Online Justification for Tabled Logic Programs*

Giridhar Pemmasani[1], Hai-Feng Guo[2], Yifei Dong[3], C.R. Ramakrishnan[1], and I.V. Ramakrishnan[1]

[1] Department of Computer Science, SUNY, Stony Brook, NY, 11794, U.S.A.
E-mail: {giri,cram,ram}@cs.sunysb.edu
[2] Department of Computer Science, University of Nebraska at Omaha
Omaha, NE, 68182, USA, E-mail: haifengguo@mail.unomaha.edu
[3] School of Computer Science, University of Oklahoma
Norman, OK 73019, USA, E-mail: dong@ou.edu

Abstract. Justification is the process of constructing evidence, in terms of proof, for the truth or falsity of an answer derived by tabled evaluation. The evidence is most easily constructed by post-processing the memo tables created during query evaluation. In this paper we introduce *online justification*, based on program transformation, to efficiently construct the evidence *during* query evaluation, while adding little overhead to the evaluation itself. Apart from its efficiency, online justification separates evidence generation from exploration thereby providing flexibility in exploring the evidence either declaratively or procedurally. We present experimental results obtained on examples that construct large evidences which demonstrate the scalability of online justification.

1 Introduction

Explaining or understanding the results of logic program evaluation, to aid in debugging or analyzing the program, has been a very active field of research. The complex machinery used to evaluate tabled logic programs makes the generation of explanations for query evaluation considerably difficult. Justification [17, 9] is the process of generating evidence, in terms of a high level proof, based on the answer tables created during query evaluation. Justification has two important advantages over techniques which generate evidence based on execution traces, viz. (i) it is independent of the evaluation machinery (e.g. the techniques scheduling goals in a tabled evaluation), and (ii) it enables direct construction of "meta-evidences": evidences for proof systems implemented as logic programs. Justification has played a fundamental role in generating proofs or counter examples for several problems in automatic verification (e.g., see [15, 16, 1]).

In earlier works [17, 9], we presented justification algorithms for logic programs by post-processing the memo tables created during query evaluation. Justification in this post-processing fashion is "non-intrusive" in the sense that it is completely decoupled from query evaluation process and is done only after

* This research was supported in part by NSF grants CCR-9876242, IIS-0072927, CCR-0205376, CCR-0311512, and ONR grant N000140110967.

the evaluation is completed. However, post-processing introduces performance overheads which affect the scalability of the technique. In this paper we present an *online* technique for justification, which generates evidence, whenever possible, *during* query evaluation itself. Online justification is presented as a program transformation, and hence is still independent of the query evaluation machinery.

At a high level, our online justification technique transforms the given program in such a way that the query evaluation in the transformed program automatically constructs the evidence. For each literal in the program, we create two transformed literals: one to generate the evidence for its truth, and the other for its falsity. Evidence for derivability (truth) of an answer can be easily constructed by adding an extra argument in the predicate definitions that captures the evidence. However, in the presence of tabled evaluation, the extra evidence argument causes serious performance problems since the number of proofs for a goal may far exceed the number of steps it takes to compute the goal. For instance, consider the problem of a parser for an ambiguous grammar[21]: determining whether there is a parse tree can be done in time cubic on the length of the string whereas there may be exponentially many parse trees. We use a different transformation scheme for tabled predicates, storing the *first* evidence for an answer in a database, and thereby avoid this blow-up (see Section 3).

Generating evidences for false literals is more difficult, since evaluation of false literal simply fails without providing any information regarding the failure. We generate evidences for false literals by first constructing a *dual* program (based the notion of completed definition [11]) such that of a literal in the dual program is true if and only if its corresponding literal in the original program is false. Thus, evidences for false literals are constructed using the evidences for the corresponding (true) literals in the dual program (see Section 4).

Related Work: Extensive research has been done to help debug, analyze, explain or understand logic programs. The techniques that have been developed can be partitioned into three (sometimes overlapping) stages: instrumenting and executing the program, collecting data, and analyzing the collected data. Most of the techniques focus primarily on one of these three stages. For instance, the works on algorithmic debugging [18], declarative debugging [10, 13], and assertion based debugging [14] can be seen as primarily focussing on instrumenting the program; works on explanation techniques (e.g.,[12, 4, 19]) focus primarily on data collection; and works on visualization (e.g. [4, 2]) focus on the data analysis stage. Justification focusses on the data collection stage.

Unlike algorithmic debugging, justification only shows those parts of the computation which led to the success/failure of the query, and unlike declarative debugging, justification does not demand any creative input from the user regarding the intended model of the program, which can be very hard or even impossible to do as will be the case in model checking [3]. Among explanation techniques, [19] proposes a source-to-source transformation technique, which is very similar to our technique, to transform logic programs in the context of deductive databases. This technique generates evidence in bottom-up evaluation and is limited non-tabled programs, making it expensive especially in the pres-

ence of redundant computations. A later work [12] generates explanations by meta-interpreting intermediate information (traces) computed by the database engine, and overcomes the problems due to redundancy by caching the results. However, the explicit cycle checking done when generating explanations imposes a quadratic time overhead for evidence generation.

Trace-based debuggers (of which Prolog's 4-port debugger is a primitive example) provide only a procedural view of query evaluation. Procedural view provides information about the proof search, rather than the proof itself. The complex evaluation techniques used in tabling make this procedural view virtually unusable. Moreover, in the presence of multiple evaluation strategies and engines (SLG-WAM [20], Local vs. Batched scheduling [7], DRA [8] etc.), there is no uniform method to convert a trace of events during a proof search into a view of the evidence (proof, or its lack thereof) itself. Finally, by delinking evidence generation from evidence navigation, justification enables a user to selectively explore parts of the evidence and even re-inspect an explored part without restarting the debugging process.

