Pseudo-Naive Evaluation

Donald A. Smith Mark Utting
Department of Computer Science
The University of Waikato
Private Bag 3105, Hamilton, New Zealand
Email: {dsmith,marku}@cs.waikato.ac.nz

June 11, 2010

Abstract

We introduce pseudo-naive evaluation, a method for execution of mixed top-down/bottom-up logic programs and deductive databases. The method is intermediate in power between naive evaluation and semi-naive evaluation. Pseudo-naive evaluation adds a data-driven component to naive evaluation without explicitly collecting the 'delta' sets of new facts derivable at each iteration. Instead, it identifies certain body atoms as 'triggers' and collects an abstraction of the delta sets, thereby simplifying the implementation. A rule is invoked only when new tuples for its trigger atoms are derived. Pseudo-naive evaluation is most efficient on strongly-stratified programs: programs for which all (positive and negative) bottom-up recursion is mediated by an increasing temporal parameter. However, the method can still be used on programs with general recursion, by using either top-down calls, timestamped tuples to represent delta sets, or tuple-at-a-time bottom-up execution. A desirable feature enjoyed by our system is that it runs piggyback on most logic programming implementations, but performance is good because most of the code is compiled and executed by the native Prolog system.

Proceedings of the Tenth Australasian Database Conference, Auckland, New Zealand, January 18–21 1999

1 Introduction

Two common evaluation techniques for bottom-up deductive database systems are naive evaluation and semi-naive evaluation [Ramakrishnan et al., 1992]. Naive evaluation evaluates all rules repeatedly, so is quite inefficient for many programs. Semi-naive evaluation improves efficiency by tracking newly-derived tuples (delta-sets) and only evaluating rules that use those tuples.

This paper describes an intermediate approach, which is more efficient than naive evaluation, but can be implemented more easily on top of standard Prolog systems than semi-naive evaluation. Essentially, our implementation compiles the bottom-up clauses into standard top-down Prolog clauses, then adds some simple control predicates that use a queue structure (an abstraction of the delta-sets of semi-naive evaluation) to execute the rules in a bottom-up fashion. Advantages of this approach are that it reuses the mature implementation and analysis techniques of standard Prolog, it naturally allows programmers to mix top-down and bottom-up code, it supports declarative programming of I/O and change, and it is appropriate as a target language for compilation of semi-naive evaluation for deductive databases.

2 Naive and Semi-Naive Evaluation

Naive evaluation [Ramakrishnan et al., 1992] is a direct implementation of the bottom-up, fixed point semantics of logic programs [Lloyd, 1987], wherein the interpreter computes a monotonically increasing set of atoms derivable from the program clauses. At each step i of the bottom-up computation, an atom H is added to the set S_i if it appears as the head of an instance of a program clause $H \leftarrow B$ whose body atoms B all appear in the set S_{i-1}.

Naive Evaluation

$$\begin{aligned} S_0 &= \{p \mid p \text{ is a fact}\} \\ S_{i+1} &= S_i \cup \{H\mu \mid (H \leftarrow B) \in \mathit{Rename}(P) \\ &\quad \wedge (B\mu \subseteq S_i) \wedge (H\mu \notin S_i)\} \quad (i \geq 0) \\ S &= \cup_{i=0\ldots\infty} S_i. \end{aligned}$$

The condition $H\mu \notin S_i$ (where μ is the most general unifier) guarantees that only atoms not derived in previous steps are added at time $i+1$. An interpreter[1] for naive evaluation can be written easily in Prolog (Figure 1). Note that the calls to `Body` and `Head` in the first clause for `naive_evaluation/0` access only

[1] The interpreter implements a semantics slightly different from the abstract definition because in the interpreter tuples derived at step i are 'visible' immediately, whereas in the abstract definition they are not visible until step $i+1$.

```
naive_evaluation:-
    rule(Head,Body),
    Body,                 % Try proving the Body.
    \+ Head,              % Check that Head is new.
    assert(Head),
    \+ some_change,       % Fail if some_change already.
    assert(some_change),
    fail.                 %Failure driven loop.
naive_evaluation:-
    (retract(some_change) ->
       naive_evaluation   % Iterate.
     ; true).             % Done!
```

Figure 1: Prolog Interpreter for Naive Evaluation

tuples previously derived by bottom-up execution; they do not invoke rules top-down. Also note that the code would need subsumption checking to handle non-ground facts.

