Speculative Beats Conservative Justification *

Hai-Feng Guo, C.R. Ramakrishnan, and I.V. Ramakrishnan

State University of New York at Stony Brook
Stony Brook, NY 11794, USA
Email: {haifeng,cram,ram}@cs.sunysb.edu

Abstract. Justifying the truth value of a goal resulting from query evaluation of a logic program corresponds to providing evidence, in terms of a proof, for this truth. In an earlier work we introduced the notion of justification [8] and gave an algorithm for justifying tabled logic programs by *post-processing* the memo tables created during evaluation. A *conservative justifier* such as the one described in that work proceeds in two separate stages: evaluate the truth of literals (that can possibly contribute to the evidence) in the first stage and construct the justification in the next stage. Justifications built in this fashion seldom fail. Whereas for tabled predicates evaluation amounts to a simple table look-up during justification, for non-tabled predicates this amounts to Prolog-style re-execution. In a conservative justifier a non-tabled literal can be re-executed causing unacceptable performance overheads for programs with significant non-tabled components: justification time for a single non-tabled literal can become *quadratic* in its evaluation time!
In this paper we introduce the concept of a *speculative justifier*. In such a justifier we evaluate the truths of literals in tandem with justification. Specifically, we select literals that can possibly provide evidence for the goal's truth, assume that their truth values correspond to the goal's and proceed to build a justification for each of them. Since these truths are not computed before hand, justfications produced in this fashion may fail often. On the other hand non-tabled literals are re-executed less often than conservative justifiers. We discuss the subtle efficiency issues that arise in the construction of speculative justifiers. We show how to judiciously balance the different efficiency concerns and engineer a speculative justifier that addresses the performance problem associated with conservative justifiers. We provide experimental evidence of its efficiency and scalability in justifying the results of our XMC model checker.

1 Introduction

Query evaluation of a goal with respect to a logic program establishes the truth or falsehood of the goal. However the underlying evaluation engine typically provides little or no information as to why the conclusion was reached. This problem broadly falls under the purview of debugging. Usually logic programs

* Research partially supported by NSF awards EIA-9705998, CCR-9876242, IIS-0072927, and EIA-9901602.

are debugged using trace-based debuggers (e.g. Prolog's four-port debugger) that operate by tracing through the entire proof search. Such traces are aided through several navigation mechanisms (e.g. setting breakpoints or spy points, skips, leaps, etc.) provided by the debugger.

There are several reasons why trace-based debuggers are cumbersome to use. Firstly, they give the entire search sequence including all the failure paths, which is essentially irrelevant if the user is only interested in comprehending the essential aspects of how the answer was derived. Secondly, the proof search strategy of Prolog, with its forward and backward evaluation, already makes tracing a Prolog execution considerably harder than tracing through procedural programs. The problem is considerably exacerbated for tabled logic programs since the complex scheduling and fixed-point computing strategies of tabled resolution makes it very difficult to comprehend the sequence produced by a tracer. Finally, from our own experience with the XMC model checker [1] (which is an application of the XSB tabled logic programming system [11]) trace-based debuggers provide no support for translating the results of the trace (which is at the logic program evaluation level) to the problem space (e.g. CCS expressions and modal-mu calculus formulas in XMC).

In [8] we proposed the concept of a *justifier* for giving evidence, in terms of a proof, for the truth value of the result generated by query evalaution of a logic program. The essence of justification is to succinctly convey to the user only those parts of the proof search which are relevant to the proof/disproof of the goal. For example, if a query is evaluated to true, the justifier will present the details of a successful computation path, completely ignoring any unsuccessful paths traversed. Similarly, when a query is evaluated to false, it will only show a false literal in each of its computation paths, completely ignoring the true literals. Figure 1 is an illustration of justification, where the predicate `reach/2` (Figure 1a) is tabled. Evaluation of the query `reach(a,d)` generates a forest of search trees (Figure 1b), (See [12] for an overview of tabled evaluation.)

Although justification is a general concept, the focus of our earlier work in [8] was on justifying tabled logic programs. Towards that end we presented an algorithm for justifying such programs by *post-processing* the memo tables created during query evaluation. To justify the answer to a query some "footprints" need to be stored during query evaluation. The justifier uses these footprints to extract evidence supporting the result. The naturalness of using a tabled LP system for justification is that the answer tables created during query evaluation serve as the footprints. Indeed during query evaluation the internally created tables implicitly represent the lemmas that are proved during evaluation. By using these lemmas stored in the tables, the justifier presents only relevant parts of the derivation to the user. In other words the additional information needed for doing justification comes for "free". Thus justification using tabled logic programming system is "non-intrusive" in the sense that it is completely decoupled from query evaluation process and is done only after the latter is completed. More importantly, justification is done without compromising the performance of query evaluation.