Online Justification vs. Post-Processing: We originally presented a technique for justification of tabled logic programs [17] and later refined the technique to efficiently handle programs that mixed the evaluation of tabled with nontabled goals [9]. However, both the techniques post-processed the memo tables to build evidences. To understand the two main drawbacks of these techniques, consider the evaluation of query `p` over the tabled logic program given below:

```
:- table p/0.
p :- p.        p :- q.        q.
```

Post-processing-based justification is done by meta-interpreting the clauses of the program and the memo tables built during evaluation. For instance, the evidence for the truth of `p` is constructed by selecting a clause defining `p` such that the definition is true. Note that, in this case, the right hand sides of both clauses of `p` are true. For `p` to be true in the least model, its truth cannot be derived from itself. Hence the first clause is not an explanation for the truth of `p`. The meta-interpreter keeps a history of goals visited as it searches for an explanation and rejects any goal that will lead to a cyclic explanation. It will hence reject the first clause. Further, the justifier will produce the explanation that `p` is true due to `q`, which in turn is true since it is a fact.

First of all, note that justification appears to perform the same kind of search that the initial query evaluation did in the first place to determine the answers. Worse still, meta-interpretation is considerably slower than the original query evaluation. Secondly, determining whether a goal has been seen before is exactly what a tabling engine does well, and one which is tricky to do efficiently in a meta-interpreter. The justifiers we had built earlier keep the history of goals as a list (to make backtracking over the history inexpensive), and hence cycle-checking makes the justifier take time quadratic in the size of the evidence.

In contrast, online justification generates evidence by evaluating a transformed program directly, exploiting the tabling engine's (optimized) techniques for cycle detection, and eliminating the meta-interpretation overheads.

Justification in Practice: We present the justification technique initially for pure logic programs with a mixture of tabled and nontabled predicates and stratified negation. The technique can be readily extended to handle programs with builtin or nonlogical predicates (e.g. `var`) and aggregation (e.g. `findall`). Programs involving database updates (e.g. `assert`) and tabling are very uncommon; nevertheless our technique can be extended to such programs as well (see Section 5).

We have implemented an online justifier in the XSB logic programming system [22] and used it to build evidences in a complex application: the XMC model checking environment [15]. We use the XMC system as the primary example, since (i) evidences generated for XMC are large and have different structures based on the system and property being verified, thereby forming a platform to easily validate the scalability and understand the characteristics of the evidence generation technique; and (ii) counter-example generation was added to XMC using justification without modifying XMC itself, thereby demonstrating the flexibility offered by justification. Preliminary performance evaluation indicates that the online justifier for the XMC system adds very little overhead to the XMC model checker (see Section 6). When the model checker deems that a property is true, the overhead for justification to collect that proof is less than 8%. When a property is false, generating the evidence for the absence of a proof the overhead is at most 50%. In contrast, the post-processing based justifier originally implemented in the XMC system had an overhead of 4 to 10 *times* the original evaluation time [9], regardless of the result of the model checking run.

2 Evidence for Tabled Logic Programs

In this section, we give the intuition and formal definition of evidence in the context of tabled logic programs with left-to-right selection rule. By evidence, we mean the data in a proof, i.e., the subgoals and derivation relation between the subgoals. Proofs for (non-tabled) logic programs are traditionally represented by trees, or so-called "proof trees", where a goal is recursively decomposed to a set of subgoals until the subgoal indicates the presence or absence of a fact. In the case of tabled logic programs, however, a proof is not necessarily a tree because of the fixed-point semantics of tabling, i.e., a tabled predicate may fail due to circular reasoning. In [17, 9], proof trees are augmented by "ancestor" nodes or edges to form justifications for tabled programs, essentially indicating that the derivation relation of the subgoals is potentially a graph with loops.

Formally, we define an evidence (for a tabled logic program) as a graph whose vertices are literals and their truth values, and edges reflect the derivation relation between the subgoals represented by the literals. We use $succ(v)$ to denote the set of successors of a vertex v in a graph.

Definition 1 (Evidence). *An evidence for a literal L being true (false) with respect to a program $\mathcal{P}$, denoted by $\mathcal{E}_P(L)$, is a directed graph $\langle V, E\rangle$ such that:*

1. *Each vertex $v \in V$ is uniquely labeled by a literal of $\mathcal{P}$ and a truth value, denoted by $(l(v), \tau(v))$;*

2. *There exists a vertex* $v_0 \in V$ *such that* $l(v_0) = L$, $\tau(v_0) = true(false)$, *and all other vertices in* V *are reachable from* v_0;
3. *For each vertex* $v \in V$
   (a) *if* $\tau(v) = true$, $succ(v) = \{v'_1, \ldots, v'_n\}$ *if and only if*
      i. $\exists\, C \equiv (\alpha :- \beta_1, \ldots, \beta_n) \in P$ *and a substitution* θ :
         $\alpha\theta = l(v) \wedge (\beta_1, \ldots, \beta_n)\theta = (l(v'_1), \ldots, l(v'_n))$.
      ii. $\forall 1 \leq i < n : \tau(v'_i) = true$
      iii. $(v, v) \notin E^+$
   (b) *if* $\tau(v) = false$, $succ(v)$ *is the smallest set such that*
      $\forall\, C \equiv (\alpha :- \beta_1, \ldots, \beta_n) \in P$:
      $\forall$ *substitution* $\theta : \alpha\theta = l(v)\theta \implies$
      $\exists\, 1 \leq k \leq n$ *and* $\{v'_1, \ldots, v'_k\} \subseteq succ(v)$:
      $(\beta_1, \ldots, \beta_k)\theta = (l(v'_1), \ldots, l(v'_k))\theta$
      $\wedge (\forall\, 1 \leq i < k : \tau(v'_i) = true)$
      $\wedge\, \tau(v'_k) = false$