The inefficiency of naive evaluation results from two phenomena:

Naive-evaluation is not data-driven. Each rule $H \leftarrow B$ is tried, even if no (new) facts have been derived for atoms appearing in the body of the rule.

Naive evaluation duplicates work. Each fact derivable at iteration i of the bottom-up computation is also derivable at iteration j for all $j > i$.

An example of a program for which naive evaluation would result in duplication is the following.

```
ancestor(A,D):- parent(A,D).
ancestor(A,D):- ancestor(A,Tmp),parent(Tmp,D).
```

This program would loop using Prolog evaluation, due to the leftmost recursion in the second rule. Using naive evaluation, at iteration 0, S_0 gets assigned the set of `parent/2` facts using the definition of S_0. At iteration 1, only the first rule fires: S_1 gets the union of S_0 and the set of `ancestor/2` facts that result from copying `parent/2` facts from S_0. The second rule cannot fire because `ancestor/2` is still empty. At iteration 2, the first rule fires again, rederiving tuples derived at iteration 1, but the check for duplicates prevents tuples from being reasserted; additionally, the second rule fires, deriving tuples for the all pairs in the grandparent relation. Subsequent iterations rederive all tuples derived in earlier iterations, though duplicate removal prevents tuples from being reasserted. Execution halts when an iteration is reached where no changes can be made.

Semi-naive evaluation [Ramakrishnan et al., 1992] overcomes both inefficiencies mentioned above, by making sure that at least one newly derived fact is used to satisfy the body of a program clause.

Semi-naive Evaluation

$$\begin{aligned} S_0 &= \delta_0 = \{p \mid p \text{ is a fact}\} \\ \delta_{i+1} &= \{H\mu \mid (H \leftarrow B) \in Rename(P) \\ &\qquad \wedge (B\mu \cap \delta_i \neq \emptyset) \wedge (B\mu \subseteq S_i) \wedge (H\mu \notin S_i)\} \quad (i \geq 0) \\ S_{i+1} &= S_i \cup \delta_{i+1} \quad (i \geq 0) \\ S &= \cup_{i=0\ldots\infty} S_i \end{aligned}$$

For each i, δ_i contains the facts newly derived at iteration i. The condition $B\mu \cap \delta_i \neq \emptyset$ is the test that avoids duplication of work and makes computation data-driven, since only rules that have newly derived enabling data are tried.

We forgo presenting a Prolog interpreter for semi-naive evaluation. Indeed, the difficulty of writing such an interpreter – the efficient management of the delta sets is the main stumbling block — is a major reason for our developing pseudo-naive evaluation.

Using semi-naive evaluation on the `ancestor/2` program, at iteration 0 δ_0 is set equal to the `parent/2` relation. At iteration 1, only the first rule fires; δ_1 gets set equal to the `ancestor/2` facts copied from δ_0. At iteration 2, the first rule is not even tried (unlike naive evaluation), because δ_1 contains no `parent/2` facts so the condition $B\mu \cap \delta_1 \neq \emptyset$ fails. The second rule fires, setting δ_2 to the *new* `ancestor/2` facts. Subsequent iterations are appropriately data driven and avoid rederiving all previous iterations' results.

Before describing pseudo-naive evaluation we must explain two preliminary notions: timestamps and strong-stratification.

3 Starlog, an Executable Temporal Logic

We have designed and implemented pseudo-naive evaluation for use as the execution mechanism of Starlog [Liu and Cleary, 1994, Cleary, 1990]. Starlog is an executable temporal logic [Tansel et al., 1993, Baudinet et al., 1993, Chomicki, 1994, Orgun and Ma, 1994, Abadi and Manna, 1989]. The primary motivation for using a temporal logic is that it enables declarative expression of I/O, destructive assignment, and other sorts of dynamic change. The present paper shows, in addition, that time can be used to structure, and simplify the implementation of, deductive database programs.

