Optimizations and Extensions for the Horn Transaction Logic Programs

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

Abstract

My thesis describes optimization techniques and extensions for the Horn Transaction Logic. The Horn Transaction Logic is an extension of the 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 the calls, states and answers in a searchable structure, so that the same call is not re-executed ad infinitum. With these techniques, we can efficiently compute queries to transaction logic programs, and when the underlying programs have the bounded term-depth property, these techniques are guaranteed to terminate. I also present extensions to Transaction Logic, for instance a definite semantics for the existentially quantified values that occur in facts, queries and updates of facts. The applications of these techniques promise 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

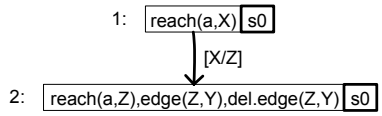
Sequential Horn Transaction Logic (Bonner and Kifer, 1994, Bonner and Kifer, 1995) is an extension of classical logic programming with state changes, adding to the syntax two fundamental ideas: serial conjunction and elementary transitions. The serial conjunction operator ⊗ specifies the order in which 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 a strict oracle, i.e., *ins.t* fails if *t* is already in the current database state, and *del.t* fails if *t* fails in the current database state. 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*).

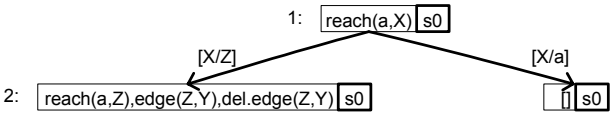
The Sequential Horn Transaction Logic's semantics is also based on a few fundamental ideas: transaction execution paths, database states and executional entailment. A query (e.g., "*reach*(*a*,*X*)") can execute on a sequence of states (i.e., a path), if there is a resolution for this query that takes the system from an initial state to a final state of the database.

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 and Sato 1986, Warren 1992) with state and updates information. These records (i.e., calls, initial states, answer unifications and final 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 SLD-like resolution. As each answer is derived during this process, it 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.

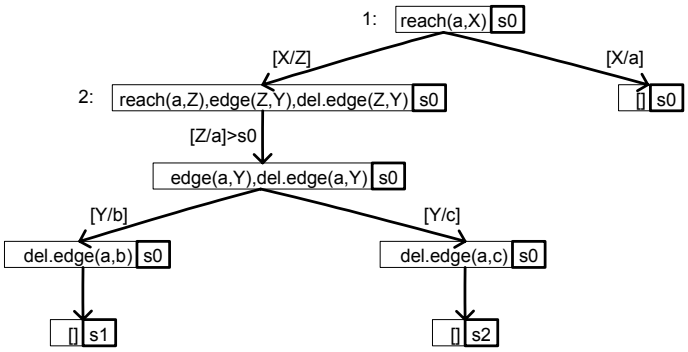
For example, the execution for the consuming reachability query *reach*(*a*,*X*) is delayed on the first refutation path when a variant of a query seen before is found (see Figure 1 a). Another path in the refutation tree succeeds (see Figure 1 b) and computation on the delayed paths should be restarted (see Figure 1 c).



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 1. 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).

Existentially Quantified Values for the Transaction Logic

I present a **definite semantics for the existential values in queries and updates of facts** by generating unique identifiers to represent existential values in facts. If we want to insert a fact *P*(*X*,*Y*) in the database, then we check if all arguments are not variables. For un-instantiated arguments, we generate special object identifiers with a special semantics. The common human intuition follows a definite semantics, i.e., if a fact with existentially quantified values is substituted by a definite fact, then the first fact is not considered anymore. For instance, if only the following two facts are in the database: *parent*("*Joseph*",*parent*("*Joseph*")) (with the Skolem function *parent*("*Joseph*")) and *parent*("*Joseph*","*Paul*"), then the answer to an aggregate query: *count*{*X* | *parent*("*Joseph*",*X*)} should return 1.

These existential values extend the usual domain where we interpret constants (with a unique-name-assumption-as-failure property: x ≠ y unless we can prove x = y) with a domain for interpreting the new *eX* symbols under the assumption that an existential might be equal to some constant, but we do not know that unless it follows from some other information. For the *eX* symbols we leave the possibility that some *eX* might be equal with some constants even if we cannot prove that.

The Transaction Logic's semantics is a modal-like semantics, where each state represents a database, and each elementary update causes a transition from one state to another. At this point, we extend this model theory, with the fact that: in each state, the facts with *eX* values that are subsumable by real (ground) facts can be eliminated. For instance, the fact *parent*("*Joseph*",*eX*) can be eliminated if there is also *parent*("*Joseph*","*Paul*") in the current database. The truth of transactional queries is still determined on paths structures, i.e, sequences of states. This semantics would return an answer if and only if the answer is definite in all possible models for the program and the execution path.

Example 1. If the current state of the database is: {*parent*("*Joseph*",*eX*), *parent*("*Joseph*",*"Paul"*)}, then *count*{*X* | *parent*("*Joseph*",*X*)} = 1 because the fact *parent*("*Joseph*",*eX*) is subsumed by the fact *parent*("*Joseph*",*"Paul"*).

Example 2. If the current state of the database is: {*parent*("*Joseph*",*eX*) }, then
count{*X* | *parent*("*Joseph*",*X*)} = 1 because we know that there is at least one fact *parent*("*Joseph*",*eX*) in the database.

Example 3. The sequence of operations:
ins.parent("*Joseph*",*eX*) ⊗ *ins.parent*("*Joseph*",*"Paul"*)} ⊗ *del.parent*("*Joseph*","*Paul*") ⊗ *parent*(?*X*,?*Y*)
would fail because it would take the database through the path of states:
{*parent*("*Joseph*",*eX*)}→{*parent*("*Joseph*","*Paul*")}→{ }
and the query *parent*(?*X*,?*Y*) would fail in the empty state.

Conclusion

Using tabling and existentially quantified values we can compute all the final states with at least one minimal execution path for transactional queries and represent incomplete effects in applications of transaction logic, such as: 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.

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.