Intuitively an evidence carries only the relevant information to establish a literal's truth or falsity. It is an AND-graph where each node is supported by all its successors. For a true literal, only one explanation matching a predicate definition is needed (Item 3.a.i); for a false literal, it must be shown that every possible combination of its predicate definition ultimately fails (Item 3.a.i). Furthermore only false literals can be in a loop, due to the least fixed-point semantics obeyed by tabled logic programs (Item 3.a.ii).

Definition 1 is logically equivalent to the definitions of justification in [17, 9] which define a spanning tree of evidence, where the backward edges are labeled by "ancestor" and leaves with truth value *true* and *false* are labeled by an edge to node "fact" and "fail" respectively. The benefit of the new definition is that same result will be generated from different traversal orders. Applying the result of [17, 9], we establish the usefulness of evidence by the following theorem.

Theorem 1 (Soundness and Completeness). *The query of a literal* l *succeeds (fails) if and only if there is an evidence for* l *being true (false).*

Hereafter when P is obvious from the context, we abbreviate $\mathcal{E}_P(L)$ to $\mathcal{E}(L)$. Sometimes we also abbreviate v's label $(l(v), \tau(v))$ to $l(v)$ or $\neg l(v)$ depending on whether $\tau(v)$ is *true* or *false*.

Example 1. Consider the following three programs:

P_1:
```
p.
```
P_2:
```
p :- q.
q.
```
P_3:
```
:- table p/0.
p :- q.
q :- p.
```

The evidence for p being *true* in P_1 is just a single node labeled by p. The node has no successor because it is fact hence does not need further explanation. The evidence for p being *true* in P_2 contains two nodes labeled by p and q respectively and an edge from p to q, meaning that p is *true* because q is *true*.

Program P_3 encodes a circular reasoning, therefore p is *false*. The evidence of p being *false* is a loop from p to q then back to p. Formally,

$$\mathcal{E}_{P_1}(p) = \langle\{(p, true)\}, \emptyset\rangle$$
$$\mathcal{E}_{P_2}(p) = \langle\{(p, true), (q, true)\}, \{(p \rightarrow q)\}\rangle$$
$$\mathcal{E}_{P_3}(\neg p) = \langle\{(p, false), (q, false)\}, \{(\neg p \rightarrow \neg q), (\neg q \rightarrow \neg p)\}\rangle$$

Example 2. In the following logic program, the predicate $reach(A, B)$ defines the reachability from node A to node B, and $arc(From, To)$ encodes the edge relation of a graph.

```
:- table reach/2.                              arc(a,b).
reach(A,B) :- arc(A,B).                        arc(b,a).
reach(A,B) :- arc(A,C), reach(C,B).            arc(c,a).
```

The evidences for $reach(a, a)$ being *true* and $reach(a, c)$ being *false* are depicted in Figure 1.

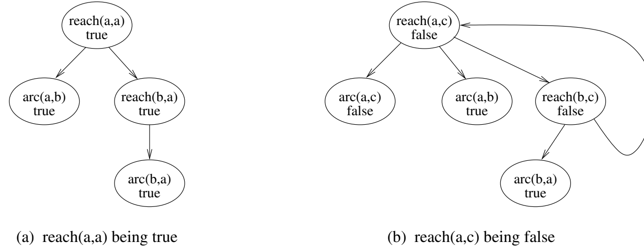


Fig. 1. Two evidence examples in reachability program

3 Evidence Generation for True Literals

The key idea of online evidence generation is to generate the evidence for a literal while the literal is being evaluated. A simple way to implement this idea is to extend each clause

$$\alpha \texttt{ :- } \beta_1, \ldots, \beta_m.$$

to

$$\alpha' \texttt{ :- } \beta'_1, \ldots, \beta'_m, \texttt{store_evid}(\alpha, [\beta_1, \ldots, \beta_m]). \quad (1)$$

where α' and β'_i are same as α and β_i, respectively, but indicate transformed predicates. When the query to a literal $L = \alpha\theta$ succeeds, the successors of L in the evidence, $[\beta_1\theta, \ldots, \beta_m\theta]$, are recorded by `store_evid`. Note that `store_evid` simply records a successful application of a clause and hence does not change the meaning of the program.

Unfortunately, this simple technique may generate more information than necessary for the purpose of evidence. Recall that an evidence carries only the relevant information to establish a literal's truth or falsity, therefore a backtracked call should not be part of the explanation for the final true answer. But the above transformation stores evidence for calls to α that are backtracked in a higher-level call.

