Pseudo-Naive Evaluation: Mixing Top-down and Bottom-up Execution for Strongly Stratified Logic Programs

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

May 30, 1998

Abstract

We introduce a new bottom-up method, pseudo-naive evaluation, for execution of mixed top-down/bottom-up logic programs. 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, pseudo-naive evaluation collects an abstraction of the delta sets, thereby simplifying the implementation. Pseudo-naive evaluation is most efficient on strongly-stratified programs: those programs for which all (positive and negative) bottom-up recursion is mediated by an increasing temporal parameter. We describe our compiler and interpreter and several interesting applications: an elevator simulator with GUI, a robot planner, and a graphical, individual-based ecology simulation. A desirable feature enjoyed by our system is that it runs piggyback on most logic programming implementations, as it uses the standard dynamic database (or equivalent library packages); but performance is good because most of the code is (compiled and) executed by the native Prolog system.

1 Introduction

Prolog, like most logic programming languages, uses a top-down execution mechanism. However, some algorithms are more naturally written in a bottom-up style. We are developing a pure temporal logic programming language, Starlog, that supports both bottom-up and top-down execution in a natural way. Developing execution mechanisms that support both efficiently is an interesting challenge.

Two common evaluation techniques for bottom-up systems are naive evaluation and semi-naive evaluation [12]. 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,

```
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),   % Fail if no new facts derived.
    naive_evaluation.       % Iterate.
naive_evaluation.           % End of evaluation.
```

Figure 1: Prolog Interpreter for Naive Evaluation

it naturally allows programmers to mix top-down and bottom-up code, and it supports declarative programming of I/O and change.

2 Naive and Semi-Naive Evaluation

Naive evaluation [12] is a direct implementation of the bottom-up, fixed point semantics of logic programs [9], 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

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

(The optional condition $H\mu \notin S_i$ guarantees that only atoms not derived in previous steps are added at time $i+1$.) An interpreter for naive evaluation can be written easily in Prolog (Figure 1).

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 $j > i$. (Bottom-up computation of `ancestor/2` is an example of a computation that would result in duplication.)

Semi-naive evaluation [12] overcomes both inefficiencies by making sure that at least one newly derived fact is used to satisfy the body of a program clause.

Semi-naive Evaluation

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

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. In addition, the same test 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 management of the delta sets is the main stumbling block — is a major reason for our developing pseudo-naive evaluation.

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 [8, 5], an executable temporal logic [18, 2, 4, 10, 1]. 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. (Section 9 discusses several applications utilizing declarative I/O and change.) The present paper shows, in addition, that time can be used to structure, and possibly simplify the implementation of, traditional (bottom-up) logic 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 are terms that must evaluate to natural numbers. 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 (builtin) 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 [10].

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[6] or Mercury[16, 15], 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 causes 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. As a consequence, a Starlog program is basically data-driven, rather than goal driven. We envisage that in many applications, the data that will drive execution will be events, such as button presses, timer expirations, arrival of messages on a channel, and notification of completion of subtasks. The programmer will certainly need to be careful that the constructed model is not too large — that not too many irrelevant facts are derived. (A similar need for care is present for programmers using forward-chaining expert system languages.) So far, we have envisioned no need to implement magic sets [12], though we have not ruled out that option. Magic sets seem unwieldly to implement, analyze, and debug [1] and may be more appropriate for query-based deductive databases than for a (reactive) programming language like Starlog.

Nonetheless, we have found that it is most natural and efficient to specify that certain relations are to be evaluated top-down. Since we do not support magic sets, having top-down predicates simplifies control for those procedures that do need a goal-driven component. Furthermore, facts that are derived bottom-up are memoized (tabled), as in typical deductive databases, while facts derived top-down are not 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 probably best executed top-down. Lastly, rules that would need arithmetic constraints if executed bottom-up, due to unbound variables, can often be executed top-down without constraints. Consider, for example, a rule like the following from the elevator simulator of Section 9:

```
existsTimeElevatorHasJustStoppedIndicator(Floor,Tm10)@T:-
    elevatorHasJustStoppedIndicator(Floor)@U,
    Tm10< U, U =< T.
```

If executed bottom-up, the variable `T` would be unbound, and so the resulting tuple would involve an arithmetic constraint. But executed top-down, `T` gets bound. This implementation of Starlog[2] does not handle (nonground) arithmetic constraints.