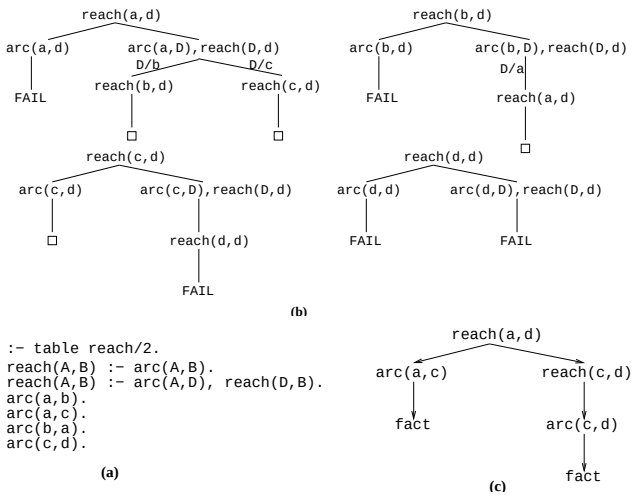


Fig. 1. Justifying reach(a,d): (a) Logic Program (b) Forest of Search Trees (c) Justification

Justifying the truth value of a given literal which we will denote as the goal, amounts to providing a proof that usually will involve searching for other literals relevant to the proof, knowing their truth values, justifying each such truth value and putting them all together to produce a justification of the goal's truth. For some of them we may fail to produce justifications relevant for justifying the goal. In Example 1 below the clause `p :- r` is irrelevant for justifying `p` is *true* since the failure of `r` is not the correct evidence for `p`'s truth. Had we selected this clause and proceeded to build a justification for `r` we would have eventually discovered that it is irrelevant. Thus avoiding irrelevant justifications is an important parameter in the design of justification algorithms.

Example 1 *Consider the following logic program:*

```
p :- r.        p :- t.
r :- ..., fail.
t.
```

The justification algorithm in [8] yields a *conservative justifier* in the sense that by design it is geared towards limiting such wasteful justifications. It does so by evaluating the truth of literals (that can possibly provide supporting evidence for the goal's truth) in the first stage. Armed with the needed truths, in a separate second stage it proceeds to construct their justifications. By evaluating the truth of `r` before hand upon selecting the clause `p :- r` in Example 1, we can avoid building the justfication of `r` to support the truth of `p` and fail eventually.

The algorithm in [8] implicitly assumed that all the predicates in the program are tabled. But real-life logic programs consist of both tabled and non-tabled predicates. How does it handle such programs? Whereas for tabled predicates evaluation is a simple table look-up during justification, for non-tabled predicates this amounts to Prolog-style re-execution. In a conservative justifier, justification

of a non-tabled literal can trigger repeated evaluations of other non-tabled literals on its proof path, causing unacceptable performance overheads for programs with significant non-tabled components. Specifically the time taken to justify the truth of a single non-tabled literal can become *quadratic* over its evaluation time! In fact on large model checking problems our XMC model checker took a few minutes to produce the results whereas the justifier failed to produce a justification even after sevaral hours!

In this paper we explore the concept of a *speculative justifier* to address the above performance problem associated with a conservative justifer. The idea underlying such a justfier is this: When we select a literal as a possible candidate for inclusion in the justificaton of the goal's truth we speculate that it will be relevent and proceed to build its justification. Since we do not know its truth value before hand we may discover eventually that we are unable to produce a justification for it that is relevant for justifying the goal's truth (such as the justification of `r` in Example 1). On the other hand if we never encounter any such literal then for a non-tabled literal we have built its justification without having to repeatedly traverse its proof path. But doing speculative justification naively can result in failing more often and thus offset any gains accrued by avoiding repeated re-executions of non-tabled literals. In this paper we discuss these subtle efficiency issues that arise in the design and implementation of speculative justifiers. We show how to judiciously balance the different efficiency concerns and engineer a speculative justifier that addresses the performance problem associated with conservative justifiers. The rest of the paper is organized as follows. In Section 2 we review the concept of justification. Section 3 reviews conservative justifier. In section 4 we present the design of a speculative justifier. In Section 5 we discuss its implementation and practical impact on real-world applications drawn from model checking. Discussion appears in Section 6. The technical machinery developed in this paper assumes *definite clause logic program*. Extensions are also disussed in Section 6.

Related Work A number of proposals to explain the results of query evaluation of logic programs have been put forth in the past. These include algorithmic debugging techniques [10], declarative debugging techniques [4, 6], assertion based debugging techniques [7], and explanation techniques [5]. A more detailed comparison between justification and these aproaches appears in our earlier work [8]. Suffice it is say here that although justification is similar in spirit to the above approaches in terms of their objectives it differs considerably from all them. It is done as a post-processing step after query evaluation, and not along with the query evaluation (as in algorithmic and assertion-based debugging) or before query evaluation (as in declarative and assertion-based debugging). 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 to do as will be the case in model checking. But beyond all that this paper examines effciency issues that arise in justifying logic programs consisting of both tabled and non-tabled predicates – a topic that has not been explored in the literature.

2 Justification

In this section we will recall the formalisms developed in [8] for justification. We generalize them here in order to deal with mixed programs containing both tabled and non-tabled predicates.