Evidence as Explicit Arguments. We solve the above problem by passing the evidence as an argument of the predicates. We add an argument `E` to each atom α to form a new atom $\alpha'_{\mathtt{E}}$ which returns the evidence for α in `E`. Suppose $\alpha = p(t_1, \ldots, t_n)$, then $\alpha'_{\mathtt{E}} = p'(t_1, \ldots, t_n, \mathtt{E})$, where p' is a new name uniquely selected for p . The clause

$$p(t_1, \ldots, t_n) \texttt{ :- } \beta_1, \ldots, \beta_m.$$

is then transformed to

$$p'(t_1, \ldots, t_n, \mathtt{E}) \texttt{ :- } \beta'_{1\mathtt{E}_1}, \ldots, \beta'_{m\mathtt{E}_\mathtt{m}}, \mathtt{E} = [(\beta_1, \mathtt{E}_1), \ldots, (\beta_m, \mathtt{E}_\mathtt{m})]. \quad (2)$$

where $\mathtt{E}, \mathtt{E}_1, \ldots, \mathtt{E}_\mathtt{m}$ are distinct new variables. The last statement in the clause combines the evidence for the subgoals to form the evidence for the top-level query, thus effectively building a tree. Similar to the first transformation, the new predicate $p'/(n+1)$ executes the same trace as the original predicate p/n. Since the evidence is now returned as an argument, backtracked predicates no longer generate unnecessary information.

Evidence for Tabled Predicates as Implicit Arguments. The situation becomes complicated when tabled predicates are present. Because all successful calls to tabled predicates are stored in tables, the space is not reclaimed when the predicates are backtracked. Furthermore, additional arguments for tabled predicates can increase the table space explosively [21]. Therefore there is no saving in passing evidence as an argument in tabled predicates.

To avoid the overhead, we do not pass evidence as arguments in tabled predicates. Instead, we use the `store_evid` method in Clause (1) to store a *segment* of evidence in database, where the evidence for a tabled subgoal β_i is recorded as $\mathit{ref}(\beta_i)$ pointing to β_i's record in the database, and the evidence for a tabled subgoal is the same as in Clause (2).

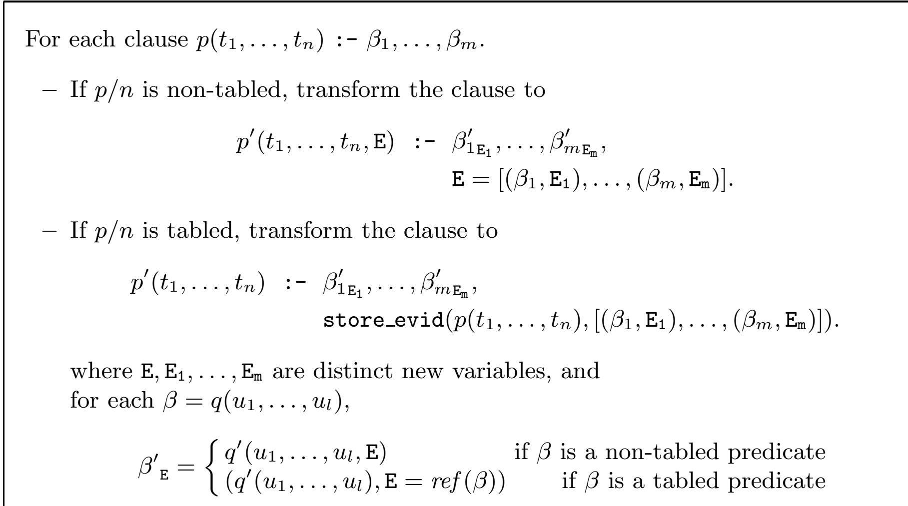
The complete transformation rule for true literals is shown in Figure 2.

Note that `store_evid` always succeeds, therefore adding it to the original predicate does not change the program's semantics. And by induction on the size of evidence, we have:

Proposition 1. *Let* `E` *be a fresh variable. For any literal* $L \in P$

- *the query* $L'_{\mathtt{E}}$ *succeeds if and only if* L *succeeds*
- *if the query* $L'_{\mathtt{E}}$ *succeeds, then* `E` *returns an evidence for* L *being true.*

Example 3. The program in Example 2 is transformed as follows.


For each clause $p(t_1, \ldots, t_n)$:- $\beta_1, \ldots, \beta_m$.

– If p/n is non-tabled, transform the clause to

$$p'(t_1, \ldots, t_n, \mathtt{E}) \ \texttt{:-}\ \beta'_{1\mathtt{E_1}}, \ldots, \beta'_{m\mathtt{E_m}}, \quad \mathtt{E} = [(\beta_1, \mathtt{E_1}), \ldots, (\beta_m, \mathtt{E_m})].$$

– If p/n is tabled, transform the clause to

$$p'(t_1, \ldots, t_n) \ \texttt{:-}\ \beta'_{1\mathtt{E_1}}, \ldots, \beta'_{m\mathtt{E_m}}, \quad \mathtt{store_evid}(p(t_1, \ldots, t_n), [(\beta_1, \mathtt{E_1}), \ldots, (\beta_m, \mathtt{E_m})]).$$

where $\mathtt{E}, \mathtt{E_1}, \ldots, \mathtt{E_m}$ are distinct new variables, and
for each $\beta = q(u_1, \ldots, u_l)$,

$$\beta'_{\mathtt{E}} = \begin{cases} q'(u_1, \ldots, u_l, \mathtt{E}) & \text{if } \beta \text{ is a non-tabled predicate} \\ (q'(u_1, \ldots, u_l), \mathtt{E} = \mathit{ref}(\beta)) & \text{if } \beta \text{ is a tabled predicate} \end{cases}$$