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. Sections 7 and 9 elaborate on this point.

4 Strongly-Stratified Programs

A *strongly stratified* program is one that does not contain any simultaneous recursion through bottom-up predicates: all recursion must occur in top-down relations, or be stratified through time. 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 either: p/n is executed top-down (by Prolog) or T must be strictly greater than T'.

[1] We base this opinion partly on our experience using the the Coral Deductive Database System [11, 17].

[2] A separate implementation does handle constraints.

In other words, within each time step, bottom-up computation is hierarchical (non-recursive). We emphasize that negation too must be strongly stratified.

A more formal definition of *strongly stratified* is that a program is strongly stratified if there exists a partial order $\prec$ over all the bottom-up predicate names (P/N) in the program such that, for every program clause

```
P@T <-- [not] P1@T1, [not] P2@T2, ...  [not] Pn@Tn.
```

(where the `[not]`'s indicate optional negation) it holds that

$$\forall\, i \in \{1, \ldots, n\}\ .\ T_i < T \vee T_i = T \wedge P_i \prec P\ .$$

Note that a strongly stratified programs can still contain recursion, but the recursion is restricted so that all recursive cycles of bottom-up predicate calls involve decreasing time.

Furthermore, the Starlog 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 restiction 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.

Is it reasonable to require all Starlog programs to be strongly stratified? We believe so, because a surprisingly large proportion of bottom-up programs are easy to write in a strongly stratified style. In addition, most programs that are not strongly stratified can be transformed into strongly stratified programs in a fairly straightforward way, or handled by a small extension to the pseudo-naive interpreter, as will be shown in Section 7. Furthermore, we believe that requiring programs to be strongly stratified is helpful from a correctness and software engineering point of view, as it forces the programmer to be explicit about termination and progress conditions of the code. Since bottom-up execution is hierarchical at a given time step, non-termination must result either from recursion through time, from top-down calls, or from infinite data.

A useful analogy is that just as Mercury supports the subset of Prolog programs that are well-moded (with statically identifiable modes and forms of nondeterminism), our pseudo-naive evaluation (efficiently) supports the subset of logic programs that are strongly stratified. Pseudo-naive evaluation itself is optimized to handle bottom-up hierarchical execution at each time step. We use the explicit timestamps or top-down calls to handle simultaneous recursion and to simulate the delta sets of semi-naive evaluation. We thereby obtain a more efficient, more data-driven implementation than naive evaluation. But the explicit timestamps are easier to implement efficiently than full delta-sets.

4.1 Compilation of Strong Stratification

It is tedious for programmers to have to specify the stratification partial order explicitly, 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),
        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] ;
```

and

```
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`[3] 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 predicate names 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 propose to 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 be inefficient since naive evaluation is not data driven. 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

[3] 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.

```
:- 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.
%In the actual implementation, Body is compiled as Prolog.
    \+Head,         % Make sure it's new.
    assert(Head),
    trigger(Head,NewToken).  % drive the data
```

Figure 2: Prolog Interpreter for Pseudo-naive Evaluation

set. Token set τ_i consists of the set of relation names (predicate plus arity) which have received new facts at iteration i. 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.