Notational Conventions We use P to denote logic programs; $HB(P)$, $M(P)$ to denote the Herbrand Base and least Herbrand model respectively; `A` and `B` to denote atoms or literals; α to denote a set of atoms or literals; β to denote a conjunction of atoms (a goal is a conjunction of atoms) or literals; θ to denote substitutions; '$\succeq$' to denote atom subsumption ($A \succeq B$ for A subsumes B); and C to denote a clause in a program. For a binary relation R, we denote its (reflexive) transitive closure by R^*.

Definition 1 (Truth Assignment) *The truth assignment of an atom A with respect to program P, denoted by $\tau_{(P)}(A)$, is:*

$$\tau_{(P)}(A) = \begin{cases} true & \forall \theta \; A\theta \in M(P) \\ false & \forall \theta \; A\theta \notin M(P) \end{cases}$$

We drop the parameter P and write the truth assignment as $\tau(A)$ whenever the program is obvious from the context. Let A be an answer to some query in program P, i.e., $\tau(A) = true$. We can complete one step in explaining this answer by finding a clause C such that (i) A unifies with the head of C, and (ii) each literal B in the body of C has $\tau(B) = true$. If A is not an answer to any query, i.e., $\tau(A) = false$, we can explain this failure by showing that for each clause C whose head unifies with A, there is at least one literal B in C such that $\tau(B) = false$. We call such one-step explainations as a *locally consistent explanations (lce's)*, defined formally as follows:

Definition 2 (locally consistent explanation (lce)) *Locally consistent explanation for an atom A with respect to program P, denoted by $\xi_{(P)}(A)$, is a collection of sets of atoms such that:*

1. If $\tau(A) = true$:
   $\xi_{(P)}(A) = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$, with each α_i being a set of atoms $\{B_1, B_2, \ldots, B_n\}$ such that:
   (a) $\forall \; 1 \leq j \leq n \;\; \tau(B_j) = true$, and
   (b) $\exists \; C \equiv A' \text{ :- } \beta$ and a substitution θ such that $A'\theta = A$ and $\beta\theta \equiv (B_1, B_2, \ldots, B_n)\theta$.
2. If $\tau(A) = false$:
   $\xi_{(P)}(A) = \{L\}$, a singleton collection where $L = \{B_1, B_2, \ldots, B_n\}$ is the smallest set such that:
   (a) $\forall 1 \leq j \leq n \;\; \tau(B_j) = false$, and
   (b) $\forall$ substitutions θ and $C \equiv A' \text{ :- } (B'_1, B'_2, \ldots, B'_l)$, $A'\theta = A\theta \implies \exists 1 \leq k \leq l$ such that $B'_k\theta \in L$ and $\forall \; 1 \leq i < k \;\; \tau(B'_i\theta) = true$.

$\xi(reach(a,d)) = \{\{arc(a,c), reach(c,d)\}\}$
$\xi(reach(c,d)) = \{\{arc(c,d)\}\}$
$\xi(arc(c,d)) = \{\{\}\}$
$\xi(reach(a,c)) = \{\{arc(a,c)\},\ \ \{arc(a,b), reach(b,c)\}\}$

(a) lce's for true literals

$\xi(reach(a,e)) = \{\{arc(a,e), reach(b,e), reach(c,e)\}\}$
$\xi(reach(b,e)) = \{\{arc(b,e), reach(a,e)\}\}$
$\xi(arc(a,e)) = \{\{\}\}$

(b) lce's for false literals

Fig. 2. A fragment of lce's for the example in Figure 1

We write $\xi_{(P)}(A)$ as $\xi(A)$ whenever the program P is clear from the context.

Observe that, for an atom A, the different sets in the collection $\xi(A)$ represent different consistent explanations for the truth or falsehood of A. An answer A can be explained in terms of answers $\{B_1, B_2, \ldots, B_k\}$ in $\xi(A)$ and then (recursively) explaining each B_i. e.g. $\xi(reach(a,d))$ in Figure 2 has a set with elements $arc(a,c)$ and $reach(c,d)$, meaning that the truth value (*true*) of $reach(a,d)$ can be explained using the explanations of $arc(a,c)$ and $reach(c,d)$. Such explanations can be captured by a graph as shown in Figure 1(c). The edges denote locally consistent explanations. We do not use cyclic explanations to justify a true literal. In contrast, cyclic explanations describe infinite proof paths and can be used to justify a false literal. Instead of explicitly representing these cycles, however, we choose to keep the justification as an acyclic graph, breaking each cycle by redirecting at least one edge to a special node marked as `ancestor`. Formally:

Definition 3 (Justification) *A justification for an atom A with respect to program P, denoted by $\mathcal{J}(A,P)$, is a directed acyclic graph $G = (V,E)$ with vertex labels chosen from $HB(P) \cup \{$`fact`,`fail`, `ancestor`$\}$ such that:*