Fig. 2. Transformation Rules for True Literals

```
:- table reach_t/2.
reach_t(A,B) :- arc_t(A,B,E),
                store_evid(reach(A,B), [((arc(A,B),true),E)]).
reach_t(A,B) :- arc_t(A,C,E1), reach_t(C,B),
                store_evid(reach(A,B),
                           [((arc(A,C),true),E1),
                            ((reach(C,B),true),ref(reach(C,B)))]).
arc_t(a,b,[]).           arc_t(b,a,[]).            arc_t(c,a,[]).
```

The query `reach_t(a,a)` will succeed with the evidence stored in two records:

$$reach(a, a) \rightarrow [((arc(a, b), true), []), ((reach(b, a), true), ref(reach(b, a)))]$$
$$reach(b, a) \rightarrow [((arc(b, a), true), [])]$$

4 Evidence Generation for False Literals

Evidence generation for false literals is more difficult than that of true literals, because when the extended predicates fail, they do not return any tracing information. We solve this problem by justifying the *negation* of false literals. We present the solution in two steps. In the first step, for each literal L, we compute a dual literal $\overline{L}$ which is equivalent to $\neg L$. In the second step, we apply the transformation rule for the true literals to $\overline{L}$.

Dual Predicates. Let p/n be a predicate, where p is the predicate name and n is the arity of p. We say that the predicate $\overline{p}/n$ is the *dual predicate* of p if $\overline{p}$ and p return complementary answers for the same input, *i.e.* for any literal instance $p(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms, $\overline{p}(t_1, \ldots, t_n) = \neg p(t_1, \ldots, t_n)$.

Recall from the definition of evidence (Definition 1) that a literal L is false if for every clause $\alpha : -\beta_1, \ldots, \beta_n$ such that L unifies with α, under any substitution θ, there is some $j \leq n$ such that $\beta_l\theta$ is false and for all $1 \leq i < j$, $\beta_i\theta$ is true. This directly leads to the following definition of dual predicates. For the sake of simplicity, let p be defined using k clauses in the form of $p(t_i)$ `:-` $\beta_{i,1}, \beta_{i,2}$. Then,

$$\overline{p}(x) \texttt{ :- } \overline{p_1}(x), ..., \overline{p_k}(x)$$

where $\overline{p_i}$ captures the failure of the i-th clause of p. Now each i-th clause fails if either the arguments of p do not match with the arguments of p_i, or $\beta_{i,1}$ fails, or for every answer of $\beta_{i,1}$, $\beta_{i,2}$ fails. This is captured by the following rule:

$$\overline{p_i}(x) \texttt{ :- } x \neq t_i$$
$$\overline{p_i}(t_i) \texttt{ :- } \overline{\beta_{i,1}} \;;\; \texttt{forall}(\beta_{i,1}, \overline{\beta_{i,2}})$$

where the predicate $\texttt{forall}(\beta_1, \beta_2)$ encodes $\forall\theta : \beta_1\theta \implies \beta_2\theta$.

Dual Definitions for Tabled Predicates. For a tabled predicate involving recursive definitions, however, the dual predicates defined in the above scheme is incorrect in general. We can view the above definition of the dual as being obtained from the completion of a logic program [11]. It is well known that completion has unsatisfactory semantics for negation. This impacts our justification algorithm. Consider the simple propositional program `p :- p`, where $p/0$ is tabled. The dual predicate produced by the above transformation rule is `p_f :- p_f`. Because there is a loop in the definition of `p_f`, if `p_f` is not tabled, the query does not terminate; if it is tabled, it will give the answer *false*. However, since `p` fails, `p_f` should succeed.

To break the loops in dual predicates, we use the explicit negation of the tabled predicates instead of their duals in the definitions. In XSB, that a tabled predicate α has no answer, *i.e.* $\neg\exists\theta : \alpha\theta$, is encoded by `sk_not(`$\alpha$`)`. In other tabled LP systems such as TALS and B-Prolog, the operator for tabled negation is the same as for non-tabled cases. Here we use `table_not` to represent this negation operator for all tabled systems. For the above example, we replace `p_f` in the body of of dual for `p` by `table_not(p)`. Now the dual predicate becomes `p_f :- table_not(p)`, so the query `p_f` correctly succeeds.

The benefit of using `table_not` for tabled predicate in the dual definitions is that there are no recursive calls to the duals of tabled predicates, hence the duals need not be tabled. This not only avoids cycle checking but also enables us to implement the justifier as a zero-storage producer, as the evidence does not consume permanent space when being passed as an argument.

Generating Evidence for the Dual Predicates. We apply the transformation rule for true literals presented in Section 3 to the dual predicates, so that for each predicate α, we generate an extended dual predicate $\overline{\alpha}'_{\texttt{E}}$ that returns the evidence for $\neg\alpha$ in the variable `E`.

The two predicates introduced during dualization, `forall` and `table_not`, need special treatment in the transformation.

- We implement a predicate $\texttt{all_evid}((\beta'_{1_{\texttt{E}_1}}, \texttt{E}_1), (\beta'_{2_{\texttt{E}_2}}, \texttt{E}_2), \texttt{E})$ to computes the evidence of $\texttt{forall}(\beta_1, \beta_2)$:

$$\mathcal{E}(\texttt{forall}(\beta_1, \beta_2)) = \bigcup_{\forall \theta : \beta_1 \theta} \{(\beta_1\theta, \mathcal{E}(\beta_1\theta)), (\beta_2\theta, \mathcal{E}(\beta_2\theta))\}$$

  where $\texttt{E}_1$, $\texttt{E}_2$, and $\texttt{E}$ hold the evidences of $\beta_1\theta$, $\beta_2\theta$, and $\texttt{forall}(\beta_1, \beta_2)$ respectively.
- To extend the predicate $\texttt{table_not}(\beta)$, we apply the same technique used for true tabled predicates, *i.e.* to store only a pointer to the evidence for $\neg\beta$, denoted by $ref(\beta)$. Similar to the evidences for true literals, the evidences for false tabled literals are stored in segments. The evidence segment for $\neg\beta$ can be generated by calling $\overline{\beta}'_{\texttt{E}}$, which can be done at any time, giving us an algorithm of generating partial evidence *on demand*. The evidence is fully generated when the evidence segments to all referred literals are computed.