Pseudo-naive Evaluation

$$
\begin{aligned}
S_0 &= \{H \mid (H \leftarrow \mathsf{true}) \in P\} \quad \text{(facts)} \\
\tau_{i+1} &= \{p/n \mid \exists t_1, \ldots, t_n,\ p(t_1, \ldots, t_n) \in S_{i+1}\} \quad (i \geq -1) \\
S_{i+1} &= S_i \cup \{H\mu \mid (H \leftarrow B) \in \mathit{Rename}(P) \\
&\qquad \wedge (\mathit{preds}(B) \cap \tau_i \neq \emptyset) \wedge (B\mu \subseteq S_i) \wedge \neg\exists j \leq i \ . \ H\mu \in S_j\} \quad (i \geq 0) \\
S &= \cup_{i=0\ldots\infty} S_i
\end{aligned}
$$

In short, pseudo-naive evaluation is data-driven at the level of whole relations, but not at the level of each element of a relation.

An interpreter for pseudo-naive evaluation appears in Figure 2. According to the abstract definition of pseudo-naive evaluation in the previous paragraph, each τ_i (token set) contains a set of predicate names and arities p/n. As a refinement on that abstract definition, the interpreter implements the optimization in which τ_i contains a list of pairs, where each pair consists of a timestamp and the index of a rule to be tried. Section 8 describes several additional optimizations.

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 indices I, for some arguments $T_1, \ldots, T_n$, then the fact `trigger(p(_,...,_),I)` must appear in the compiled program.

The next section elaborates on the compilation of triggers.

We note some similarity between our pseudo-naive interpreter and the techniques described in [19] for converting bottom-up rules into Prolog rules.

6 Compilation for Pseudo-Naive Evaluation

The most important aspect of compiling Starlog 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. 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, $A@U$, such that U is provably the latest time at which any other input tuples for this clause can arrive, 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.

7 Handling Simultaneous Recursion

There are at least three 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.
2. Make the procedure top-down and use tabling (OLDT resolution [14]), as in XSB [13]. This option will work for a larger class of programs (e.g., those 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. Add explicit timestamps to make the recursion strongly stratified, and use pseudo-naive bottom-up evaluation.

The last option, which we explain below, is effectively semi-naive evaluation, using the timestamped relations to hold the delta sets. We will illustrate these three methods via an example.

Consider the following program.

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

Then the following query eventually loops using Prolog evaluation.

```
| ?-  ancestor(john,Who).
```

One could rewrite it to eliminate the left recursion. Alternatively, one used OLDT resolution (say, XSB) with `ancestor/2` declared to be tabled. Then both termination and efficiency would be achieved.

If we tried to use pseudo-naive evaluation with simultaneous recursion, the program would look like

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

The program would terminate correctly, but because `ancestor/2` tuples derived at time `T` immediately enable additional `ancestor/2` tuples to be derived at the same time `T`, pseudo-naive evaluation (like naive evaluation) would be inefficient. The lack of "delta" relations would result in each invocation of the body of the recursive clause at time `T` re-deriving all the tuples derived in earlier iterations at time `T`.

We are currently investigating another option, that preserves termination and efficiency. This is to strongly stratify the bottom-up recursion, by writing the program as follows:

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

ancestor(A,D)@T:- not(ancestor_delta(_,_)@T),ancestor_delta(A,D)@U,U<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`. When a time `T` is reached when no more tuples can be derived, the union of the `ancestor_delta(A,D)@I` tuples for I$=0,\ldots,$T-1 is the entire `ancestor/2` relation; hence, the last clause.

Now, for the above 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`. For example, suppose the solutions to `ancestor/2` are needed at some time `T` using the then current values of the `parent/2` relation. We propose using the module system to isolate a local computation in its own 'timespace.' First, we place the code for `ancestor/2` in its own module, with an interface definition that specifies `parent/2` as input and `ancestor/2` as output. One then invokes the module with a rule like

```
ancestor:parent(P,C)@T:- parent(P,C)@T,.....
```

This rule causes the `parent/2` facts of time `T` to be bound as the `parent/2` input tuples of the `ancestor/2` procedure. The solutions are placed, at the calling module's time `T`, in a relation accessible in rule bodies by an atom like `ancestor:ancestor(A,D)@T`.

