to update the cached proof structure.

Algorithm 5 Propagating the effect of asserting a new PROLOG fact.

Input: *this*, the new PROLOG fact's TMS node.
Output: *void*

```
RQ= this.relatedQueries;
for (each Query in RQ) {
        if (Query.label == Full) {
                Rules= Query.getRules();
                for (each Rule in Rules) {
                        Query.executePartialQuery(unify(Rule, this));

                }
        }
}
```

Algorithm 6 Propagating the effect of asserting a new PROLOG rule.

Input: *this*, the new PROLOG rule's TMS node.
Output: *void*

```
ruleQueryNode=tms.find(this.head);
if (ruleQuery != null) {
        if (ruleQuery.getLabel() == Full) {
                Jtms.executePrologRule(head, this);
        }
}
```

Propagate the Asserting of a New Rule

This method is called when a new PROLOG rule is asserted through the JPL interface. Figure 6 outlines the algorithm for propagating the effect of inserting a new PROLOG rule. The process starts with locating the TMS query node that matches the new rule head. If such query node exists and it is marked as fully proved, then the system requests the PROLOG inference engine attached to JPL to execute the right hand side of the new rule to update the cached proof structure. This is achieved by invoking the JTMS method *executePrologRule.*

7. The JTMS CLASS

The JTMS class is the main component of JPL. It interacts with the system's top interface (JPL class) to provide the desired functionality (see Figure 2). The main data component and operations of this class are described briefly below.

7.1 Attributes

PROLOG Engine

The PROLOG engine points to the PROLOG inference engine. The integration between the PROLOG inference engine and JPL is done by using InterProlog package. See Section 3 for more details about InterProlog.

TMS Object

The object points to an instance of the TMS class. See Section 8 for more details about this component.

Algorithm 7 Executing a PROLOG Query

Input: *query*, the PROLOG query(goal).
Output: *void*,

```
solutionVars=engine.deterministicGoal(query);
for (each solution in solutionVars ){
        this.installJustifications(solution);
}
```

Algorithm 8 Handling the the assertion of a fact/rule to the PROLOG program.

Input: term, the asserted PROLOG fact or rule.
Output: *void*,

```
engine.deterministicGoal(assert(term));
TmsNode t1 = myTms.findNode(term);
if (t != Null ) {
        t1.setLabel(IN)
} else {
        if (term == Fact ) {
                Tms newNode = myTms.addNode(term);
                newNode.propagateNewAtom();
        } else {
                Term newRule = convert(term);
                TmsNode newNode = myTms.addNode(newRule);
                newNode.propagateNewRule();
        }
}
```

Justifications Table

The justification object of JTMS class points to the list of all current justifications in the system. Justifications are maintained as a hash table to allow fast retrieval of information related to them. See Section 5.1 for more details about how the justifications are installed and maintained by JPL.

7.2 Operations

Executing PROLOG Queries

This method allows the JTMS class to execute the Prolog queries with the help of the Prolog inference engine object. Usually this method is called by the query engine when JPL's end user requests to execute the query for the first time. Later, the method is invoked by the TMS node objects to maintain the correctness and completeness of the cached proof structure for the same query. Algorithm 7 outlines the execution of a Prolog query. This is a two step process. First the incoming query is executed by invoking the method *deterministicGoal* which returns the set of solutions for this particular query. In the next step, the algorithm takes each solution returned by the Prolog inference engine and calls the method *installJustifications* to install the justification in the JTMS network of JPL.

Algorithm 9 Handling the the retraction of a fact/rule from the PROLOG program.

Input: term, the retracted PROLOG fact or rule.
Output: *void*,

```
engine.deterministicGoal(retract(term));
if (t != Null ) {
        TmsNode t1 = myTms.findNode(term);
        t1.setLabel(OUT)
}
```

jAssert

The method *jAssert* is invoked when the end user initiates the PROLOG command assert/1, through JPL's user interface to insert a fact/rule into a PROLOG program. Algorithm 8 describes the steps to handle the assertion of a fact/rule to the PROLOG program. The first step in the algorithm is to inform the PROLOG engine about this change in the set of facts/rules related to the associated program. This is achieved by executing the assert command by the PROLOG engine. In the next step, the algorithm tries to locate the TMS node associated with the asserted fact/rule. If the TMS node already exists in the database of TMS nodes then the method setLabel (See Section 6.2) is invoked to propagate the effect of this assertion through out the JTMS network.

