Incremental Tabling in Support of Knowledge Representation and Reasoning

Terrance Swift

Coherent Knowledge Systems, Inc. and NOVALincs, Universidade Nova de Lisboa
(*e-mail:* terranceswift@gmail.com)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Resolution-based Knowledge Representation and Reasoning (KRR) systems, such as Flora-2, Silk or Ergo, can scale to tens or hundreds of millions of facts, while supporting reasoning that includes Hilog, inheritance, defeasibility theories, and equality theories. These systems handle the termination and complexity issues that arise from the use of these features by a heavy use of tabled resolution. In fact, such systems table by default all rules defined by users, unless they are simple facts.

Performing dynamic updates within such systems is nearly impossible unless the tables themselves can be made to react to changes. Incremental tabling as first implemented in XSB (Saha 2006) partially addressed this problem, but the implementation was limited in scope and not always easy to use. In this paper, we introduce *transparent incremental tabling* which at the semantic level supports updates in the 3-valued well-founded semantics, while guaranteeing full consistency of all tabled queries. Transparent incremental tabling also has significant performance improvements over previous implementations, including lazy recomputation, and control over the dependency structures used to determine how tables are updated.

1 Introduction

Tabled Logic Programming has supported a variety of applications that would be difficult to implement in Prolog alone, including model checking, program analysis, ontology-based deductions and decision making for collaborative agents. Typically such applications are written mainly as Prolog programs, but with a subset of the predicates tabled in order to support termination, reduce complexity, to use well-founded negation or to exploit other features.

However, systems such as Flora-2 (Yang et al. 2013) and its extensions: Silk (cf. silk.semwebcentral.org), Ergo (cf. coherentknowledge.com/publications) and the RAVE system (cf. www.sri.com/about/people/grit-denker) have been recently developed for knowledge representation and reasoning (KRR), and rely on tabled resolution for their computational underpinning. For instance, Flora-2 (Yang et al. 2013), which is based on XSB (Swift and Warren 2012), supports the non-monotonic inheritance of F-logic, prioritized defeasibility with multiple levels of conflicts, rule identifiers, function symbols, logical constraints, and HiLog. Silk and Ergo, both based on Flora-2, support all of the above features plus *omni axioms*, which are contrapositional rules whose bodies and heads are comprised of any formulas that can be supported by the Lloyd-Topor transformation (Lloyd and Topor 1984).

As an example of using these features, given the sentence: *A contractile vacuole is inactive in an isotonic environment* from (Reece et al. 2010), a tool called Linguist (www.haleyai.com) produces a Silk or Ergo formula in a mostly automatic manner (knowledge engineers may have to choose between translations in ambiguous cases), resulting in the axiom:

```
forall(?x6)^contractile(vacuole)(?x6)
        ==> forall(?x9)^isotonic(environment)(?x9)
        ==> inactive(in(?x9))(?x6);
```

Such an axiom is next translated into several Flora-2 rules about conditions of contractile vacuoles, inactive contractile vacuoles, and isotonic environments. These Flora-2 rules are then transformed to support HiLog, defeasibility and other features, resulting in numerous normal rules executed in XSB. Once a knowledge base has been constructed from axioms such as the one above, queries can be made such as: *If a Paramecium swims from a hypotonic environment to an isotonic environment, will its contractile vacuole become more active?* The translation of queries is similar to that of knowledge, but may include *hypothetical* information, e.g., that *?x* is a Paramecium swimming from a hypotonic environment to an isotonic environment. Knowledge bases themselves are built from a collection of rules and omni axioms usually written by different knowledge engineers using a shared background vocabulary. The limited coordination among knowledge engineers is critical for producing knowledge bases at a low cost.

All of the the KRR-systems mentioned above employ what may be called *pervasive tabling* where a predicate is tabled unless it is explicitly declared non-tabled. Such programs have an operational behavior that is vastly different from (tabled) Prolog. Among other matters, as many of these tables represent background knowledge, it is critical for good system performance to reuse tables between queries. However, because queries may include hypothetical knowledge, and because knowledge bases are created by interactively adding or modifying rules, good performance demands the use of *incremental tabling* (Saha and Ramakrishnan 2005; Saha 2006).

The main idea behind incremental tabling is to maintain an *Incremental Dependency Graph (IDG)*, indicating how tabled goals depend both on dynamic code and on one another. When an update is made to dynamic code, the IDG is traversed, and affected tables are updated if necessary. However, while previous versions of incremental tabling were robust enough to support a commercial application (Ramakrishnan et al. 2007), they were not sufficient to support high-level KRR applications. Most significantly, a programmer had to decide when tables were updated: either an update was forced immediately upon an assert or retract, or the programmer performed "bulk" updates, after which a command propagated the updates to all affected tables. This methodology was complicated and had semantic drawbacks: unless an update was manually invoked, there was no guarantee that tables would be updated and no provision for stronger forms of view consistency. In fact, because of the brittleness caused by the need for low-level control along with other drawbacks, previous versions of incremental tabling, (designated here as *manual incremental tabling*) were suitable only for careful use by tabling experts.