A Starlog program consists of a set of timed and untimed program clauses. Untimed program clauses are identical in form and in execution to standard top-down Prolog clauses. Timed program clauses are distinguished from untimed clauses in that the head atom and zero or more of the body atoms have

an additional distinguished argument, the timestamp, written as the second argument of the binary infix operator `@`. Timestamps must evaluate to natural numbers.[2] A typical timed clause looks like

```
p(X)@T :- q(X,Y)@T, d(D), r(Y)@T-D.
```

Its declarative reading is roughly: "If `q(X,Y)` holds at time `T`, if `d(D)` holds, and if `r(Y)` holds at time `T-D`, then `p(X)` holds at time `T`." Timed program clauses are executed either top-down or bottom-up, depending on the presence or absence, respectively, of a declaration, such as `:- top_down p/1`, specifying that all clauses for a given head predicate symbol shall be handled top down. Thus, by default timed clauses are bottom-up.

The heads of bottom-up program rules represent facts to be made true, or actions (e.g., output) to be performed, at a given time. Atoms in the body represent conditions to be checked. Conditions are facts derivable either by earlier bottom-up rules, by top-down calls, or by builtin environmental checks such as for the availability of data on an I/O channel. Crucially, clauses are causal: the bodies of bottom-up rules must refer to the past and may not refer to the future. (This will be made more precise in the next section.) Thus Starlog rules follow the "Imperative Future" paradigm [Orgun and Ma, 1994].

The typical structure of a Starlog program is basically bottom-up: the outermost loop of execution, which interacts with the external world, is bottom-up and data-driven. (By analogy, in Haskell[Hudak and et al., 1992] or Mercury[Somogyi et al., 1996, Somogyi, 1989], interaction with the real world is mediated by the outermost, singly threaded state of the world.) The reason for this is that in implementing the imperative future paradigm, Starlog uses the facts true at the current time, along with a record of some facts true in the past, to compute the state of the world in the future. If one tried to execute Starlog top-down, say, by running a query `p(X)@100` or `p(X)@T`, then the system would have to backward chain on the rules and *guess* (recursively) which clauses lead to the query's being true. This may result in a lot of search. Executed bottom-up, a Starlog program involves model construction rather than search.

Nonetheless, we have found that it is most natural and efficient to specify that certain relations are to be evaluated top-down. Since Starlog does not support magic sets [Ramakrishnan et al., 1992], having top-down predicates is useful for those computations needing a goal-driven component. Furthermore, facts derived bottom-up are usually memoized, as in typical deductive databases, while facts derived top-down are not (by default) memoized. Since Starlog allows data structures as terms (unlike many temporal deductive databases), relations such as `append/3` and `sort/2` that create numerous intermediate data structures and that are likely used but once per input arguments, are best executed top-down. Lastly, rules that would need arithmetic constraints if executed

[2] Real time timestamps are supported as well, but for the purposes of this paper, natural number timestamps suffice.

bottom-up, due to unbound variables, can often be executed top-down without constraints, since the appropriate (numeric) arguments will bound at call time.

Having both top-down and bottom-up control allows the programmer to take appropriate advantage of the complementary strengths of both Prolog and bottom-up deductive database technology.

4 Strongly-Stratified Programs

A *strongly stratified* program is one that does not contain any simultaneous recursion through timed predicates (those with heads of the form `H@T`). Instead, all recursion must be stratified through time.[3] That is, if the call graph of the program contains a path from atom $p(A_1, \ldots, A_n)@T$ (or literal $\neg p(A'_1, \ldots, A'_n)@T'$) in the body of a rule to an atom $p(A'_1, \ldots, A'_n)@T'$ in the head of some rule, then T must be strictly greater than T'.