The complete transformation rule for false literals is in Figure 3. A special case of this transformation is that when a predicate p/n has no definition (thus p trivially fails), a fact $\overline{p}'(\texttt{X}_1, \ldots, \texttt{X}_n, [])$. is generated according to Step 1, meaning $\overline{p}'$ trivially succeeds.

Based on the definition of dual predicates and Proposition 1, we can establish the consistency of the transformed predicate. Also by induction on the size of evidence, we have:

Proposition 2. *Let* E *be a fresh variable. For any literal $L \in P$,*

- *the query $\overline{L}'_{\texttt{E}}$ succeeds if and only if L fails*
- *if the query $\overline{L}'_{\texttt{E}}$ succeeds then* E *returns an evidence for L being false.*

Example 4. The program in Example 2 is transformed as follows.

```
reach_f(A,B,E) :- reach_f1(A,B,E1),reach_f2(A,B,E2),concat([E1,E2],E).
reach_f1(A,B,E) :- arc_f(A,B,E).
reach_f2(A,B,E) :- arc_f(A,C,E).
reach_f2(A,B,E) :- all_evid((arc_t(A,C,E1),E1),
                            (reach_f(C,B),E2=ref(reach(C,B))), E).
arc_f(A,B,E)   :- arc_f1(A,B,E1), arc_f2(A,B,E2), arc_f3(A,B,E3),
                  concat([E1,E2,E3],E).
arc_f1(A,B,[]) :- (A,B) \= (a,b).
arc_f2(A,B,[]) :- (A,B) \= (b,a).
arc_f3(A,B,[]) :- (A,B) \= (c,a).
```

The query `reach_f(a,c,E)` will succeed, returning one evidence segment:

$$reach(a, c) \rightarrow [((arc(a, c), false), []), ((arc(a, b), true), []), \\ ((reach(b, c), false), ref(reach(b, c)))]$$

To produce the evidence segment for $reach(b, c)$, we call `reach_f(b,c,E)`, which returns

$$reach(b, c) \rightarrow [((arc(b, a), true), []), ((reach(a, c), false), ref(reach(a, c)))]$$

Now since all referred literals have been visited, the evidence is fully generated.

For each predicate p/n whose definition is composed of k clauses in the form of

$$p(t_{i,1}, \ldots, t_{i,n}) \texttt{ :- } \beta_{i,1}, \ldots, \beta_{i,m_i}.$$

where $1 \leq i \leq k$, the extended dual predicate $\overline{p}'$ is defined in two parts:

1. The top-level predicate is

$$\overline{p}'(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{E}) \texttt{ :- } \overline{p}'_1(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{E_1}), \ldots, \overline{p}'_k(\mathtt{X_1}, \ldots, \mathtt{X_n}, \mathtt{E_k}), \\ \mathtt{concat}([\mathtt{E_1}, \ldots, \mathtt{E_k}], \mathtt{E}).$$

   where $\mathtt{concat}([\mathtt{E_1}, \ldots, \mathtt{E_k}], \mathtt{E})$ is a predicate that concatenates $\mathtt{E_1}, \ldots, \mathtt{E_k}$ together to a single list $\mathtt{E}$.
2. For each $1 \leq i \leq k$, the predicate $\overline{p}'_i/(n+1)$ is defined by two clauses:

$$\overline{p}'_i(\mathtt{X_1}, \ldots, \mathtt{X_n}, []) \texttt{ :- } \mathtt{not}((\mathtt{X_1}, \ldots, \mathtt{X_n}) = (t_{i1}, \ldots, t_{in})). \\ \overline{p}'_i(t_{i,1}, \ldots, t_{i,n}, \mathtt{E}) \texttt{ :- } \mathit{fevid}([\beta_{i,1}, \ldots, \beta_{i,m_i}], \mathtt{E}).$$

   where *fevid* is a macro recursively defined as

$$\mathit{fevid}([], \mathtt{E}) \stackrel{def}{=} \mathtt{fail}$$

$$\mathit{fevid}([\beta_1|B], \mathtt{E}) \stackrel{def}{=} (\overline{\beta_1}'_{\mathtt{E^f_1}} \texttt{ -> } \mathtt{E} = [((\beta_1, false), \mathtt{E^f_1})] \\ ; \ \mathtt{all_evid}((\beta'_{1\mathtt{E^t_1}}, \mathtt{E^t_1}), (\mathit{fevid}(B, \mathtt{E^f_2}), \mathtt{E^f_2}), \mathtt{E}).$$

   and $\mathtt{E}$,$\mathtt{E^f_1}$,$\mathtt{E^t_1}$, and $\mathtt{E^f_2}$ are distinct new variables. For each atom $\beta = q(u_1, \ldots, u_l)$ appearing in the body of a clause, its extended dual expression is defined as

$$\overline{\beta}'_{\mathtt{E}} = \begin{cases} (\mathtt{table_not}(q'(u_1, \ldots, u_l)), \mathtt{E} = \mathit{ref}(\beta)) & \text{(if } \beta \text{ is a tabled predicate)} \\ \overline{q}'(u_1, \ldots, u_l, \mathtt{E}) & \text{(otherwise)} \end{cases}$$

Fig. 3. Transformation Rules for False Literals

5 Practical Aspects

In Sections 3 and 4, we described general transformation rules for pure logic programs. In practice, however, most programs have non-logical constructs (such as `assert`, `retract`) and meta-operators (such as `var`, `nonvar`). To make online justification practical, below we describe transformation techniques necessary to handle such programs. In addition, we show how online justification can be used as a flexible evidence explorer.