1. G `is rooted at` A`, and is connected`
2. $(B_1, \texttt{fact}) \in E \iff \{\} \in \xi(B_1) \wedge \tau(B_1) = true$
3. $(B_1, \texttt{fail}) \in E \iff \{\} \in \xi(B_1) \wedge \tau(B_1) = false$
4. $(B_1, \texttt{ancestor}) \in E \iff \tau(B_1) = false \wedge \xi(B_1) = \{L\}$
   $\wedge\ \exists\ B_2 \in L$ `s.t.` $(B_2, B_1) \in E^* \ \vee\ B_2 = B_1$
5. $(B_1, B_2) \in E \ \wedge\ \tau(B_1) = false \iff$
   $\xi(B_1) = \{L\} \ \wedge\ B_2 \in L \ \wedge\ (B_2, B_1) \notin E^* \ \wedge\ B_2 \neq B_1$
6. $(B_1, B_2) \in E \ \wedge\ \tau(B_1) = true \implies$
   $\exists L \in \xi(B_1)$ `s.t.` $B_2 \in L \ \wedge\ \{\forall B' \in L\ (B', B_1) \notin E^* \ \wedge\ B' \neq B_1\}$
7. $B_1 \in V \ \wedge\ \tau(B_1) = true \implies \exists\ unique\ L \in \xi(B_1)$ `s.t.`
   $\forall B_2 \in L\ (B_1, B_2) \in E \ \wedge\ (B_2, B_1) \notin E^* \ \wedge\ B_2 \neq B_1$

Rule 1 ensures that A is the root of justification. Rules 2 and 3 are the conditions for adding leaf nodes based on facts. Rules 4 and 5 specifies conditions for justifying false literals, while Rules 6 and 7 deal with true literals.

We will denote the justification graph built for a true (false) literal as *true-justification (false-justification)*.

e.g.. the true-justification in Figure 1(c) is built as follows: $reach(a,d)$ is the root (by rule 1). Now consider the lce $\{arc(a,c), reach(c,d)\}$ in $\xi(reach(a,d))$. Since every element in this lce does not form a cyclic explanation, and is different from $reach(a,d)$, both edges $(reach(a,d), arc(a,c))$ and $(reach(a,d), reach(c,d))$ are added to the justification (by Rule 6). Rule 7 guantees that one and only one lce is added into the justification. Next we construct true-justifications for $arc(a,c)$ and $reach(c,d)$ recursively.

3 Conservative Justifier

We review our algorithm in [8] to construct the justification graph. Its high-level aspects are skecthed in Figure 3. V denotes the vertices (labelled by literals in the ξ's) and E denotes the edges in this graph.

Given a literal A the algorithm builds the graph recursively, traversing it depth-first even as it is constructed. At any point, V is the set of "visited" vertices, and *Done* is the set of vertices whose descendents have been completely explored. $V - Done$ contains exactly those vertices that are ancestors to the current vertex A.

```
algorithm Justify(A : atom)
    (* Global: P : program, (V, E): Justification, Done ⊆ V *)
  if (A ∉ V) then (* A has not yet been justified *)
    set V := V ∪ {A}
    if (τ(A) = true) then (* true-justification *)          (1)
      let α_A ∈ ξ(A) such that (α_A ∩ V) ⊆ Done             (2)
      if (α_A = {}) then
        set E := E ∪ (A, fact)
      else
        for each B ∈ α_A do
          set E := E ∪ (A, Justify(B))
    else (* false-justification *)
      let {α_A} = ξ(A)                                      (3)
      if (α_A = {}) then
        set E := E ∪ (A, fail)
      else
        if ((α_A ∩ V) ⊈ Done) then
          set E := E ∪ (A, ancestor)
        for each B ∈ (α_A − (V − Done)) do
          set E := E ∪ (A, Justify(B))
    set Done := Done ∪ {A}
```

Fig. 3. Justification Algorithm

The algorithm is structured as follows: it takes the literal A whose truth is to be justified as the input parameter. It will determine a locally consistent explanation for either a true-justification in case $\tau(A) = true$ (line 2) or a false-justification otherwise (line 3). Finally it justifies the literals in the explaination

set recursively. The selection of the justification is done by backtracking through **let**. Correctness of *Justify* appears in [8].

Algorithm *Justify* in [8] had assumes that all the predicates in the program are tabled. Let us analyse its behavior on "mixed" programs containing both tabled and non-tabled predicates. Observe that the algorithm computes the explanation set for A prior to building the justification graph rooted at A. Computing the explanation set corresponds to evaluating the truth values of literals in the set. Observe that this evaluation is done prior to justifying the truths of the literals in α_A. This ensures that the justifications of the truths of literals in α_A do not fail. In fact the only time a justification gets discarded is when there is a cycle in a true-justification. Algorithm *Justify* is the basis of a *conservative justifier.*

3.1 Efficiency Issues in Conservative Justification

Using the XSB tabled LP system we implemented *Justify* as a post-processing step following query evaluation. The advantage of using a tabled system for justification is that the answers in the tables can be directly used for computing the ξ's (lines 2 and 3). In particular if all the predicates are tabled then the truth value of all the literals are stored in the tables. Hence selecting a $\xi(A)$ amounts to a simple table lookup. In fact we can show:

Proposition 1 *For a logic program consisting of tabled predicates only, the running time of Justify is proportional to the time taken by initial query evaluation.*

Let us examine the behavior of *Justify* on a program containing both tabled and non-tabled predicates. In a tabled LP system there is no provision for storing the truth value of non-tabled literals. Consequently computing ξ's can become expensive since non-tabled predicates must be re-executed (a-la Prolog style) to ascertain their truth values. In fact, as is shown below, the time for justifying a single non-tabled literal can become quadratic its original evaluation time..