If the TMS node, associated with the asserted fact/rule, does not exist in the database of TMS nodes, then this case is related to the insertion of a new PROLOG fact/rule to the PROLOG program which was not presented at the program consult time. The system handles the insertion of new PROLOG predicates according to the following scenarios:

1. Asserting a new fact

   A new TMS node is created for the new fact. The algorithm 8 then invokes the method *propagateNewAtom* (See Section 6.2) to update the JTMS network in response to this change in data.

2. Asserting a new rule

   A new TMS node is created for the new rule. The rule is converted according to the format described in Section 4. Then the algorithm invokes the method *propagateNewRule* (See Section 6.2) to update the JTMS network in response to this change in the set of rules.

jRetract

The inverse logic of *jAssert*. This method is invoked when the end-user initiates the Prolog command retract=1, through JPL's user interface, to remove a fact/rule from the Prolog program. Algorithm 9 shows the three required steps to handle the retraction of a fact/rule from the Prolog program. In the first step, the system informs the Prolog engine about this change in the database of facts/rules by executing the retract command on the Prolog engine. Then the algorithm tries to locate the TMS node associated with the asserted fact/rule. If the TMS node already exist in the database of TMS nodes, then the algorithm invokes the setLabel method to propagate the effect of this retraction throughout the JTMS network. If the TMS node associated with the retracted fact/rule does not exist in the database of TMS nodes, then no action is required because the retracted fact/rule is not part of the current JTMS network.

8. TMS CLASS

The TMS class is used in JPL to link all TMS nodes together in a parent/child relationship. Basically, the TMS class stores and links the TMS Nodes using the Graph data structure.

Algorithm 10 Adding a PROLOG term to the TMS Network.

Input: term, the PROLOG term; type, the type of the PROLOG term; label, the label of the PROLOG term.
Output: t, the TMS node associated with the PROLOG term.

```
t1 = new TmsNode(node, label, type);
this.addVertex(t1, null);
TmsNode t2=this.addNode( t1.getGrandParent(),'I',New);
this.addEdge(t2, t1);
switch (nodeType) {
        case Fact:
                t1.propagateNewAtom();
        break;
        case Rule:
                t1.propagateNewRule();
        break;
}
return t1;
```

In the rest of this section, a brief description is given about the main components of the TMS class.

Attributes

There are two main attributes in this class. The first attribute is the *hashTable* that keeps the track of Prolog terms (facts, rules, and queries) with their associated TMS nodes in the *HashTable* data structure to allow fast retrieval of the terms whenever required by other system components. The other attribute of the TMS class is the *adjhash* which represents the adjacency list of each TMS node. The adjacency list of each TMS node links it to other TMS nodes in the JTMS network of JPL.

Operations

The operations of the TMS node are the typical ones of any Graph or *HashTable* abstract data type. Mainly the most important operations are:

Add a TMS node

The algorithm for adding a PROLOG term to the TMS Network is presented in 10. The algorithm starts with creating a new TMS node for the Prolog term. The newly created TMS node is added to the Graph as a vertex. Then the algorithm links the TMS node to its most general term. Once this setup is done, the algorithm propagates the effect of this new Prolog term throughout the JTMS network.

Find a TMS Node

Given a Prolog term (fact, rule, or query), the find operation returns a pointer to the TMS node object associated with the Prolog Term. If such TMS node does not exist in the *HashTable*, then the method returns null.

9. CONCLUSION

This paper presented JPL implementation details. The main aim of this research is to develop a tabled Prolog system that is capable of caching and maintaining the correct answers of the query under the non-monotonic logic in an efficient way. The design try to avoid, as mush as possible, the limitations and disadvantages in the existing approaches to support incremental evaluation of tabled PROLOG programs. To judge that our design meets it's objectives, a comprehensive performance analysis of JPL is required. This will be the target of future work.

REFERENCES

[1] W F. Clocksin and Christopher S. Mellish. Programming in Prolog (2nd ed.). Springer-Verlag New York, Inc., New York, NY, USA, 1984.

[2] Michael A. Covington. Natural Language Processing for Prolog Programmers. Prentice-Hall, Englewood Cliffs, NJ, 1994.