In other words, within each time step, bottom-up computation is hierarchical (non-recursive). We emphasize that negation too must be strongly stratified. Note that a strongly stratified programs can still contain recursion, but the recursion is restricted so that all bottom-up recursive cycles involve increasing time. The second clause for `ancestor/2` is not strongly stratified because of the recursion in the second clause. (However, since the recursion is linear[4], it presents no problem; see below.)

Furthermore, the compiler must be able to statically derive the precedence relation. All predicates callable at a given time step must be ordered, statically, by the compiler, into strata such that if predicate q appears (positively or negatively) in the body of a bottom-up clause with head p, then q is in a strictly lower strata than p. This restriction on the structure of programs is by design. By analogy, just as Mercury places upon the programmer the obligation to write a well-moded program, so Starlog places upon the programmer the obligation to write a program for which the strong stratification is derivable by the compiler.

4.1 Compilation of Strong Stratification

It is tedious (but perhaps good software engineering) for programmers to explicitly specify the stratification partial order, so the first stage of our Starlog compiler infers it, by analyzing all the bottom-up clauses of the given program, then generating a suitable partial order. This is done by the following Prolog code in the compiler:

```
...
\+ (is_bottom_up_clause(Head@Time, Body),
```

[3] Except that untimed predicates may be recursive, because they are executed top-down by Prolog and cannot call timed predicates. Conceptually they are executed instantaneously at whatever time instant that they are called.

[4] A rule is linear [Ullman, 1989] if only a single body atom is recursive with the head.

```
        path(Body, Path, Constraints),
        member(Call@Ti, Path),
        \+ stratified(Call@Ti, Head@Time, Constraints)),
    tsort,
    ...
```

Note that the Prolog code `\+(P, \+ Q)` is best read as *for all solutions to P, Q is true*. The `is_bottom_up_clause/2` predicate iterates through each of the bottom-up clauses in the Starlog program being analyzed. The `path/3` predicate converts Body to disjunctive normal form and for each disjunct, binds Path to the list of the timed calls within that disjunct and binds Constraints to a list of all the arithmetic constraints within that disjunct. For example, if Body is:

```
p(X)@T, T1 is T-1,
(X = 0  ->  q(Y)@T1  ;  r(Y)@T2, T2 < T1)
```

then `path(Body,Path,Constraints)` will return two solutions:

```
Path = [p(X)@T, q(Y)@T1] Constraints = [T1 = T-1, X = 0] ;
Path = [p(X)@T, r(Y)@T2] Constraints = [T1 = T-1, T2 < T1].
```

The `stratified(Call@Ti, Head@Time, Constraints)` predicate first tries to prove that `Ti<Time` follows from `Constraints` (it uses a simple theorem prover to do this). If it cannot prove this, then it proves that `Ti=<Time` follows from `Constraints`[5] and asserts an ordering tuple `is_before(P/N, H/M)`, where `P/N` and `H/M` are the outermost functors of `Call` and `Head`, respectively.

Finally, `tsort/0` performs a topological sort on the `is_before/2` tuples and generates an error if they contain any cycles, or flattens them into a total order over the predicates otherwise. This total order is used by the pseudo-naive evaluator to determine the order of execution within each time step.

5 Pseudo-Naive Evaluation

Pseudo-naive evaluation optimizes naive-evaluation by adding a data-driven component to computation — thereby overcoming the first source of inefficiency mentioned in Section 2 — without maintaining 'delta sets' δ_i. Section 7 discusses how we overcome the second problem of naive evaluation, duplication of work, and how pseudo-naive evaluation can be used for programs involving simultaneous recursion.

For hierarchical or strongly stratified programs, delta sets provide no benefit, due to the lack of simultaneous recursion. However, naive evaluation would still

[5] It gives a warning if it fails to prove this, because it means that either the theorem prover is not strong enough, or the call is referencing future tuples which is a programming error.

be inefficient since naive evaluation would try executing each program rule, whether or not new facts have been derived for atoms in the body of the rule.