8 Other Features and Optimizations

Starlog implements a declarative form of state variable based on the `from ... until` operator of temporal logic. As in the event calculus [7], one specifies the initiating (terminating) conditions that cause a state variable to become true (false). For example, our elevator simulator contains the following rules

```
initiates(doorIsOpeningState)@T :-  doorHasStartedOpeningIndicator@T.
terminates(doorIsOpeningState)@T :- doorHasFinishedOpeningIndicator@T.
```

which say that when the elevator controller receives the specified indicator signal from the sensors, the `doorIsOpeningState` variable becomes true, and it remains true until the specified other indicator signal is received. The initiates/terminates idiom simplifies the coding of rules (e.g., frame axioms) involving persistent relations that would otherwise have to be written clumsily (and inefficiently) like

```
doorIsOpeningState@T+1:- doorHasStartedOpeningIndicator@T.
doorIsOpeningState@T+1:- doorIsOpeningState@T,not(doorHasFinishedOpeningIndicator@T).
```

Our system incorporates some optimizations that have not been described above. First, if every call to a nullary predicate $P/0$ is in a trigger position, it does not assert $P/0$ tuples at all, since the token that is inserted into the priority queue carries all the information that is needed.

In addition, the `enables/2` predicate is implemented more efficiently than shown in Fig. 2, where rule bodies are treated as data. 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:[4]

```
    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. These factors are some of the major reasons for achieving good speed and leverage from the underlying Prolog.

[4] In practice, all $P@T$ atoms are transformed so that the time argument T becomes an additional (first) argument of P. This improves indexing performance when atoms are looked up.

Our current implementation uses the dynamic database (`assert/1`) to store relations. This has the advantages of simplicity, transparency, and portability. The bodies of both bottom-up and top-down clauses can be compiled and executed as normal Prolog rules, as discussed in the previous paragraph. Since our implementation is written in 'vanilla' Prolog, it is easy to port it to different Prolog implementations. Under XSB Prolog, we take advantage of user-specifiable indexing to specify additional indexing, where appropriate. In a language like Mercury that does not have `assert`, we could instead use data structures from a library package to store the relations.

Currently insertion of tokens into the priority queue (procedure `merge_tokens` of Fig. 2) is done in Prolog. Since tokens consist of pairs of integers (time plus rule index), management of the queue could easily be done instead by a foreign language procedure written in, say, C.

Garbage collection is important for Starlog since in the abstract bottom-up semantics, nothing is ever forgotten. The programmer can add declarations specifying when bottom-up tuples may be safely garbage collected. For example, the declaration