Support for pervasive tabling requires that a tabling engine be redesigned in several ways, including the mechanisms whereby tables are updated. This paper introduces *transparent incremental tabling* to support applications that rely on pervasive tabling such as the KRR-systems described above. The papers major contributions are:

- A description of core changes that allow table updates to be made in a safe and efficient manner: first, tables are updated automatically and efficiently by *lazy recomputation*; second, updates always guarantee view consistency for incremental tables.
- A description of how incremental recomputation is extended to support updates according to the three-valued well-founded semantics.
- Introduction of the notion of *IDG abstraction* to reduce the size of the IDG when necessary.
- Detailed performance analyses of transparent incremental tabling for both small program

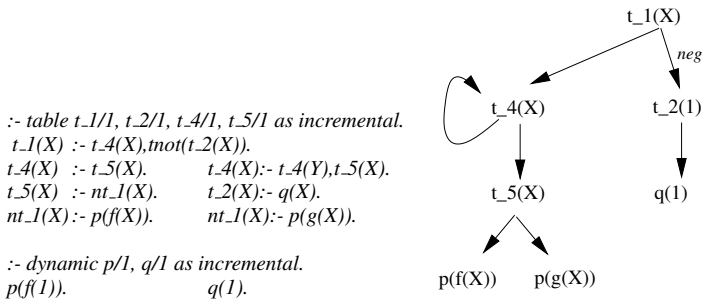

Fig. 1. A Program P_{inc}, and schematic Incremental Dependency Graph (IDG) for the query *t_1(X)*

fragments and for KRR-style examples over Extensional Databases (EDBs) up to size $\mathcal{O}(10^7)$. These results indicate that transparent incremental tabling efficiently supports the KRR uses previously mentioned, and may also provide a basis for *reactive KRR*.

Transparent incremental tabling is available in the current version of XSB. In addition to the extensions mentioned above, its implementation is based on a significant rewriting of the previous implementation of manual incremental tabling. Incremental tabling is not yet available in tabling engines other than XSB. However, while transparent incremental tabling adds data structures such as the IDG, it interfaces with a tabling engine mostly through routines for maintaining table space. Accordingly, most of the features described below are relatively portable, as tabling engines have similar table space operations, and sometimes similar data structures.

2 A Review of Manual Incremental Tabling

In this section we describe the previous version of incremental tabling using the main data structures and algorithms of (Saha 2006), which form the starting point for the features of transparent incremental tabling described in later sections. The description is as self-contained as possible, but sometimes uses the terminology of the SLG-WAM (Sagonas and Swift 1998).

Fig. 1 shows an XSB program P_{inc} where predicates are declared to use incremental tabling. In general both tables and dynamic code may be declared with various attributes: not only *incremental* as here, but also *subsumptive*, *trie-indexed*, and so on. Note that *tnot/1* is an XSB operator for tabled negation. Execution of the query *t_1(X)* creates the *Incremental Dependency Graph (IDG)* schematically shown in Fig. 1. The IDG has a node for each tabled subgoal but not for non-tabled subgoals such as *nt_1(X)* – though the bindings made by the rules for *nt_1/1* are implicitly propagated. Leaf nodes in the IDG correspond to predicates such as *p/1* and *q/1* that are declared to be both dynamic and incremental. Each downward edge in a IDG represents an element of the *direct dependency* relation; the inverse relation is the *direct affected* relation. Note that paths in the IDG may be cyclic.

At the level of data structures, each node in the IDG is represented via an *IDG node frame* (Fig. 2). For a tabled incremental subgoal *t/n*, the IDG node frame is created by the tabletry instruction, by registering it into the subgoal trie for *t/n* [1], and linking it with the *subgoal frame*, which contains information about each tabled subgoal. For dynamic incremental subgoals a new

[1] In XSB, the default data structure for tabled subgoals and their answers is based on tries (Ramakrishnan et al. 1999). While XSB offers basic support for answers that are "hash-consed" (Zhou and Have 2012) and not maintained as tries, our presentation assumes subgoal and answer tries throughout.

SLG-WAM instruction, try_dynamic_incremental performs these tasks. Each time a (tabled or dynamic) incremental subgoal S is called, the IDG may be updated. If S is new, an IDG node frame is created; also whether or not S is new, if S has a nearest tabled subgoal S_{par} as an ancestor, edges between S and S_{par} are added if not already present. As answers are derived for S, their count is maintained in the *nbr_of_answers* field of the IDG node frame.

affected_edges	Subgoals that this subgoal directly affects
dependent_edges	Subgoals upon which this subgoal directly depends
subgoal_frame	Pointer back to the subgoal frame
nbr_of_answers	Counts the number of answers rederived
previous_IDG_node	Used to determine if re-evaluation has changed the set of answers
new_answer	set to true if a new answer has been derived
falsecount	determines whether subgoal is valid

Fig. 2. The IDG node frame for incremental tables

At a high level, the use of the IDG is easy to understand. If a fact, say *p(g(2))*, is asserted, the incremental update subsystem must call traverse_affected_nodes() (Fig. 3) to traverse the IDG. Separate traversals start from each leaf node with which *p(g(2))* unifies, and the traversals will increment the *falsecount* field of their IDG node frame (cf. Fig. 2), marking them as *invalid* (i.e., having a *falsecount* greater than 0). As it is unclear whether sensible semantics can be given to updating a subgoal that is incomplete (i.e., that is still being computed), a permission error is thrown if this is attempted. In our running example, assuming that no nodes in the IDG are already invalid, the algorithm will traverse depth-first through all nodes affected by *p(g(X))* (directly or indirectly). In so doing, the affected non-leaf nodes are added to a global *invalid list* in the same order. In our example, the nodes for *t_5(X)*, *t_4(X)* and *t_1(X)* are traversed, and the invalid list represents this sequence.