Pseudo-naive evaluation adds a data-driven component to naive evaluation by maintaining, at each iteration i, a *token set* τ_i, rather than a delta set δ_i. A token set is an abstraction of a delta set. A pseudo-naive interpreter uses the token set to skip over rules that cannot possibly fire, since no new facts have been derived for atoms in the bodies of those rules. After presenting the basic method, we present some important refinenments.

Pseudo-naive Evaluation (unrefined)

$$\begin{aligned} S_0 &= \{p \mid p \text{ is a fact}\} \\ \tau_i &= \{p/n \mid \exists t_1, \ldots, t_n,\ p(t_1, \ldots, t_n) \in S_i\} \quad (i \geq 0) \\ S_{i+1} &= S_i \cup \{H\mu \mid (H \leftarrow B) \in \mathit{Rename}(P) \\ &\qquad \wedge (\mathit{preds}(B) \cap \tau_i \neq \emptyset) \wedge (B\mu \subseteq S_i) \wedge (H\mu \notin S_j\} \quad (i \geq 0) \\ S &= \cup_{i=0\ldots\infty} S_i \end{aligned}$$

In unrefined pseudo-naive evaluation the token set τ_i consists of the set of relation names (predicate plus arity) which have received new facts at iteration i. It is clear that the method is sound and complete, since each τ_i contains all relation names that have received new tuples in the previous iteration.

But if pseudo-naive evaluation is used on non-strongly stratified programs, duplicated work may occur. For example, for the `ancestor/2` program, at iteration 0, S_0 gets all `parent/2` facts, and τ_0 gets {`parent/2`}. At iteration 1, both rules are tried, because both contain `parent/2` facts; only the first succeeds and `parent/2` facts are copied into `ancestor/2`; τ_i gets {`ancestor/2`}. At iteration 2, the first rule is not tried (unlike in naive evaluation), while the second rule is tried; again τ_i gets {`ancestor/2`}. During subsequent iterations (until the fixpoint is reached), only rule 2 is tried, but it rederives tuples derived at earlier iterations. With pseduo-naive evaluation only rules having some new data are tried, but if rules have simultaneous recursion, work will be repeated, since delta sets are not maintained to distinguish new tuples from old. We remedy this problem in Sections 5.2 and 7.

5.1 Refinement 1: Triggers

In the `ancestor/2` example above, at iteration 1 both rules are tried, because both contain `parent/2` facts. However, the existence of `parent/2` tuples alone is insufficient to cause the second rule to fire. The second rule need be tried only when new `ancestor/2` tuples have been derived. We say that `ancestor/2` is the *trigger* relation for the second rule.

The interpreter for pseudo-naive evaluation shown in Figure 2 is a refinement of the previous abstract definition. For each rule we identify one or more trigger relations, which appear as body goals. A trigger relation is a further abstraction

```
:- mode pseudo_naive_evaluation(+).
pseudo_naive_evaluation([]).   % No more tokens.
pseudo_naive_evaluation([Token|RestTokens]):-
    setof(NewToken, enables(Token,NewToken), NewTokens),
    merge_tokens(NewTokens,RestTokens,Tokens),
    pseudo_naive_evaluation(Tokens).

:- mode enables(+,-).
enables(Token,NewToken):-
    rule(Token,Head,Body), % Index on Token
    Body,          % Try proving the body.
    \+Head,        % Make sure it's new.
    assert(Head),
    trigger(Head,NewToken). % Generate tokens for dependent rules.
```

Figure 2: Prolog Interpreter for Pseudo-naive Evaluation

of a delta set. We need only attempt the rule when a tuple for a trigger atom is derived. Furthermore, each τ_i contains a set of rule *indices* that may fire in the next iteration, instead of a set of relation names. A rule index appears in τ_i if a tuple for some trigger atom in the body of that rule has been derived in the previous step. If r appears in τ_i, then the interpreter has to check if rule r can fire.

