Querying Sequential and Concurrent Horn Transaction Logic Programs using Tabling Techniques

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

Abstract

In this poster we describe the tabling techniques for Sequential and Concurrent Horn Transaction Logic. Horn Transaction Logic is an extension of classical logic programming with state updates and it has a SLD-style evaluation algorithm. This SLD-style algorithm enters into infinite loops when computing answers to many recursive programs when they change the underlying state of the knowledge base. We solve this problem by tabling (caching) the calls, call states and answers (unifications and return states) in a searchable structure for the Sequential Transaction Logic, or building a graph for the query and memoize the "hot" vertices (vertices, currently, possible to execute) for the Propositional Concurrent Transaction Logic, so that the same call is not re-executed ad infinum. With these techniques, we can efficiently compute queries to transaction logic programs, and when the underlying programs have the bounded term-depth property (Transaction Datalog) the techniques are guaranteed to terminate. The applications of these techniques promise termination and great improvements in the uses of transaction logic: state-changing systems, artificial intelligence planning, dynamic constraints on transaction execution, workflow modeling and verification, and systems involving financial transactions.

Tabling for Transaction Logic

Sequential Horn Transaction Logic (Bonner and Kifer, 1994, Bonner and Kifer, 1995) is an extension of classical logic programming with state changes, enlarging the syntax with two fundamental ideas: serial conjunction and elementary transitions. The serial conjunction (i.e., ⊗) specifies the order in which predicates have to be executed. The elementary transitions (i.e., ins. and del.) specify basic updates of the current state of the database, executed by an oracle. For instance, a consuming reachability relation program that computes a path where walked edges are deleted is exemplified 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 Sequential Horn Transaction Logic's semantics is also based on a few fundamental ideas: transaction execution paths, database states, executional entailment. Basically, a query (e.g., "*reach*(*a*,*X*)") can execute on a sequence of states (i.e., 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*)} (see Figure 1) and a query *reach*(*a*,*X*), the set of all solutions is represented in table 1.

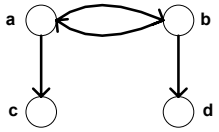


Figure 1. Initial graph

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*)}

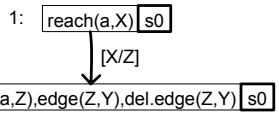
Table 1. Answer unifications and returning states of the database for the transactional query *reach*(*a*,*X*)

Similar to logic programming, Sequential Transaction Logic has a SLD-style resolution algorithm which may enter in infinite loops for recursive queries. For instance, the evaluation algorithm enters into an infinite loop when computing the answers to the query: *reach*(*a*,*X*). Our solution to avoid repeating calls is to table (cache) the calls 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, H. and Sato, T. 1986, Warren, D.S. 1992) with state and updates information.

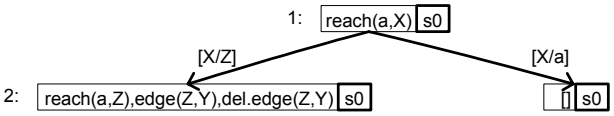
This tabled calls-initial states paired with their answers-returning states is consulted whenever a new call *C* to a tabled predicate is issued. If the call *C* issued in an initial database state *I* is similar to (i.e., subsumed or variant) a tabled call *T* issued in the same state, then the set 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 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.

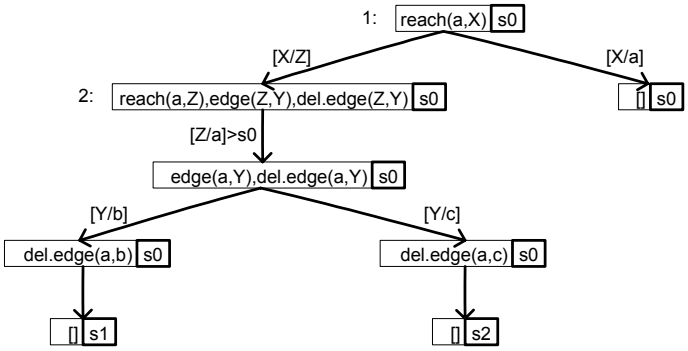
For example, the execution for the consuming reachability query *reach*(*a*,*X*) is delayed on the first refutation path when a variant of a query seen before is found (see Figure 2 a). Another path in the refutation tree succeeds (see Figure 2 b) and computation on the delayed paths should be restarted (see Figure 2 c).



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

Figure 2. Part of the execution for *reach*(*a*,*X*)

Since the call-answer table is indexed on the call and the state in which the call was made, the execution can use a high amount of space and time. To make an implementation feasible, the database states should be signed efficiently for state sharing and comparison. We used an efficient signing technique for states in our implementation in XSB Prolog (see website: http://cs.sunysb.edu/~pfodor/webpageTabledTR).

Concurrent Transaction Logic (Bonner, A. J., and Kifer, M. 1996, Davulcu, H., Kifer, M., Ramakrishnan, C. R., and Ramakrishnan, I. V. 1998, Davulcu, H., Kifer, M. and Ramakrishnan, I.V. 2004) extends the syntax of Transaction Logic with the concurrent conjunction "|" and multi-path concurrent execution. Its execution searches for the successful concurrent executions of the queries. For instance, the query "?- *a*|*b*" searches for the successful, concurrent executions of a and b. Given the following program (⊗ for sequential conjunction, | for concurrent conjunction, and "*ins*." and "*del*." for strict updates (i.e., *ins.t* fails if *t* is already in the current database state, respective, *del.t* fails if *t* fails in the current database state), the execution would fail to find answers (i.e., final states) for the "?- *a*|*b*" query, not knowing how many times to unfold the predicate *b*.

a ← *a* ⊗ *ins.t* ⊗ *ins.t* ⊗ *ins.t*
a ← state
b ← *b* ⊗ *del.t*
b ← state

A sequential tabling on the tuple: [query, current state, hot components] might not terminate. For instance, expanding the predicate *a* ad infinitum would expand the query to infinite. Our solution for tabling Propositional Concurrent Transaction Logic is to construct an AND-OR graph for the query and cache the possible paths to execute.

Conclusion

Using tabling we can compute all the final states with at least one execution for transactional queries to Transaction Logic programs and it is guaranteed to terminate when these programs have the bounded term-depth property. Each final state is also associated with the minimal execution path for that state. Basically, the result is the set of all non-floundering execution paths. Important applications include: artificial intelligence planning, and workflow modeling and verification.

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.

Davulcu, H., Kifer, M., Ramakrishnan, C. R., and Ramakrishnan, I. V. 1998, *Logic Based Modeling and Analysis of Workflows*, Principles of Database Systems.

Davulcu, H., Kifer, M. and Ramakrishnan, I.V. 2004, *CTR-S: A Logic for Specifying Contracts in Semantic Web Services*, WWW.

Tamaki, H. and Sato, T. 1986, *OLD Resolution with Tabulation*, ICLP.

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