Meta-operators. Since we transform predicate names into different names using p' and $\overline{p}'$, literal `call(P)` in the original program has to be transformed so that it calls the appropriate transformed literal during execution. If `P` is a non-variable, then it is transformed using the general transformation rules; otherwise, it is transformed into $\mathit{call}'(P)$ or $\overline{\mathit{call}}'(P)$, depending on whether `call` is being

transformed as true literal or false literal, respectively. $call'(P)$ and $\overline{call}'(P)$ are part of run-time support for the justifier, which transforms P during execution and call the resulting predicate.

XMC model checker uses predicate forall(Vars, Antecedent, Consequent) which is defined as

```
forall(_Variables, Antecedent, Consequent) :-
   findall(Consequent, Antecedent, AllConsequents),
   all_true(AllConsequents).
all_true([]).
all_true([C|Cs]) :- call(C), all_true(Cs).
```

If forall is transformed using the general transformation rules, then all_true generates quadratic amount of evidence. The transformed predicates for forall should first collect evidence for antecedent from findall, evidence for consequents from all_true and then assemble both to generate the evidence for forall. There is no general technique to handle this behavior; hence we implemented these transformed predicates by hand.

Controlling Evidence. As can be seen in Section 6, the transformed program has very little time overhead, but in some cases, the space overheads may be high; e.g., justification of non-tabled predicates such as append take exponential amount of space to store the evidence due to recursive definitions. To avoid generating evidence for such predicates, we provide a mechanism to specify the predicates the user intends to justify and transform only those predicate definitions. Limiting the amount of evidence not only results in reducing the overheads, but also helps the user explore the evidence easily.

Constructs with Side-effects. Pure logic predicates have no side-effects, hence can be executed many times, without changing the semantics of the program. However, assert and retract cannot be executed more than once, as they change the semantics of the program. For true literals, the evidence is generated during the evaluation of the program, which can be retrieved without executing the literal any more. The dual definitions for the literals, on the other hand, should not call assert and retract, as the original program wouldn't have executed them. To avoid changing the database during execution of false literals, the predicates with assert/retract can be transformed in such a way that they first make a copy of the current database into another database, during the execution change only the copy and at the end delete the copied database. Thus, executing the dual definition doesn't change the database.

Flexible Evidence Generation and Exploration. The segments of evidence generated during query evaluation in online justification can then be used to generate the full evidence graph. So online justification can be used as a tool to debug programs. However, it has a few fundamental differences with the traditional trace-based debuggers. Justification gives a declarative view of the logic program, displaying sufficient and necessary information to establish the truth or

falsity of a query, whereas debuggers provide only the procedural view of the execution. Online justification gives flexibility in both generating and exploring the evidence generated during evaluation by allowing the user to explore the evidence as and when necessary, skipping over uninteresting portions and even revisiting the skipped portions later without restarting the debugging process.

6 Experimental Results

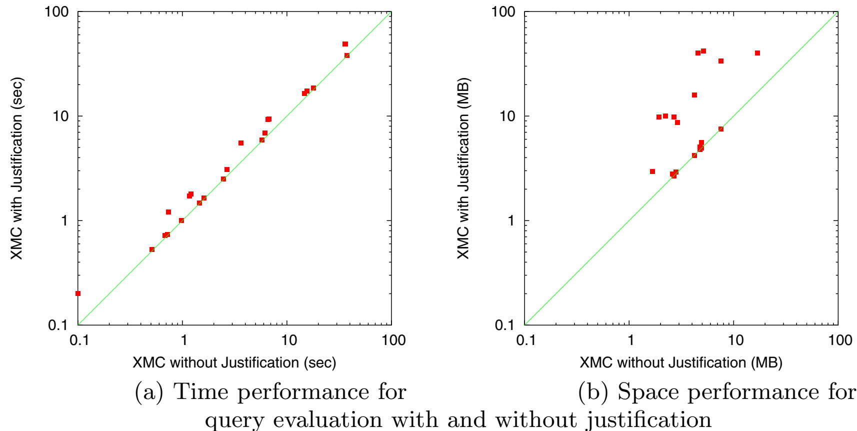
One of the primary goals of our work is to implement a *practical* tool for justification of tabled logic programs. To measure the overheads of time and space on such programs, we have used the justifier to automatically transform the XMC model checker[15] to construct the evidence for model checking. The entire implementation of the model checker consists of about 150 lines of XSB, most of which are definitions for the non-tabled predicate `models(S,F)`, which checks if state `S` models the μ-calculus formula `F`, and one definition for a tabled predicate `rec_models(S,F)`, which checks if state `S` models formula definition of `F`. A fragment of this model checker is given below:

```
:- table rec_models/2.
rec_models(State_s, X) :- fDef(X, Y), models(State_s, Y).
models(_State_s, tt).
models(State_s, fAnd(X_1, X_2))  :-
   models(State_s, X_1), models(State_s, X_2).
models(State_s, fOr(X_1, X_2))   :-
   models(State_s, X_1) ;  models(State_s, X_2).
models(State_s, fDiam(Act_a, X)) :-
   transition(State_s, Act_a, State_t), models(State_t, X).
```