Note that the iteration index i of τ_i is represented as part of the tokens. So each token is a pair $\langle t, r \rangle$, consisting of a timestamp t and the index r of a rule to be tried. The procedure `merge_tokens(+,+,-)` sorts tokens according to the stratification order derived by the compiler. If an atom `p(T1,...,Tn)` appears as the trigger in the body of some program rule with index I, then the fact `trigger(p(_,...,_),I)` must appear in the compiled program. Notice that the arguments are not stored in the trigger; hence the abstraction.

For example, the facts `trigger(parent(_,_),1)` and `trigger(ancestor(_,_),2)` must appear in the compiled code for the `ancestor/2` program. Using our refined version of pseudo-naive evaluation, at iteration 1 only the first rule is tried, because `parent/2` is a trigger only for the first rule.

Section 6 describes how triggers are chosen.

We note some similarity between our pseudo-naive interpreter and the techniques described in [Wunderwald, 1996] for converting bottom-up rules into Prolog rules.

5.2 Refinement 2: Tuple-at-a-time Immediate Execution

So far we have assumed that when a new tuple is derived, the tokens it triggers are added to the queue and that any rules that these tokens enable won't fire until later on. Relaxing this assumption by allowing tuples to immediately

trigger other rules is often a big win. Suppose a tuple $q(T_1, \ldots, T_n)@T$ appears as trigger in the body of some rule
$p(U_1, \ldots, U_m)$`@T :-` $q(T_1, \ldots, T_n)$`@T, .....`
in which the timestamp of the head is identically `T`. Suppose further that it is safe according to the stratification order to fire the rule immediately (no rule earlier in the stratification order needs to fire first, due, say, to a negation or to the possibility of looping — see next section). Then as soon as a q/n tuple is derived, the p/n rule is tried using that tuple and its arguments as input.

If the compiler can identify that all rules which consume a tuple are such that the tuple is in the trigger position, then the tuple need not be saved at all; such *virtual* triggers exist only in the runtime stack and registers of the implementation, thereby saving copying and garbage collection time.

Tuple-at-a-time bottom-up execution is used in Section 7 for handling linear recursions.

6 Compilation for Pseudo-Naive Evaluation

The most important aspect of compiling programs for the pseudo-naive evaluator is choosing one or more appropriate triggers for each clause. This is important, because clauses are only evaluated when triggered, so the choice of trigger can affect both correctness (if a clause is not triggered at times when it might produce output, then outputs will be lost) and efficiency (tuples that occur infrequently make better triggers, because rules are evaluated less often).

Currently, our compiler automatically chooses a trigger for two simple cases, which cover the majority of clauses, and evaluates the remaining clauses at every time step. However, we also allow the programmer to annotate a clause to manually define one or more triggers for a clause, which overrides the compiler's choice. (For example, for real-time execution, it is often sufficient to evaluate certain clauses periodically, e.g., every 0.5 seconds.) This is more flexible, but means that the programmer is responsible for completeness (choosing the wrong trigger may cause output tuples to be lost). When adding triggers by hand, we have occasionally found it useful to add several triggers to a single clause. It is not clear to us yet whether sophisticated triggering like this can be chosen automatically, or even checked for correctness by the compiler.

Given a clause, `H@T <-- Body`, (to simplify discussion, we assume in this section that all clauses are in Horn clause format) the two strategies used by the compiler are:

1. If `Body` contains some positive atom, $A@U$, such that $T - U$ is a constant natural number K (e.g. if T and U are identical, then $K = 0$), then the compiler uses A as the trigger, and evaluates the clause exactly K time units after any A tuple is created. The compiler does this by generating the following `rule/3` and `trigger/2` clauses (ID is a unique integer generated for each clause):

```
rule(ID@T, Head@T, Body).
trigger(A@U, ID@T) :- T is U+K.
```

Note that there may be several atoms $A@U$ that satisfy this criterion. The choice between them does not affect soundness, but may affect efficiency. Ideally, the atom that is least frequent triggered should be chosen. Currently, we choose the left-most one that satisfies the above criterion, but it would be interesting to extend the compiler to use statistical information from previous runs as the basis for this choice.