Several properties of the traversal are worth noting. First, use of the *falsecount* field in traverse_affected_nodes() prevents the same node from being traversed multiple times. Also, note that invalidation simply represents *some* change in the underlying data so that retracts are handled in the same manner as asserts, and both positive and negative dependencies are treated in the same way. In fact, since the traversal starts with dependency leaf nodes that unify with a given atom, propagation of a rule update is handled in the same manner as a fact update: traverse_affected_nodes() is invoked for leaf nodes that unify with the rule head. In either case, the unification of leaf nodes with a given atom can also prevent unnecessary updates: for instance, if the fact *q(g(2))* were added, it would not cause any update, since no leaf node of the IDG unifies with this fact.

After the invalidation phase is finished, reevaluation of the affected nodes may be done either immediately, or at a later time through an explicit command. Note that once the invalid list has been set up, the affected tables can be updated in a bottom-up manner simply by removing them in order from the list. Specifically, for each IDG node *IDGN* removed from the invalid list, incremental_reeval(*IDGN*) called (Fig. 3). If $IDGN.falsecount$ is 0, the subgoal does not need to be recomputed. Otherwise, the answers for T, the table associated with *IDGN*, are marked as deleted, although their space is not reclaimed [2]. A new IDG node $IDGN_{new}$ is created for T, and its *previous_IDG_node* field is set to the old IDG node, *IDGN* (cf. Fig. 2). The subgoal

[2] The *answer list* of an answer trie, which allows easy traversal of all answers in the trie, is reclaimed at the completion of each non-incremental table, but retained by incremental tables for traversals during re-evaluation.

```
traverse_affected_nodes(IDG node frame IDGN)
    /* IDGN is the IDG node frame for an incrementally tabled predicate */
    If the table associated with IDGN is not completed, throw a permission exception
    For each IDGN_aff that is directly affected by IDGN
        IDGN_aff.falsecount++;
        If (IDGN_aff.falsecount == 1) traverse_affected_nodes(IDGN_aff)
        Add IDGN to the global invalid list.

incremental_reeval(IDG node frame IDGN)          /* S is the subgoal to be recomputed */
    If IDGN.falsecount > 0
        Let S_T be the subgoal frame associated with IDGN (i.e., S_T = IDGN.subgoal_frame)
        For each A in S_T.answer_list
            Mark A as deleted, but do not adjust answer trie choice points or reclaim space
        Create a new IDG node IDGN_new for S_T
        IDGN_new.new_answer := false; IDGN_new.falsecount = IDGN_new.nbr_of_answers = 0
        Call S and for each new derived answer A_deriv
            Increment IDGN_new.nbr_of_answers
            If A_deriv was marked as deleted, remove the deletion mark
            Else IDGN_new.new_answer = true
        After completion of S, for each A in S_T.answer_list
            If A is still marked as deleted, remove A from S_T.answer_list
                Reset answer trie choice points and reclaim space for A
        If IDGN_new.new_answer = false and IDGN_new.nbr_of_answers = IDGN.nbr_of_answers
            propagate_validity(IDG node frame IDGN)

propagate_validity(IDG node frame IDGN)
        For each IDGN_aff that is directly affected by IDGN
            IDGN_aff.falsecount- -
            if IDGN_aff.falsecount == 0 propagate_validity(IDGN_aff)
```

Fig. 3. Schematic algorithms for manual incremental tabling

for T is re-evaluated, and for each answer A, $IDGN_{new}.nbr\_of\_answers$ is incremented; in addition if A is new, (i.e., the addition of the answer A does not undelete a previously obtained answer) $IDGN_{new}.new\_answer$ is incremented. Clearly, if $IDGN.nbr\_of\_answers$ is not equal to $IDGN_{new}.nbr\_of\_answers$, the answers for T have changed; also if the two numbers are the same but $IDGN_{new}.new\_answer$ is set, the answers for T have changed. Otherwise, the answers for T have not changed, and the subgoals T affects are traversed to decrement their $falsecount$ fields, which may transitively prevent other subgoals from having to be recomputed (cf. propagate_validity() in Fig. 3).

3 Ensuring Transparency through Lazy Recomputation and View Consistency

Perhaps the main drawback of manual incremental tabling is the level of control it requires from a programmer. A programmer can specify that an incremental update is to be done immediately after an assert or retract, but this is inefficient when multiple updates are required. Alternatively, a programmer can specify that an assert or retract simply invalidate affected subgoals, but later must make a call to reevaluate subgoals on the invalid list. In either case, if choice points exist to an incrementally tabled subgoal S that is completed, the semantics of an update are undefined (and in fact the program may crash). In addition to these issues, manual incremental tabling may

cause unnecessary work as all affected goals are recomputed even if they are never re-queried. We show how these problems are fixed in transparent incremental tabling.