Example 2 *Consider the following non-tabled factorial logic program:*

```
fac(0, 1).
fac(N, S) :- N > 0, N1 is N - 1, fac(N1, S1), S is S1 * N.
```

Assume that `fac(N,S)` is evaluted for some fixed n. It is easy to see that evaluation time is $O(n)$. The call to *Justify*`(fac(n,n!))` will first compute ξ`(fac(n,n!))`. This set will include `fac(n-1,(n-1)!)`. Algorithm *Justify* takes $O(n-1)$ steps to compute ξ`(fac(n,n!))` since evaluting the truth value of `fac(n-1,(n-1)!)` requires that many steps. Next *Justify*`(fac(n-1,(n-1)!))` is invoked and the above process is repeated. It is easy to see that *Justify*`(n,n!)` will require $O(n^2)$ time.

One can however table all the predicates in a program. In such a case the truths of `fac(n,n!)`, `fac(n-1,(n-1)!)`, ..., `q(0,1)` are all stored in an answer table upon completion of query evaluation. Justifiation will require $O(n)$

time since evaluating the truths of each of the *fac*'s can be done in $O(1)$ time. But for time and space efficiency predicates are selectively tabled in practice [3]. The interesting question now is this: Can we design an efficient justifier for mixed programs without having to suffer the overheads of re-execution of non-tabled predicates? Indeed our interest in this question was mainly motivated by our expereince with our XMC justifer for model checking [1]. On large model checking problems the XMC model checker took a few minutes to produce the results whereas the justifier failed to produce the justification even after sevaral hours! In the next section we will present an answer to this question.

4 Speculative Justifier

The idea behind a speculative justifer is as follows: Suppose we wish to justify the truth of p and further suppose there is a clause p :- $q_1, q_2, ..., q_n$ in the program. Further suppose we wish to build a true-justification for p. If $\{q_1, q_2, \ldots, q_n\} \in \xi(p)$ then one can build a justification for p by building true-justifications for each of the q_i's, $(1 \leq i \leq n)$. Without evaluating their truths *apriori* we speculate that $\{q_1, q_2, \ldots, q_n\} \in \xi(p)$ and attempt to build a true-justification for all of them. If $\{q_1, q_2, \ldots, q_n\} \in \xi(p)$ then all these justifications will succeed and result in a true-justification for p. If $\{q_1, q_2, \ldots, q_n\} \notin \xi(p)$ then there must exist at least one q_i for which the attempt at building a true-justification for it will fail. Hence this clause cannot provide any evidence as to why p is *true* and we proceed to find another candidate clause. Now suppose we wish to build a false-justification for p. We speculate again that there must exist at least one q_i that is *false*. So we attempt building a false-justification for each of the q_i's in sequence. If we fail to build a false-justification for any of the q_i's then we can conclude that a false-justification for p does not exist. On the other hand if we do succeed then we repeat this process on the next clause that unifies with p. Recall from definition of justification that to justify that p is *false* there must exist a false literal in each of these clauses.

The main advantage of speculative justifiers can be seen when justifying non-tabled predicates. Recall non-tabled literals are re-executed during justification. Speculative justifiers re-execute less often than their conservative counterparts. Consider $\{q_i$:- $q_{i-1}.|1 \leq i \leq n\}$, n is a constant and q_0 is a fact. To build a true-justification for q_n the speculative justifier will attempt to build a true-justification for q_{n-1} which in turn build a true-justification for q_{n-2}, and so on. All of these justifications succeed without ever having to repeat re-execution of any of the q_i's in q_n's proof.

4.1 Efficiency Issues

But speculative justifiers can suffer from inefficiencies. For example, the gains accrued by re-executing non-tabled literals less often can be easily offset by wasted justifications. We discuss these problems below:

- **The Problem of Wasteful Justifications:**
  Naive implementation of speculative justifiers can result in building wasteful justifications that are eventually discarded. For example, suppose we wish to build a true-justification for p using the clause pick $p :- q, r$. Suppose q is *true* and r is *false*. We will succeed in building a true-justificaton for q but fail to do so for r. So using this clause we will fail to build a true-justification for p. But the true-justification built for q is wasted.
- **The Problem of Rebuilding Justifications:**
  In the above example justification of q was discarded as being irrelevant for justifying p. Now suppose later on we encounter the literal q again during justification. If we do not save the justifcation of q then we will have rebuild its justification all over again.

We now propose solutions to these two main sources of inefficiency in a speculative justifier.

