Initial Results on Justification for the Tabled Transaction Logic

Paul Fodor
Department of Computer Science
Stony Brook University
Stony Brook, NY 11794-4400, U.S.A
pfodor@cs.sunysb.edu

Abstract

In this paper, we describe our preliminary justification techniques for the transactional queries executed in the tabled Horn Transaction Logic. Transaction Logic is an extension of classical logic programming with state updates similar to database transactions suitable for tasks ranging from processing complex event workflows to AI planning. Execution with tabling is a technique to cache the calls, call-states and answers in a searchable structure, so that the same call in the same state is not re-executed ad-infinitum and the computation is guaranteed to terminate for the Transaction Datalog. The justification or truth maintenance is the process of constructing evidence for the answers derived by tabled evaluation. The tabled execution entailment does not support easy extraction of the justification due to the construction of the derivation tree with results from various parts of the call-answer table. We extract the justification using pointers from the call-answer table to nodes of the derivation tree and additional evidence constructed from the execution trail.

Transaction logic

Horn Transaction Logic (Bonner and Kifer, 1994, 1995, 1996) is an extension of classical logic programming with state changes, enlarging the logic programming syntax with two fundamental ideas: serial conjunction, concurrent execution and elementary transitions. The serial ⊗ and concurrent | conjunctions specifies the order in which the predicates have to be executed and concurrent processes, while the elementary transitions "ins." and "del." specify basic database updates. For instance, a consuming reachability relation program computes a path where walked edges are deleted (see below). The *reach*/2 left recursive predicate specifies that a node can be reached from itself (i.e., the second clause), or the node *Y* can be reached from the node *X* if there is a consuming path from *X* to *Z* and an edge from *Z* to *Y* and consuming the edge (i.e., the first clause).

reach(*X*,*Y*) ← *reach*(*X*,*Z*) ⊗ *edge*(*Z*,*Y*) ⊗ *del.edge*(*Z*,*Y*).
reach(*X*,*X*).

The Transaction Logic's semantics is defined using a few fundamental ideas: transaction execution paths, database states, and executional entailment. A query *reach*(*a*,*X*) is executed on a sequence of states, namely a *path*, if there is a resolution for this query that takes the system from an initial state to a final state of the database. For instance, for an initial database {*edge*(*a*,*b*), *edge*(*a*,*c*), *edge*(*b*,*a*), *edge*(*b*,*d*)} and a query *reach*(*X*,*Y*), the set of all solutions is represented in table 1.

Answer unification	Return state after execution
reach(*a*,*a*)	{*edge*(*a*,*b*), *edge*(*a*,*c*), *edge*(*b*,*a*), *edge*(*b*,*d*)}
reach(*a*,*b*)	{*edge*(*a*,*c*), *edge*(*b*,*a*), *edge*(*b*,*d*)}
reach(*a*,*a*)	{*edge*(*a*,*c*), *edge*(*b*,*d*)}
reach(*a*,*c*)	{*edge*(*a*,*b*), *edge*(*b*,*a*), *edge*(*b*,*d*)}
reach(*a*,*c*)	{*edge*(*b*,*d*)}
reach(*a*,*d*)	{*edge*(*a*,*c*), *edge*(*b*,*a*)}
reach(*b*,*a*)	{*edge*(*a*,*b*), *edge*(*a*,*c*), *edge*(*b*,*d*)}
reach(*b*,*c*)	{*edge*(*a*,*b*), *edge*(*b*,*d*)}
reach(*b*,*b*)	{*edge*(*a*,*b*), *edge*(*a*,*c*), *edge*(*b*,*a*), *edge*(*b*,*d*)}
reach(*b*,*b*)	{*edge*(*a*,*c*), *edge*(*b*,*d*)}
reach(*b*,*d*)	{*edge*(*a*,*b*), *edge*(*a*,*c*), *edge*(*b*,*a*)}
reach(*b*,*d*)	{*edge*(*a*,*c*)}

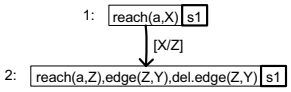
Table 1. Answer unifications and returning states of the database for the transactional query ***reach*(*X*,*Y*)**

Justification for tabled transaction logic

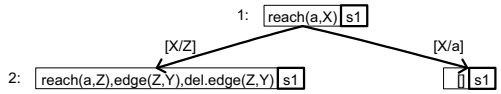
The tabled execution of transaction logic avoids repeated calls, by caching them into a searchable structure together with their proven instances. Because the model theory of transaction logic is defined on the concept of states and paths, the tabling structures contain, beside the calls and answer unifications, the initial and returning states. This solution extends the tabling technique for logic programs (Tamaki and Sato 1986 and Warren 1992) with state and update information, and, to our knowledge, such a tabled technique, hasn't been applied to logics for representing and reasoning about events, actions and their effects until now. The tabled calls-states tuples paired with their answers-returning states are consulted whenever a new call *C* to a tabled predicate is issued. If the call *C* issued in an initial database state *I* is similar to (i.e., subsumed or variant) a tabled call *T* issued in the same state, then the set of answers *A* and returning states *R* for *T* may be used to satisfy *C*. If there is no entry in the call table for *C*, then it is entered into the table and is resolved against program clauses using the normal SLD-like resolution. As each answer is derived during this process, is inserted into the table entry associated with *C* if it contains information not already in *A*. After the answer is added to this set, then it is scheduled to be returned to all calls of *C* stored in a lookup

table. If no answer was found, then the evaluation fails and the execution backtracks.