2. If `Body` contains some positive atom, $A@U$ which is provably the latest in the body (relative to the stratification order) then it is safe to trigger the clause at time U (it will produce outputs at time T, which may be in the future, but at least no outputs will be missed). The compiler generates :

```
rule(ID@U, Head@T, Body).  % Note that Body contains A@U
trigger(A@U, ID@U).
```

A disadvantage of this strategy is that some clauses may create many tuples in the future, which slows the system down.

In reality, a rule that we have described as being compiled into: `rule(ID@U, Head@T, (P@U, Rest...).` is actually compiled into something like: `rule(ID, U, Head@T) :- P@U, Rest...` so that the body of the rule is compiled directly by the underlying Prolog compiler and so that first-argument indexing means that each token directly selects the correct triggered clause.

7 Handling Simultaneous Recursion

There are at least four ways for a pseudo-naive interpreter to deal with programs that are not strongly stratified.

1. Make the culprit procedures top-down and use Prolog evaluation. For very many procedures — those efficiently executable by Prolog — this option will be entirely appropriate, even ideal.
2. Make the procedure top-down and use tabling (OLDT resolution [Sato and Tamaki, 1986]), as in XSB [Sagonas et al., 1994]. This option will work for a larger class of programs (e.g., those like the `ancestor` program with left recursion) and, unlike bottom-up computation without magic sets, has the advantage of being goal driven. Our implementation of Starlog runs also under XSB, and this option has been successfully used.
3. Use tuple-at-a-time bottom-up triggering, provided the simultaneous recursion is linear, as is the case for the `ancestor` rules. Each derived `ancestor/2` tuple then gets joined with the `parent/2` relation to derive

additional `ancestor/2` tuples, using a simple stack to handle multiple firings.

4. Add explicit timestamps to make the recursion strongly stratified. This option is effectively semi-naive evaluation, using the timestamped relations to hold the delta sets.

Here we briefly expand on the final technique. A compiler for semi-naive evaluation transforms the input program to a version utilizing delta sets. As pointed out in [Freire et al., 1996], in semi-naive evaluation "a form of timestamp denoting the iteration number is explicitly represented in both answers and rules." (Timestamps are explicit in the implementation's transformed program, not in the original program.) Consider again the `ancestor` program. Using timestamps, the transformed program is as follows:

```
ancestor_delta(A,D)@0:- parent(A,D).
ancestor_delta(A,D)@T:-
        ancestor_delta(A,Tmp)@T-1,
        parent(Tmp,D),
        not(exists(U, (U<T,ancestor_delta(A,D)@U))).

ancestor(A,D):- ancestor_delta(A,D)@T.
```

At each time `T`> 0, the second clause is used to derive tuples that are newly enabled by tuples derived at time `T-1`. The negated call at the end of the recursive clause is needed to assure that a tuple `ancestor_delta(A,D)@U` derived at some time `U`$<$`T` in the past is not re-derived at time `T`. Timestamped relations correspond directly to delta sets. Now, for this fourth technique to be generally useful, a way is needed to *modularize* time, so that the code can be called at times other than `T=0`. This is a topic of our current research, on which we plan to report elsewhere.

8 Conclusion

We have presented a novel implementation technique, pseudo-naive evaluation, intermediate in power between naive evaluation and semi-naive evaluation. The essential idea is to prohibit *simultaneous* bottom-up recursion by requiring all (non-linear) recursion to be either top-down or mediated by a temporal stratification. Pseudo-naive evaluation optimizes execution of such *strongly-stratified* programs by introducing a data-driven component (triggers) to naive evaluation, without maintaining the explicit delta sets of semi-naive evaluation. But pseudo-naive evaluation can still handle non-strongly stratified programs, either by using top-down calls, by converting to strongly stratified code via the addition of explicit time parameters for representing delta sets, or (provided the recursion is linear) by the use of bottom-up tuple-at-a-time execution. We have

described an implementation of pseudo-naive evaluation in Starlog, a temporal logic programming language. The language supports both bottom-up, data-driven execution and top-down, goal-driven execution, since top-down calls are allowed in rule bodies. Programs are compiled into Prolog rules executed by an event list driven top-level driver; the system can thereby take advantage of mature Prolog implementation technology and programming techniques.