Lazy Justification To avoid wasteful justifications we justify tabled literals *lazily*. The idea is this: Let us suppose we select the clause $p :- q_1, q_2, ..., q_n$ for justifying p. Assume we wish to build a true-justification for p. Suppose the literal currently on hand, say q_i, is tabled. Then we do a simple table-lookup to verify that $\tau(q_i)$ is *true*. If this is the case we defer building its justification and move on to the next literal in the sequence. If q_i is non-tabled then we build true-justification for it. We proceed to build justifications for the tabled literals in the clause only after we succeed building true-justifications for all of its non-tabled literals. This idea carries over for false-justifications also.

Sharing Justfications The solution to re-building justifications is to save all of them after they are built the first time. We save the justifications of both tabled and non-tabled literals. But this can result in space ineffciencies especially if sharing is infrequent and irrelevant justifications outweigh relevant ones. A practical compromise between never re-building and always re-building is to share the justifications of tabled literals only. But note that justification of a tabled literal might involve other tabled literals. So we will have to avoid copying the entire justification. Instead we save a "skeleton" of the justification from which we can reproduce the complete justification. We call this skeleton *partial justification*. Intuitively the leaf nodes of a partial justification are either labelled `fail, fact, ancestor` or by a tabled literal. All of the interior nodes except the root are labelled by non-tabled literals. Formally:

Definition 4 (Partial Justification) *A partial justification for an atom A with respect to a program P and table T, denoted by $\mathcal{P}_{(P,T)}(A)$, is a directed acyclic graph $G = (V, E)$ with vertex labels chosen from $HB(P) \cup \{$`fact`, `fail`, `ancestor`$\}$ and the edges from $\{(B_1, B_2) | B_1 = A \vee B_1 \notin T\}$. The conditions for selecting the edges are the same as those used in defining justification (def. 3).*

We drop the parameter P and T and write the partial justification as $\mathcal{P}(A)$ whenever the program and the table are obvious from the context.

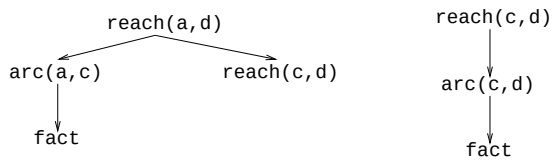


Fig. 4. Partial Justification of $reach(a,d)$ and $reach(c,d)$ in Figure 1

Figure 4 denotes the partial justifications of $reach(a,d)$ and $reach(c,d)$ for the example in Figure 1.

We can compose partial justifications together to yield a complete justification for a literal. Informally composition amounts to "stringing" together the partial justifications of tabled literals at the leaf nodes labelled by those literals. For example in Figure 4, by attaching the partial justification of $reach(c,d)$ to the leaf node labelled $reach(c,d)$ in the partial justification of $reach(a,d)$ yields its complete justification. However care must be exercised when composing partial justifications. In particular compositions that produce cycles in true-justifications must be discarded.

4.2 Algorithmic Aspects of Speculative Justification

The speculative justifier builds a justifcation by composing several partial justifications. The algorithm for partial justification is shown in Figure 5. It takes the following parameters as its input: (i) A which is the literal to be justified, (ii) A's truth value *Tval* and (iii) *Anc* which is a list of tabled literals that are ancestors of A in the justification. The algorithm builds a true(false)-justification if *Tval* is *true (false)*. It returns in J the partial justification of A and D those tabled calls which appears in the leaf nodes of J. We use `clause(A,B)` to pick a clause that unifies with A and `findall` for aggregation. T denotes the tabled literals and their answers.

Recall that to build the complete justification of A we need to know the partial justifications of all the tabled literals that the justification of A depends upon (e.g. *reach(a,d)* depends on *reach(c,d)* in Figure 4). Let $D_A = \{P | P$ is a tabled literal that appears as the label of a leaf node in $\mathcal{P}(A)$ $\}$. We refer it to as the *dependent set*. We will drop the subscript from the notation for the dependent set if the literal that it is associated with is clear from the context.

4.3 Properties of a Speculative Justifier

We will suppose that a speculative justifier is based on algorithm *partial-justify* and that the complete justification for any literal is obtained by composing all the partial justifications of tabled literals it depends on. We state below some of its important properties.

Proposition 2 *On purely tabled logic programs, speculative justifier coincides with conservative justifier.*

The above is based on the observation that to justify `A` the speculative justifier generates a partial justification which includes its dependent set and fact nodes. They correspond to a `lce` for `A`.

Proposition 3 *On purely non-tabled logic programs, justification time required by a speculative justifier is proportional to query evaluation time.*

This proposition is based on the observation that when a program has no tabled predicates then the partial justification for A corresponds to complete justification and that evaluation proceeds in Prolog-style.