```
:- persists(elevatorHasJustStoppedIndicator/1,12).
```

says that tuples of the indicated bottom-up relation can be garbage collected after 12 time steps. We have yet to address automatic compile-time or runtime determination of when data can be garbaged collected. We suspect though, that some version of our current scheme will be necessary for complex relations.

9 Applications

We briefly describe three medium-size applications that run successfully in Starlog.

Elevator simulator This program implements a fully declarative simulator for a single elevator, with graphical animation and user input. For a specification of the simulator, see `<http://www.cs.waikato.ac.nz/~dsmith/CHALLENGES/details.html>`.

The animation shows, on the left, the position of the elevator. In the middle it shows unserviced call requests at each floor ("up" or "down"). On the right it shows (with a "+" sign) unserviced floor requests that result from passengers selecting a floor to go to.

There are, in effect, six processes that run concurrently:

1. The elevator controller itself (generates control signals, based on values of indicator signals, time delays, and global state variables).
2. The door motor (reacts to door control signals, generates door indicator signals)
3. The main motor (reacts to main motor control signals, generates main motor indicator signals)
4. Elevator-motion-detectors (generates indicator signals)
5. The animation process.
6. The passenger simulator.

There are two versions of the last process. One version randomly generates button presses, thereby simulating passengers. The second version allows asynchronous, external input of button presses by mouse. Currently, all animation is done with ASCII curses-style character graphics.

For reasonable parameters, our pseudo-naive interpreter executes the elevator simulation approximately four times faster than an earlier interpreter that used naive evaluation. Exactly how much faster it runs depends on the density of events: the longer the average interval between button presses, the more benefit accrues from data-driven execution.

graphplan [3] robot planner Space limitations preclude a full description of this application. The planning graph and its add, delete, and mutual exclusion edges are all represented relationally. Using relations instead of data structures has the advantage that data is more accessible. Performance depends critically on the underlying Prolog's ability to do indexing. On some examples graphplan ran up to five times faster using XSB's flexible indexing declarations as compared to using XSB without extra indexing. The backward phase of the graphplan algorithm is executed top down, by virtue of a `top_down` declaration. It is easy to add heuristics; on some examples, we could thereby significantly outperform the original C-based implementation of graphplan.

Ecology simulation . This application is an individual-based ecology simulation with ASCII graphics animation. Each specie has its own subwindow showing population sizes at each location. For each specie one specifies its predators, preys, death rate (depending on hunger and age), initial population size, and moving speed. The code requires aggregates, randomness, and indeterministic choice. Using several hundred individuals, each 'generation' takes about a second running on a SPARCserver-1000.

Of these three applications, only the first benefits significantly from pseudo-naive evaluation: the last two would run about as efficiently (but not significantly more efficiently) using naive evaluation, since nearly every rule fires during every time step.[5] Indeed, naive evaluation is quite adequate for many procedures. This is symptomatic of a general phenomenon in high-level languages: the high-level features (e.g., unification, backtracking, delta sets, data-driven execution) are used in restricted contexts and in restricted ways. A well-designed language optimizes the common patterns of use. We think that in a typical application, the outermost 'event loop' will be bottom-up and data-driven (i.e., reactive), while much of the algorithmic logic of the code will be top-down, using Prolog data structures, backtracking, and unification, as needed.

10 Conclusion

We have presented a novel implementation technique, pseudo-naive evaluation, that combines the advantages of bottom-up and top-down logic programming. The essential idea is to prohibit simultaneous bottom-up recursion by requiring all recursion to be either top-down or mediated by a temporal stratification. Pseudo-naive evaluation optimizes execution of such programs by introducing a data-driven component to naive evaluation, without maintaining the explicit 'delta sets' of semi-naive evaluation. Pseudo-naive evaluation combines the simplicity of naive evaluation with much of the data-driven reactivity of semi-naive evaluation. By construction, pseudo-naive evaluation will be as efficient as semi-naive evaluation on strongly-stratified programs. A reactive, data-driven program will benefit significantly from pseudo-naive evaluation.

Areas of future work include

- automatic compilation of complex triggers
- garbage collection (automatic derivation of `:- persists` declarations)
- program analysis to support the above two
- modular time, using nested timestamps or local timespaces.

[5] In the language of Section 6, the rules have 'every_time' triggers.

Acknowledgements

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

References

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

[2] M. Baudinet, J. Chomicki, and P. Wolper. Temporal deductive databases. In Tansel et al. [18].

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

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

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

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

[7] Robert Kowalski and Fariba Sadri. The situation calculus and event calculus compared. In Maurice Bruynooghe, editor, *Logic Programming - Proceedings of the 1994 International Symposium*, pages 539–553, Massachusetts Institute of Technology, 1994. The MIT Press.

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

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

[10] Mehmet A. Orgun and Wanli Ma. An overview of temporal and modal logic programming. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic: First International Conference*. Springer-Verlag, LNCS 827, 1994.

[11] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. CORAL: Control, Relations and Logic. In *Proceedings of the International Conference on Very Large Databases*, 1992.

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

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

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

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

[16] Z. Somogyi, F. J. Henderson, and T. C. Conway. Code generation for Mercury. In J. Lloyd, editor, *Logic Programming, Proceedings of the 1995 International Symposium*, Portland, Oregon, USA, 1995. MIT Press.

[17] Divesh Srivastava, Raghu Ramakrishnan, S. Sudarshan, and Praveen Seshadri. Coral++: Adding object-orientation to a logic database language. In *Proceedings of the International Conference on Very Large Databases*, 1993.

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

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