[3] Ivan Bratko. Prolog Programming for Articial Intelligence. Pearson Addison-Wesley, Harlow, England, 3 edition, 2000.

[4] Hisao Tamaki and Taisuke Sato. Old resolution with tabulation. In Proceedings on Third international conference on logic programming, pages 84 98, New York, NY, USA, 1986.Springer-Verlag New York, Inc.

[5] David Scott Warren. Memoing for logic programs. Commun. ACM, 35(3):93 111, 1992.

[6] Terrance Swift. Principles, practice, and applications of tabled logic programming. SIGSOFT Softw. Eng. Notes, 25(1):87 88, January 2000.

[7] Taher Ali, Ziad Najem, and Mohd Sapiyan. Query proof structure caching for incremental evaluation of tabled prolog programs. International Journal of Computer Science and Information Technology, 6(1):311, Feb. 2014.

[8] Diptikalyan Saha and C. R. Ramakrishnan. Incremental evaluation of tabled logic programs. In ICLP, pages 392 406, 2003.

[9] Taher Ali, Ziad Najem, and Mohd Sapiyan. A belief revision system for logic programs. Computer Science and Information Technology, 4(9):227 231, Sep. 2014.

[10] Konstantinos Sagonas, Terrance Swift, and David S. Warren. Xsb: An overview of its use and implementation. SUNY Stony Brook, pages 11794 4400, 1993.

[11] Jan Wielemaker and Vítor Santos Costa. Portability of prolog programs: theory and casestudies. CoRR, abs/1009.3796, 2010.

[12] Changguan Fan and Suzanne Wagner Dietrich. Extension table built-ins for prolog. Softw. Pract. Exper., 22(7):573 597, July 1992.

[13] R. Rocha, C. Silva, and R. Lopes. Implementation of Suspension-Based Tabling in Prolog using External Primitives. In J. Neves, M. Santos, and J. Machado, editors, Local Proceedings of the 13th Portuguese Conference on Articial Intelligence, EPIA'2007, pages 11 22, Guimarães, Portugal, December 2007.

[14] Miguel Calejo. Interprolog: Towards a declarative embedding of logic programming in java. In José Júlio Alferes and João Alexandre Leite, editors, JELIA, volume 3229 of Lecture Notes in Computer Science, pages 714 717. Springer, 2004.

[15] Terrance Swift and David Scott Warren. Xsb: Extending prolog with tabled logic programming. TPLP, 12(1-2):157 187, 2012.

[16] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. Theory and Practice of Logic Programming, 12(1-2):67 96, 2012.

[17] Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The yap prolog system. TPLP, 12(1-2):5 34, 2012.

AUTHORS

Taher M. Ali received his B.Sc. and M.Sc. from Kuwait University and PhD from University of Malaysia. Dr. Tahir has over 15 years of experience in higher education sector. The experience is divided between academia as faculty member and IT related jobs. He is currently working as an Assistant Professor of Computer Science in Gulf University for Science and Technology (ABET and ACCSB accredited), and also serving as a head of computer science department since Fall 2015. He has also served the university Chief Information Officer (CTO) between 2009-2015. During his time as CTO, he has stabilized the IT department infrastructure, services, policies and procedures. Under Dr. Taher leadership, the IT department at GUST university is fully maintained by in-house team with high utilization of open source technologies and effective integration of different systems such as: student information, learning management, attendance, security, access, web, library, work flow, faculty information, and other sub systems required to manage the daily business activities of students/faculty/staff. In 2013, Dr. Taher has received The Kuwait Electronic Award for Enriching e-Content under category of Under Category of Electronic Sciences.

Ziad H. Najem received his BSc from Kuwait University and Ms and PhD from University of Illinois at Urbana-Champaign. Prior to joining the Department of Computer Science at Kuwait University in 1999, Dr.Najem worked as a Scientific Researcher at Kuwait Institute for Scientific Research.

Mohd Sapiyan Baba is currently a Professor of Computer Science in Gulf University for Science and Technology, Kuwait. He was a lecturer in University of Malaya for more than 30 years, teaching Mathematics and Computer Science courses, and supervised numerous students for their research projects at undergraduate and postgraduate levels. His main research interest is in field of Artificial Intelligence (AI), in particular, the application of AI in Education