3.1 Lazy Recomputation

In lazy recomputation assert and retract hooks invalidate tables when a change is made to a dynamic incremental predicate. However, an invalid subgoal S is not re-evaluated until it is called, at which time incremental tabled subgoals upon which S depends are also re-evaluated. The algorithm for lazy recomputation is shown in Fig. 4 within a schematic description of the SLG-WAM's tabletry instruction, which is executed upon calling a tabled subgoal [3]. Specifically, if S is completed and invalid, lazy recomputation is handled within lines 12-17, using a *reeval_ready* field, which transparent incremental tabling adds to each IDG node frame. If the *reeval_ready* field for S is set to *compute_dependencies_first* the IDG nodes upon which S depends are traversed in a depth-first manner by traverse_dependent_nodes() and the traversed subgoals are added to the invalid list (Fig. 4). This predicate, analogous to traverse_affected_nodes() of Fig. 3, traverses dependency edges rather than affected edges. Once the invalid list is constructed, its subgoals are recomputed by recompute_dependent_tables() which iteratively calls the version of incremental_reeval() in Fig. 5. By default the *reeval_ready* field is set to *compute_dependencies_first*, but when traverse_dependent_nodes() adds a subgoal S' to the invalid list the *reeval_ready* field for S' is set to *compute_directly* so that the next call to S' will not add it again. Later, the *reeval_ready* field is reset to *compute_dependencies_first* in incremental_reeval() after its associated goal is re-evaluated (Fig. 5, line 22); or it is reset when the IDG node frame's *falsecount* is set to 0 by propagate_validity() (this change to Fig. 3 is not shown).

The implementation of line 15 of Fig. 4 uses a general interrupt mechanism whereby a given goal G may dynamically interrupt the current execution environment Env so that G is immediately executed and success and failure continuations of G are (a modification of) Env[4]. In line 17, the interrupt mechanism intersperses a call to recompute_dependent_tables(), to traverse the invalid list and recompute subgoals. When recompute_dependent_tables() finishes, its continuation will make a fresh call S, which will see a completed and valid table, and will then simply backtrack through answers for S (starting with line 18 of Fig. 4).

3.2 View Consistency

A fundamental principle of databases is to support view consistency: that is, to ensure that answers to a query Q should be those derivable at the time Q was begun, and should not be affected by any updates. Accordingly, the ISO standard for Prolog (ISO working group JTC1/SC22 1995) specifies that an update v to dynamic code should not affect the behavior of choice points that were created before v. Extending view consistency to incremental tables is critical for understandable system behavior, especially when KRR features such as hypothetical reasoning must be supported. Because XSB's incremental tabling does not allow updates that affect tables that are still being computed (Section 2), supporting view consistency effectively means ensuring consistency for choice points into completed tables. As such choice points correspond to database cursors, we term them *Open Cursor Choice Points, (OCCPs)*.

[3] XSB's tabletry instruction is substantially more complex as it supports call subsumption, subgoal abstraction, multi-threaded tabling and other features.

[4] In XSB, as in other Prologs, such interrupts are used to handle unification of attributed variables, signaling among Prolog threads, and other tasks.

```
Instruction tabletry  /* SLG-WAM instruction for calling a tabled subgoal S*/
    Check whether there is a table for a variant of S and make a table for S if not
    If S is incremental create an IDG node frame
       If S has a nearest tabled ancestor S_anc add IDG edges between S and S_anc if not present
5   If there was not a table for S
       Create a subgoal frame for S
       Create a generator choice point to produce answers via program clause resolution
    Else if S is incomplete      /* all answers for S may not yet have been derived */
       Create a consumer choice point to perform answer resolution
10  Else, if S is completed      /* all answers for S have been derived */
       Set up a consumer choice point to perform answer resolution
       If S is incremental and invalid
             If S.IDG_node.reeval_ready == compute_dependencies_first
                invalid list = traverse_dependent nodes(S.IDG_node.invalid)
15           Interrupt to call recompute_dependent_tables with continuation S
             Else                 /* S.IDG_node.reeval_ready == compute_directly */
                incremental_reeval(S.subgoal_frame)
       Branch to the instruction of the root of the answer trie for S

traverse_dependent_nodes(IDG node frame IDGN)
    For each IDGN_dep upon which IDGN directly depends
       If (IDGN_dep.reeval_ready == compute_dependencies_first)
          Add IDGN_dep to the global invalid list.
          traverse_dependent_nodes(IDGN_dep)
```