Theorem 4 *The time taken by a speculative justifier for justification is no more than the time taken by a conservative justifier*

We sketch only the main observation for establishing the above propoerty. Note that a conservative justifier computes a lce for A by re-executing non-tabled

```
algorithm Partial-Justify(A : atom, Tval : truth value, Anc : Ancestors)
    (* Local: J : Justification (V, E); D : Dependent Set *)
  set (J, D) := (({A}, {}), {})
  if ( Tval = true ) then (* build true-justification *)
    clause(A, B)
    if ( B = true ) then(* the selected clause is a fact *)
      set J := ({A, fact}, {(A, fact)})
    else
      for each G ∈ B then
        if ( G ∈ T ) then
          if ( τ(G) = true ) then
            if (G ∈ Anc) then
              fail
            else
              set E := E ∪ {(A, G)}
              set D := D ∪ {G} (* add G to the dependent set *)
          else (* τ(G) ≠ true *)
            fail
        else (* G is a non-tabled call *)
          set E := E ∪ {(A, G)}
          set (J, D) := (J, D) ∪ partial-justify(G, Tval, Anc)
  else (* build false justification *)
    findall(B, clause(A, B), BL)
    if (BL = {}) then (* no clause unifies with A *)
      set J := ({A, fail}, {< A, fail >})
    else
      for each B ∈ BL do
        let G ∈ B (* G is choosen from B sequentially *)
        if ( G ∈ T ) then
          if ( τ(G) = false ) then
            if (G ∈ Anc) then
              set E := E ∪ {(A, ancestor)}
            else
              set E := E ∪ {(A, G)}
              set D := D ∪ {G} (* add G to the dependent set *)
          else (* τ(G) ≠ Tval *)
            fail
        else (* G is a non-tabled call *)
          set E := E ∪ {(A, G)}
          set (J, D) := (J, D) ∪ partial-justify(G, Tval, Anc)
  return (J, D)
```

Fig. 5. Speculative Justification

literals and consulting the answer table for tabled literals. This coresponds to computing the partial justification of A by a speculative justifier. Besides the search paths for computing lce's in a conservative justifier and partial justifications in a speculative justifiier also correspond.

While the above theorem only says that the time taken is proportional, speculative justifiers can do better. Consider the non-tabled factorail program in Example 2. By avoiding repeated re-executions the speculative justifier will build a true justification for `(fac(n,n!))`in $O(n)$ steps whereas it took $O(n^2)$ steps for the conservative justifier.

5 Experimental Results

In [8] we reported on the performance of a conservative justifier based on *Justify* (in Section 3) and implemented using the XSB tabled LP system. It was de-

Benchmark	**Leader (ae_leader)**					**Leader (one_leader)**				
Size	**2**	**3**	**4**	**5**	**6**	**2**	**3**	**4**	**5**	**6**
Conservative	0.18	1.51	10.86	130.3	n/a	0.19	1.41	11.01	136.6	2252.7
Speculative	0.05	0.24	1.21	6.80	35.2	0.06	0.22	1.17	6.04	33.2

Benchmarks	**Sieve (ae_finish)**									
Size	**(3,4)**	**(3,5)**	**(4,5)**	**(4,6)**	**(5,6)**	**(5,7)**	**(6,7)**	**(6,8)**	**(6,9)**	**(6,10)**
Conservative	1.12	1.24	3.65	4.60	11.92	15.71	46.83	51.5	62.8	78.29
Speculative	0.16	0.18	0.42	0.52	1.17	1.45	3.38	3.69	4.13	4.80

Benchmarks	**Meta-lock (mutex)**						**ABP**	**Iproto (bug)**
Size	**(1,2)**	**(1,3)**	**(1,4)**	**(2,1)**	**(3,1)**	**(2,2)**		fix(1)
Conservative	2.11	21.95	310.1	4.77	239.0	488.4	1.81	n/a
Speculative	0.18	0.97	4.98	0.32	4.09	6.16	0.20	193.2

(a) Running Time (in Seconds.)

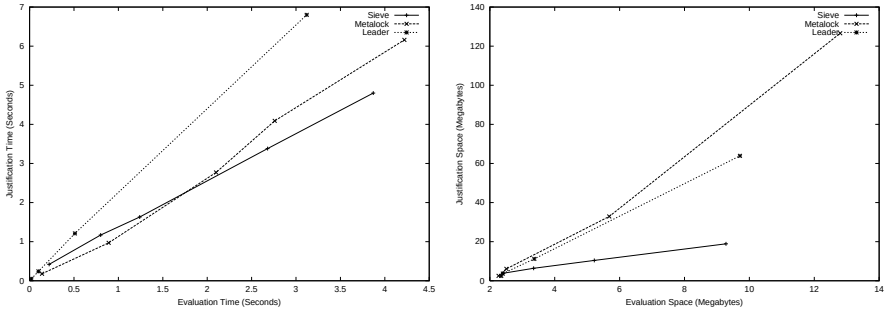
Benchmark	**Leader (ae_leader)**				**Leader (one_leader)**			
Size	**2**	**3**	**4**	**5**	**2**	**3**	**4**	**5**
Conservative	2.35	4.96	17.6	81.1	2.40	2.62	8.25	43.7
Speculative	2.48	3.68	10.4	63.7	2.48	3.68	10.5	63.9

Benchmarks	**Sieve (ae_finish)**									
Size	**(3,4)**	**(3,5)**	**(4,5)**	**(4,6)**	**(5,6)**	**(5,7)**	**(6,7)**	**(6,8)**	**(6, 9)**	**(6,10)**
Conservative	5.03	4.86	9.26	9.16	17.6	33.6	66.4	66.7	67.1	67.6
Speculative	2.63	2.67	3.87	4.04	6.57	10.1	18.9	19.3	19.9	20.8