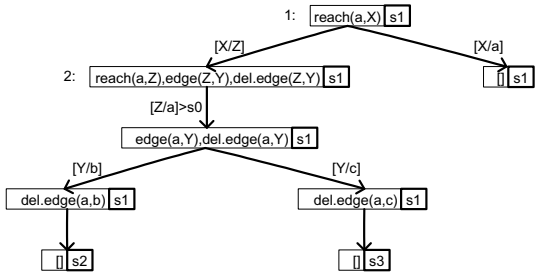
For example, the execution for the consuming reachability query *reach*(*a*,*X*) is delayed on the first refutation path when a variant of a query seen before is found (see Figure 1,a). Another path in the refutation tree succeeds (see Figure 1,b) and computation on the delayed paths should be restarted (see Figure 1,c). The state notation table is exemplified in the table 2, while the (initialState,call, answerUnification,finalState) tuples are exemplified in table 3.



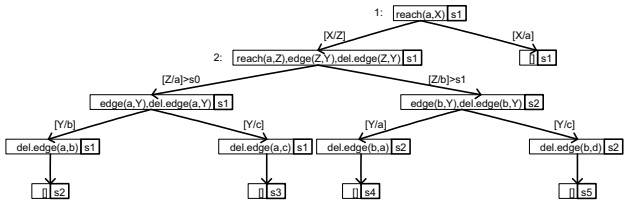
a. Refutation path is delayed because reach(a,Z) is a variant of reach(a,X)



b. Another path in the refutation tree succeeds and computation on the delayed paths should be restarted



c. Another path in the refutation tree succeeds and computation on the delayed paths should be restarted



d. Computation on the delayed paths is again restarted

Figure 1. Part of the deduction tree for *reach*(*a*,*X*)

StateId	State
s1	{*edge*(*a*,*b*), *edge*(*a*,*c*), *edge*(*b*,*a*), *edge*(*b*,*d*)}
s2	{*edge*(*a*,*c*), *edge*(*b*,*a*), *edge*(*b*,*d*)}
s3	{*edge*(*a*,b), *edge*(*b*,*a*), *edge*(*b*,*d*)}
s4	{*edge*(*a*,*c*), *edge*(*b*,*d*)}
s5	{*edge*(*a*,*c*), *edge*(*b*,*a*)}

Table 2. The state reference

initialState	call	answerUnification	finalState
s1	*reach(a,X)*	*reach(a,a)*	*s1*
s1	*reach(a,X)*	*reach(a,b)*	*s2*
s1	*reach(a,X)*	*reach(a,c)*	*s3*
s1	*reach(a,X)*	*reach(a,a)*	*s4*
s1	*reach(a,X)*	*reach(a,d)*	*S5*

Table 3. Part of the call-answer table

The justification or truth maintenance is the process of constructing evidence for the answers derived by the execution of a query. We extract this evidence efficiently using the call-answer table and additional evidence constructed from the execution trail. Extraction of the justification for *reach(a,d)* requires additional information than the call-answer table. During execution we leave a trail of evidence nodes in our deduction to efficiently construct the evidence during query justification, while adding little overhead to the evaluation itself. For instance, the trail for execution above contains:

evidenceEdge(*node*(*G1*,*s1*,*s2*),*node*(*G2,TempState*,*s2*)),
evidenceEdge(*node*(*G1*,*s1*,*s2),node*(*G11a*,*s1*,*TempState*))

These evidences are collected in the justification phase by finding all evidence nodes and extracting the justification from the call-answer tables. Such a justification is shown in figure 2 for the query reach(a,d).

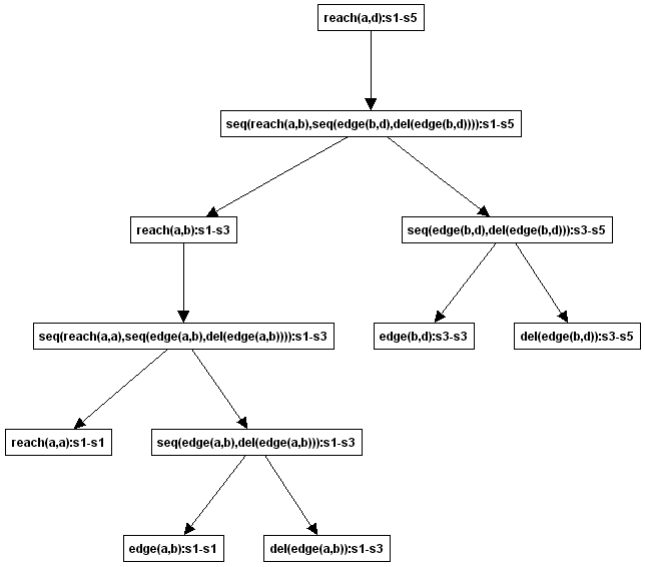


Figure 3. Justification for *reach*(*a*,*d*)

Conclusion

We implemented the justification algorithm for the Tabled Transaction Logic on top of the XSB Prolog to generate derivation graphs. In this scheme, an answer is automatically justified and evidence in built during query evaluation and is presented to the user using a graphical interface (i.e., uDraw in our case). Our current experiments with complex planning applications show that the extra overhead due to the evidence generation is unnoticeable.

References

Bonner, A.J. and Kifer, M. 1994, *An Overview of Transaction Logic*, in Theoretical Computer Science, 133(2), 205-265.

Bonner, A. J., and Kifer, M. 1995, *Transaction Logic Programming (or, a logic of Procedural and Declarative Knowledge*), Technical report, University of Toronto.

Bonner, A. J., and Kifer, M., 1996, *Concurrency and Communication in Transaction Logic*, in JICSLP: 142-156.

Tamaki, H. and Sato, T., 1986, *OLD Resolution with Tabulation*, in International Conference of Logic Programming.

Warren, D.S. 1992, *Memoing for Logic Programs*, in Communications ACM 35(3): 93-111.