Fig. 4. Schematic pseudo-code for lazy recomputation

The approach to view consistency adopted by transparent incremental tabling is summarized in this section, with further details provided in Appendix A. A main goal is to avoid overhead when there are no choice points whose "view" needs to be maintained (including those of non-incremental tables). For this purpose, an *occp_num* field is maintained in the subgoal frame of a completed incremental table T to indicate whether there are OCCPs for T (Appendix A.1). *occp_num* is incremented when the subgoal for T is called; and decremented when the last answer for T has been returned to the call, or when a cut or throw removes the call from the choice point stack. Only if *occp_num*> 0 must the OCCP's view be preserved. Transparent incremental tabling performs this preservation during incremental_reeval() by calling preserve_occp_views() (Fig. 5, line 4). While preserve_occp_views() is fully described in Appendix A.2, its main actions are as follows. The choice point stack is traversed, and for each OCCP CP_T for T, the answer substitutions that have not yet been resolved by CP_T are determined and then copied from T into the heap as a list (making sure that their heap space is frozen so they are not lost upon backtracking). For each answer substitution that corresponds to a answer whose truth value is *undefined*, the copying includes a special marker *undef*. Next, the structure of CP_T is altered, and its instruction is modified to backtrack through the list on the heap rather than through the table. Once preserve_occp_views() has executed, incremental_reeval() proceeds as it would otherwise do. Later, when the modified version of CP_T is backtracked into, a new instruction, preservedViewMember is called to return the answer substitutions for the preserved view (Appendix A.3) using the correct truth value. When the answers in the list have been exhausted, the heap space used for the list is unfrozen if it is safe to do so.

4 Supporting Well-Founded Negation

A necessary extension for incremental tabling to support KRR applications of the type mentioned in the introduction is to support full well-founded negation. KRR applications make use of the *undefined* truth value to represent conflicts in the defeasibility theory used by a program, as well

as to handle infinite models through a type of answer abstraction called *restraint* (Grosof and Swift 2013), and to support debugging of KRR programs (cf. (Naish 2006)).

As mentioned in the previous section, the IDG maintains information about the dependency and affected relations without representing whether these changes are positive or negative. One advantage of this is that manual incremental tabling is correct for stratified negation — here, meaning well-founded negation with two-valued models. However, to support full well-founded negation, the update process must handle tables in which some atoms are *undefined*. To explain how this is done, we overview those aspects of well-founded negation in SLG resolution (Chen and Warren 1996) that are relevant to the incremental update algorithms.

Essentially, a query evaluation by SLG resolution builds up a partial model of those parts of a program P that are relevant to the query. To make this specific, an SLG evaluation $\mathcal{E}$ is modeled as a sequence of states, called *forests*. Let $\mathcal{F}$ be one such forest in $\mathcal{E}$. $\mathcal{F}$ contains a set of tabled subgoals that have been encountered so far in $\mathcal{E}$. Each such tabled subgoal S in $\mathcal{F}$ is associated with a table T_S containing computed answers for S; T_S may be marked as *completed* in $\mathcal{F}$ if it has been determined that all necessary resolution has been performed to derive answers for S. To support 3-valued interpretations of $\mathcal{F}$, answers are distinguished as *unconditional* answers representing true derivations, and *conditional* answers representing derivations of atoms with truth value *undefined*. Accordingly, let T_S be a completed table for a subgoal S in $\mathcal{F}$, and let S_G be an atom in the ground instantiation of S. S_G is true if it is in the ground instantiation of some unconditional answer in T_S, and is false if is *not* in the ground instantiation of *any* answer in T_S (conditional or unconditional).

Formally, for a subgoal S, a conditional answer has the form $S\theta$:-$DL|$ where $S\theta$ is termed the *answer substitution*; and DL, the *delay list*, is a list of literals needed to prove $S\theta$ but whose resolution has been delayed because they do not have a well-founded derivation (based on the current state of the evaluation if T_S is not completed). During the course of an evaluation, if a literal L in a delay list becomes *true* or *false*, the SLG SIMPLIFICATION operation respectively removes L from the delay list or indicates that the conditional answer itself is false.

Example 4.1

The goal *p(X)* to the program

p(1) *p(2):- not q(2)* *p(2):- not q(3)* *q(X):- not p(X)*

has an unconditional answer *p(1)* along with two conditional answer: *p(2):- not q(2)|*, *p(2):- not q(3)|*. Note that the delay lists for answers to *p(2)* contain only *undefined* literals upon which *p(2) directly* depends (e.g., *not q(2), not q(3)*), but not indirect dependencies such as *not p(3)*. □

As mentioned above, XSB represents both tabled subgoals and their answers using tries, a representation that is supported by other Prologs such as YAP (Santos Costa et al. 2012) and Ciao (Hermenegildo et al. 2012). In XSB this representation is extended as follows (Sagonas et al. 2000). If an atom A_{cond} is *undefined*, the leaf node of the answer representing A_{cond} points to an *answer information frame* which in turn points to other answers conditional on A_{cond} as well to a *delay trie* representing all delay lists upon which A_{cond} is conditional. In Example 4.1 the delay trie for *p(2)* would contain the lists *[not q(2)]* and *[not q(3)]*. Whenever an unconditional answer $S\theta$ is derived in a table for subgoal S, the answer information frame and delay trie for conditional answers to $S\theta$ are deallocated if they exist.

To extend incremental recomputation to correctly handle changes involving conditional answers, several previously unconsidered cases must be addressed for a given answer substitution $S\theta$ in a table S. Each case below considers only those answers in the table S.