Space limitations have prevented us from discussing applications of pseudo-naive evaluation; these include an elevator simulator with X Windows GUI, an individual-based ecology simulation, and an implementation of the graphplan [Blum and Furst, 1997] planning algorithm. Areas of current work include automatic compilation of complex triggers; garbage collection and annotations to specify tuple longevity; abstract, modular, and real time; and comparative performance evaluation.

Acknowledgements

We thank John Cleary and Lunjin Lu for their considerable input into this research.

References

[Abadi and Manna, 1989] Abadi, M. and Manna, Z. (1989). Temporal logic programming. *Journal of Symbolic Computation*, 8(3):277–295.

[Baudinet et al., 1993] Baudinet, M., Chomicki, J., and Wolper, P. (1993). Temporal deductive databases. In [Tansel et al., 1993].

[Blum and Furst, 1997] Blum, A. and Furst, M. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300.

[Chomicki, 1994] Chomicki, J. (1994). Temporal query languages: A survey. In Gabbay, D. M. and Ohlbach, H. J., editors, *Temporal Logic: First International Conference*. Springer-Verlag, LNCS 827.

[Cleary, 1990] Cleary, J.G. (1990). Colliding pucks solved in a temporal logic. In *Proc. Distributed Simulation Conference*.

[Freire et al., 1996] Freire, J., Swift, T., and Warren, D.S. (1996). Taking i/o seriously: Resolution reconsidered for disk. Technical report, SUNY Stony Brook.

[Hudak and et al., 1992] Hudak, P. and et al., P. Wadler (1992). Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *SIGPLAN Notices*.

[Liu and Cleary, 1994] Liu, M. and Cleary, J.G. (1994). Declarative updates in deductive databases. *Journal of Computing and Information*, 1:1435–1446.

[Lloyd, 1987] Lloyd, J.W. (1987). *Foundations of Logic Programming*. Springer-Verlag, 2nd edition.

[Orgun and Ma, 1994] Orgun, Mehmet A. and Ma, Wanli (1994). An overview of temporal and modal logic programming. In Gabbay, D. M. and Ohlbach, H. J.,

editors, *Temporal Logic: First International Conference*. Springer-Verlag, LNCS 827.

[Ramakrishnan et al., 1992] Ramakrishnan, Raghu, Srivastava, Divesh, and Sudarshan, S. (1992). Efficient Bottom-up Evaluation of Logic Programs. In Vandewalle, J., editor, *The State of the Art in Computer Systems and Software Engineering*. Kluwer Academic Publishers.

[Sagonas et al., 1994] Sagonas, K., Swift, T., and Warren, D.S. (1994). XSB as an Efficient Deductive Database Engine. In *SIGMOD*.

[Sato and Tamaki, 1986] Sato, T. and Tamaki, H. (1986). Old resolution with tabulation. In Shapiro, E., editor, *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science, Berlin Heidelberg. Springer-Verlag.

[Somogyi, 1989] Somogyi, Z. (1989). *A Parallel Logic Programming System Based on Strong and Precise Modes*. PhD thesis, University of Melbourne.

[Somogyi et al., 1996] Somogyi, Z., Henderson, F. J., and Conway, T. C. (1996). The execution algorithm of mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64.

[Tansel et al., 1993] Tansel, A., Clifford, J, Gadia, S., Jajodia, S., Segev, A., and (editors), R. Snodgrass, editors (1993). *Temporal Deductive Databases: Theory, Design, and Implementation*. Benjamin/Cummings.

[Ullman, 1989] Ullman, J. (1989). *Principles of Database and Knowledgebase Systems*, volume 2. Computer Science Press.

[Wunderwald, 1996] Wunderwald, J. (1996). *Adding Bottom-up Evaluation to Prolog*. PhD thesis, Technical University of Munich.