Benchmarks	**Meta-lock (mutex)**						**ABP**
Size	**(1,2)**	**(1,3)**	**(1,4)**	**(2,1)**	**(3,1)**	**(2,2)**	
Conservative	2.45	6.50	21.3	2.32	14.3	25.4	2.57
Speculative	2.53	6.11	32.9	3.60	19.6	34.3	2.54

(b) Space Usage (in MBs)

Fig. 6. Time and Space Comparison between Conservative and Speculative



(a) Time Usage (b) Space Usage

Fig. 7. Time and Space Comparison between Evaluation and Justification

veloped for our XMC model checking environment Model checking in XMC corresponds to evaluating a top-level query that denotes the temporal property of interest. The query succeeds whenever the system being verifed satisfies the property. To succinctly explain the success or failure of the query we use the XMC justifier. We have now implmented the speculative justifier based on *Partial-Justify* (see Section 4). This impelmentation also uses the XSB system. Both the impelmentations only share the justifications of tabled literals.

We compare the performance of both the justfiers on the model checking application using our XMC system. Figure 6(a) compares their running times while Figure 6(b) shows their space usage. The model checking examples used in these experiments ((`i-Protocol`, `ABP`,`Leader`,`Sieve`) were taken from the XMC collection. `i-Protocol` is a sliding window protocol in the GNU UUCP stack, `ABP` is the alternating protocol, `Leader` and `Sieve` are taken from the SPIN [2] example suite.

Observe that the running times of the speculative justifier is significntly better, sometimes by several orders of magnitude. Because of its significant speedups the speculative justfier is able to scale up to large problem sizes. For example, on `i-Protocol`(window size 1, no livelock) and `Leader`(size 6), which are instances of large model checking examples, the speculative justifier took a few minutes whereas the conservative justifier did not finish even after several hours!

Also observe that the space usage of the speculative justifier appears comparable to its conservative counterpart.

Figure 7(a) compares justification time of the speculative justifier to query evauation time while Figure 7(b) compares their space usage. Observe that the running times and space usage of the speculative justifier seems to suggest that they are both nearly proportional to those of query evaluation.

6 Discussion

We introduced the concept of a sepeculative justifier, presented an algoritihm for it and provided experimental evidence of its efficiency and scalabity. The jus-

tification algorithm in this paper assumed definite clause logic programs. In [8] we show how to extend the justification algorithm in a conservative justifier to normal logic programs evaluated under well-founded semantics. The same extensions carry over to the justifcation algorithms used in the speculative justifier.

In this paper our primary focus was on improving the running time of justification so as to scale to large problem sizes that we encountered in our model checking application. The justifier described in this paper can be used with any other tabled LP system. As far as space usage is concerned it is possible to improve it further. One possibility is to control the size of partial justification. Recall that partial justification can include justification of non-tabled literals. There are several reasons for controlling the justification of non-tabled literals and thereby control the size of partial justificaion. Firstly, justification of non-tabled literals can be arbitrarily big. Secondly, users may not be interested in justifying non-tabled calls. Thirdly users may prefer to use the familiar 4-port debugger for non-tabled literals over a justifier. Users can specify the non-tabled literals that they are not interested in justifying. The justifier will simply evaluate away such literals without explicitly building a justification for them. Improving space efficiency using such techniques is a topic that deserves further exploration.

References

1. 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 *Computer Aided Verification*, 2000.
2. G.J. Holzmann. The model checker SPIN. In *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
3. S. Dawson, C.R. Ramkrishnan, I.V. Ramakrishnan and T. Swift. Optimizing Clause Resolution: Beyond Unification Factoring. In *International Logic Programming Symposium*, 1995.
4. J.W. Lloyd. Declarative error diagnosis. In *New Generation Computing*, 5(2):133–154, 1987.
5. S. Mallet and M. Ducasse. Generating deductive database explanations. *Proceedings of ICLP*, 154–168, 1999
6. L. Naish, P.W. Dart, and J. Zobel. The NU-prolog debugging environment. In *ICLP*, pages 521–536, 1989.
7. G. Puebla, F. Bueno, and M. Hermenegildo. A framework for assertion-based debugging in constraint logic programming. In *Pre-proceedings of LOPSTR*, 1999.
8. Abhik Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying proofs using memo tables. In *PPDP 2000*, pages 178–189.
9. Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, and D.S. Warren. Efficient model checking using tabled resolution. In *CAV, LNCS 1254*, 1997.
10. Ehud Y. Shapiro. Algorithmic program diagnosis. In *POPL*, 1982.
11. XSB. The XSB logic programming system v2.3, 2001. Available by anonymous ftp from `www.cs.sunysb.edu/~sbprolog`.
12. Prasad Rao, C.R. Ramakrishnan, and I.V. Ramakrishnan. A Thread in Time Saves Tabling Time. In *IICSLP*, 1996.