- *Informational Weakening 1,* There were previously no answers for $S\theta$; after the update there are one or more conditional answers for $S\theta$.
- *Informational Weakening 2,* There was previously an unconditional answer for $S\theta$; after the update there are one or more conditional answers for $S\theta$.
- *No Informational Change,* There were previously one or more conditional answers for $S\theta$; after the update further conditional answers $S\theta$ were added, or some but not all conditional answers for $S\theta$ were deleted.
- *Informational Strengthening 1,* There were previously one or more conditional answers for $S\theta$; after the update $S\theta$ becomes *true*, with an unconditional answer.
- *Informational Strengthening 2,* There were previously one or more conditional answers for $S\theta$; after the update $S\theta$ becomes *false*, with no answers.

The cases above are grouped by their action on the information ordering of truth values, where both *true* and *false* are stronger than *undefined*. From the perspective of table updates, no action need be taken in the case of *No Informational Change*, as the truth value of $S\theta$ is unchanged. To see this, recall that delay lists contain only direct dependencies. Thus any answer A' that is conditional on $S\theta$ will contain $S\theta$ or $not\ S\theta$ in its delay lists so that changes to the delay list of $S\theta$ need not be propagated. Strengthening and weakening of answers are addressed by the extensions to incremental_reeval() shown underlined in Fig. 5.

```
incremental_reeval(IDG node frame IDGN)                    /* S is the subgoal to be re-computed */
    If IDGN.falsecount > 0
        Let S_T be the subgoal frame associated with IDGN (i.e., S_T = IDGN.subgoal_frame)
        if S_T.occp_num > 0 preserve_occp_views(S_T)       /* Ensure view consistency: see Section 3.2 */
5       For each answer Sθ in S_T.answer_list
            Sθ.deleted = true
            If Sθ is unconditional Sθ.unconditional = true else Sθ.unconditional = false
        Create a new IDG node IDGN_new for S_T
        IDGN_new.new_answer := false; IDGN_new.falsecount = IDGN_new.nbr_of_answers = 0
10      Call S and for each new derived answer Sθ:-DL|
            If Sθ.deleted == true
                Sθ.deleted = false; IDGN_new.nbr_of_answers++
            If Sθ.unconditional == false but Sθ is now unconditional
                Sθ.unconditional = true; invoke simplification
15          Else /* Sθ.deleted was false */ IDGN_new.new_answer = true
        After completion of S, for each Sθ in S_T.answer_list
            If Sθ.deleted == true, remove Sθ from S_T.answer_list
                If Sθ.unconditional = false invoke simplification
                Adjust trie choice points and reclaim space for Sθ
20          Else if Sθ.unconditional == true, and Sθ is now conditional
                IDGN_new.new_answer = true
        IDGN.reeval_ready = compute_dependencies_first
        If IDGN_new.new_answer == false and IDGN_new.nbr_of_answers = IDGN.nbr_of_answers
            propagate_validity(IDGN)
```

Fig. 5. Schematic algorithm for updates in transparent incremental tabling

As shown in Fig. 5 setup for the re-derivation of S now also sets a new *unconditional* field of an answer, representing whether the answer was unconditional at the start of the re-derivation (line 7). In the re-derivation, $IDGN_{new}.nbr\_of\_answers$ is incremented whenever a new answer substitution $S\theta$ is encountered, whether $S\theta$ is conditional or unconditional (Fig. 5 lines 11-12), so that $IDGN_{new}.nbr\_of\_answers$ will be updated at most once regardless of how many conditional answers exist for $S\theta$. Thus, there are no changes required for *Informational Weakening 1* as the addition of new conditional and unconditional answer substitutions is handled in the

same manner. Also, if more than one conditional answer is derived for $S\theta$, only the first will increment *IDGN*$_{new}$.$nbr\_of\_answers$, in effect handling the case of *No Informational Change*. A similar check of $S\theta.unconditional$ during re-derivation (lines 13-14) handles *Informational Strengthening 1*, the case where $S\theta$ had only conditional answers, but is now unconditional. This case can actually be handled directly by SLG simplification and does not require propagation through the incremental update system. Once S has been rederived, its answer list is traversed as before (cf. Fig. 3). During this traversal, line 18 handles the case of *Informational Strengthening 2* where $S\theta$ had been conditional but is now false and uses simplification; Lines 20-21 handle *Informational Weakening 2* where $S\theta$ had been *false* but now is *undefined*.

Fig. 5 reflects a bilattice of the information ordering and the truth ordering (where $true >$ *undefined*>*false*). As discussed, SLG simplification propagates changes of an answer's truth value when it is informationally strengthened. Changes to the truth value of $S\theta$ that reflect a strengthening in the truth ordering can be detected during re-derivation (*Information Weakening 1*, *Information Strengthening 1*). Changes that reflect a weakening in the truth ordering must wait until the re-derivation is complete (*Information Weakening 2*, *Information Strengthening 2*).