Since `models(S,F)` is a non-tabled predicate, the justifier transforms it into two predicates: `models_t(S,F,E)`, corresponding to the true-literal justification and `models_f(S,F,E)`, corresponding to the false-literal justification, where `E` is the evidence. The `rec_models` is transformed so that the evidence of the definition and the evidence from `models/2` are stored in the evidence database. The transformed model checker is then used on some examples from XMC test suite [5] with various system sizes: Iprotocol, Leader election, Java meta-lock and Sieve. The system sizes that we have tried are very large (requiring upto 1GB of system memory). Here we report the time and space performance of this model checker along with the original XMC model checker.

All the tests were performed on Intel Xeon 1.7GHz machine with 2GB RAM running RedHat Linux 7.2 and XSB version 2.5 (optimal mode, slg-wam with local scheduling) with the garbage collector turned off.

Figure 6(a) compares the query evaluation time of the original XMC (without justification) with the transformed XMC (with justification). In our experiments, the transformed program took at most 50% (and on average 23%) longer time compared to the original program. Note that all the graphs are drawn in logarithmic scale, with performance of XMC without justification on X-axis and the performance of transformed XMC with justification on Y-axis.



(a) Time performance for (b) Space performance for
query evaluation with and without justification

Fig. 4. Time and Space Performance of XMC with and without Justifier

Figure 6(b) shows the space overhead due to evidence generation. In our experiments, the transformed program has maximum overhead of 11 times when evidence is generated for every state along every path in the model (due to `forall`) in the case of Leader election protocol, and on average about 3.5 times the overhead. The comprehensive details about time and space performance of the online justifier tool can be found at `http://www.lmc.cs.sunysb.edu/~lmc/justifier`.

7 Conclusions

In this paper, we presented a new justification scheme using program transformation. In this scheme, a logic program is automatically translated such that the translated program builds evidence during query evaluation. The evidence so generated can be presented later to the user using an interactive interface. The extra overhead due to the evidence generation is so little that the tool we implemented using this scheme has been used in practice to generate evidence for model checking practical systems. We plan to extend our scheme to handle logic programs with side effects, and to integrate our implementation with the Evidence Explorer [6], so that the user can easily navigate the evidence.

References

1. S. Basu, D. Saha, Y.-J. Lin, and S. A. Smolka. Generation of all counter-examples for push-down systems. In *FORTE*, 2003.
2. M. Cameron, M. García de la Banda, K. Marriott, and P. Moulder. Vimer: a visual debugger for mercury. In *Proceedings of the 5th ACM SIGPLAN*, 2003.

3. E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *32*nd *Design Automation Conference*, 1995.
4. P. Deransart, M. Hermenegildo, and J. Maluszynski. *Analysis and Visualization Tools for Constraint Programming: Constraint Debugging*, volume 1870 of *LNCS*. Springer, 2000.
5. Y. Dong and C. R. Ramakrishnan. Logic programming optimizations for faster model checking. In *TAPD*, Vigo, Spain, September 2000.
6. Y. Dong, C. R. Ramakrishnan, and S. A. Smolka. Evidence Explorer: A tool for exploring model-checking proofs. In *Proc. of 15*th *CAV*, LNCS, 2003.
7. J. Freire, T. Swift, and D. Warren. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. In *PLILP*, 1996.
8. H.-F. Guo and G. Gupta. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In *ICLP*, volume 2237 of *LNCS*, 2001.
9. H.-F. Guo, C. R. Ramakrishnan, and I. V. Ramakrishnan. Speculative beats conservative justification. In *ICLP*, volume 2237 of *LNCS*, 2001.
10. J. Lloyd. Declarative error diagnosis. In *New Generation Computing*, 1987.
11. J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
12. S. Mallet and M. Ducasse. Generating deductive database explanations. In *ICLP*, 1999.
13. L. Naish, P. Dart, and J. Zobel. The NU-Prolog debugging environment. In *ICLP*, 1999.
14. G. Puebla, F. Bueno, and M. Hermenegildo. A framework for assertion-based debugging in constraint logic programming. In *LOPSTR*, volume 1817 of *LNCS*, 1999.
15. C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V. N. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *12*th *CAV*, volume 1855 of *LNCS*. Springer, 2000.
16. P. Roop. *Forced Simulation: A Formal Approach to Component Based Development of Embedded Systems*. PhD thesis, Computer Science and Engineering, University of New South Wales, 2000.
17. A. Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying proofs using memo tables. In *Principles and Practice of Declarative Programming*. ACM Press, 2000.
18. E. Shapiro. Algorithmic program diagnosis. In *Proceedings of POPL'82*, 1982.
19. G. Specht. Generating explanation trees even for negations in deductive database systems. In *ILPS'93 Workshop on Logic Programming Environments*, 1993.
20. T. Swift and D. S. Warren. An abstract machine for SLG resolution: definite programs. In *Proceedings of the Symposium on Logic Programming*, 1994.
21. D. S. Warren. *Programming in Tabled Prolog*. 1999. Early draft available at `http://www.cs.sunysb.edu/~warren/xsbbook/book.html`.
22. XSB. The XSB logic programming system. Available at `http://xsb.sourceforge.net`.