5 Abstracting the IDG

The IDG is clearly essential to efficiently update incremental tables, but in certain situations constructing the IDG can cause non-trivial overheads in query time and table space. These overheads can be addressed in many cases by *abstracting* the IDG. When a tabled subgoal S is called, rather than creating an edge between S and its nearest tabled ancestor S' (if any), one could abstract S, S' or both. The semantics and implementation of subgoal abstraction was defined in (Riguzzi and Swift 2013), here we appeal to an intuitive notion of *depth abstraction*: given a subgoal S and integer k, subterms of S with depth $k+1$ are replaced by unique new variables. For instance, in Fig. 1, abstracting *q(f(1))* at level 1 gives *q(f(*X_1*))*; abstracting at level 0 gives *q(*X_1*)*.

Fig. 6 illustrates an important case where abstracting the IDG can be critical to good performance for incremental tabling. In the case of left-linear recursion, if no abstraction is used a new node will be created for each call to *edge/2* as shown on the left side of this figure. If a large number of data elements are in fact reachable, the size of the IDG can be very large. If calls to the *edge/2* predicate make use of depth-0 abstraction, the graph may be much smaller as seen on the right side of Fig. 6. Whether abstracting a IDG in this manner is useful or not is application dependent; however, performance results in the next section illustrate cases where abstraction greatly reduces both query time and space.

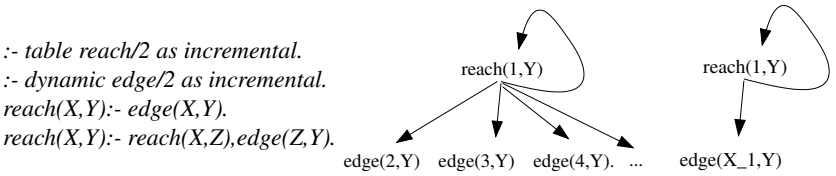

Fig. 6. A left-linear program and schematic IDGs: Left without IDG abstraction; Right: with IDG abstraction

Abstracting the *edge/2* predicate has subtle differences from abstracting tabled subgoals. In the first place, the *edge/2* predicate of Fig. 6 is not tabled. Furthermore, the actual *edge/2* subgoal itself should not be abstracted to depth 0 since losing the first argument instantiation would prevent

the use of indexing. Rather, only the IDG's representation of the subgoal should be abstracted. Fortunately, in XSB the code to intern dynamic goals for the IDG shares code used for tabling, so that extending abstraction to handle dynamic incremental predicates is relatively straightforward. In XSB, abstraction of dynamic code for the IDG can be specified via the declaration:
:- dynamic edge/2 as incremental, abstract(0).

6 Performance Results and Analysis

The performance of manual incremental tabling in XSB has been analyzed previously, most extensively in (Saha 2006). By and large the behavior of manual incremental tabling features are not affected by the rewriting to support transparent incremental tabling. Accordingly, the performance questions addressed here analyze new features, scalability, and the behavior of incremental tabling for KRR-style computations. A summary of performance results is given in this section, with tables and other details provided in Appendix B.

Left-Linear Recursion Recursion is heavily used in KRR-style programs that make use of features such as Hilog or defeasibility. As a first test, queries of the form $reach(\langle free\rangle, \langle free\rangle)$ were made to a left recursive predicate (Fig. 6) with and without IDG abstraction on the *edge/2* predicate (cf. Appendix B.1). In the benchmarks, *edge/2* consists of ground facts representing randomly generated graphs of 50,000 – 5,000,000 edges. As shown in Fig. B.1 if IDG abstraction is not used, creating the IDG adds a CPU time overhead of roughly 50% and a table space overhead of about 300% compared to non-incremental tabling. By using IDG abstraction at depth 0, the table space overhead becomes approximately 30%, and the time overhead 5-10%. Fig. B.2 shows that for a batch updates (0.02%-2% of EDB), the overhead of re-evaluation is negligible, particularly if abstraction is used.

Non-Stratified Linear Left Recursion Similar tests were made using the predicate *ureach/2* (Fig. B.3), constructed to perform transitive closure, but producing answers with truth value *undefined*. The query $ureach(\langle free\rangle, \langle free\rangle)$ was evaluated on a graph of 500,000 edges. Overhead results for the initial query (Fig. B.4) are similar to those for $reach(\langle free\rangle, \langle free\rangle)$ in terms of time; however the space overhead for incremental tabling is proportionally less (around 10-15% with IDG abstraction) as storing the conditional answers used in this test imposes its own space overhead. Fig. B.5 shows the time to perform various inserts that cause new answers to be added to the table for $ureach(\langle free\rangle, \langle free\rangle)$, and that also change the truth value of some known answers from *undefined* to *true* as discussed in Section 4. The figure shows that updating conditional answers imposes essentially no overhead compared to unconditional answers.

Performance Analysis on a Program with KRR Features The program in Fig. B.6 represents a social network in which certain members of a population are at risk, and other members of the population may influence the behavior of the at-risk members. While the program contains stratified negation, its main computational challenge arises from its heavy use of equality between constants and functional terms – a reasoning capability similar in flavor to some description logics. This use of equality over functional terms quickly leads to non-termination and unsafe negative subgoals during query evaluation. As discussed in Appendix B.2, these behaviors are handled using various tabling mechanisms, so that the ability to incrementally maintain tables for queries to this program requires the ability to update three-valued models that arise from answer abstraction (Grosof and Swift 2013), tabled negation and subgoal abstraction.

As a first benchmarkof this program, a small EDB of about 10,000 facts was generated, and *good_influence*$(\langle bound\rangle, \langle free\rangle)$ was queried for 200 randomly chosen values for its first argument. In this case, incremental tabling caused a time overhead of about 240% and a space

overhead of 280% – although further exploration of abstraction would likely reduce these numbers. Next, updates of substantial sizes were performed on the EDB, and the revaluation time was computed (Fig. B.7). While most of these times are near the level of noise, recomputation of several of the predicates timed out. Analysis of these timeouts showed that they arose because the additional facts caused a large number of new tables to be created when the 200 queries were re-evaluated. Fig. B.8 shows the times to assert or retract, plus time to invalidate affected subgoals via traverse_affected_nodes(). Except for updates to *parent_of_edb/2* which directly affect equality, invalidation did not take a significant amount of time

Scalability Analysis on a Program with KRR Features As a next step, the computational burden of the equality relation in the previously mentioned program was reduced by specializing it as discussed in Appendix B.2.1. Tests were performed on EDBs from around 100,000 – 10,000,000 facts. As shown in Fig. B.9, the space and time for these computations scales roughly linearly. For the EDB of about 10,000,000 facts, times were obtained for various large batch updates and for query re-evaluation (Figs. B.10 and B.11). Except for updates to *parent_of_edb/2*, re-evaluation time was very low compared to initial query time (even for initial queries using non incremental tabling), illustrating the promise of incremental tabling for large, reactive systems. These benchmarks also demonstrate the scalability of this implementation, even for very large IDGs. In Fig. B.9 the IDG contained over 750 million edges; after the update sequences mentioned above were applied, it contained more than 1 billion edges.

7 Discussion

This paper has introduced transparent incremental tabling, which improves previous versions of incremental tabling in both semantics and efficiency. The semantics of lazy recomputation (Section 3) together with the preservation of view consistency (Section 3.2 and Appendix A) guarantee that incremental tables will *always* reflect the state of the underlying knowledge base at the time they were queried. This view consistency takes tabled logic programming a step closer to deductive databases, and supports hypothetical reasoning in KRR applications. In addition, the ability to update 3-valued computations (Section 4) is necessary when defeasibility is used over the well-founded semantics, as well as for other features such as answer abstraction. In terms of efficiency, lazy recomputation avoids recomputing invalidated queries until they are requeried, and IDG abstraction (Section 5) can significantly reduce the amount of time and space required for queries. The efficiency and scalability of the resulting implementation was summarized in Section 6 and discussed in detail in Appendix B. Appendix C provides further information about how to use transparent incremental tabling in practice.

Although the major semantic issues for incremental tabling have been addressed in this paper, KRR-style computations incur a heavy computational burden, and the benchmark programs do show cases where transparent incremental tabling incurs more cost than is desirable. An important goal is to "guarantee" bounds for transparent incremental tabling when used on representative KRR programs. For instance, for constant bounds b_i, initial query time using incremental tabling should never be more than b_i times that of non-incremental tabling; recomputation time should never be significantly more than initial query time; and the space for the IDG should never be more than b_j times the space of the tables themselves. Such bounds may be obtained through a mixture of program analysis (some of which may itself be incremental cf. (Hermenegildo et al. 2000)) and adaptive incremental tabling algorithms. Even today, incremental tabling is starting to be used to prototype applications in stream deductive databases and event monitoring; continued efficiency improvements should make commercial applications in these areas possible.

References

CHEN, W. AND WARREN, D. S. 1996. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM 43,* 1, 20–74.

GROSOF, B. AND SWIFT, T. 2013. Radial restraint: A semantically clean approach to bounded rationality for logic programs. In *Conference of the American Association for Artificial Intelligence*.

HERMENEGILDO, M., PUEBLA, G., MARRIOTT, K., AND STUCKEY, P. 2000. Incremental Analysis of Constraint Logic Programs. *ACM TOPLAS 22,* 2 (March), 187–223.

HERMENEGILDO, M. V., BUENO, F., CARRO, M., LÓPEZ-GARCIÁ, P., MERA, E., MORALES, F., AND PUEBLA, G. 2012. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming 12,* 1-2, 219–252.

ISO WORKING GROUP JTC1/SC22. 1995. Prolog international standard ISO-IEC 13211-1. Tech. rep., International Standards Organization.

LLOYD, J. AND TOPOR, R. 1984. Making Prolog more expressive. *Journal of Logic Programming 1,* 3, 225–240.

NAISH, L. 2006. A three-valued semantics for logic programmers. *Theory and Practice of Logic Programming 6,* 5, 509–538.

RAMAKRISHNAN, C., RAMAKRISHNAN, I., AND WARREN, D. S. 2007. XcelLog: A deductive spreadsheet system. *Knowledge Engineering Review 22,* 3, 269–279.

RAMAKRISHNAN, I. V., RAO, P., SAGONAS, K., SWIFT, T., AND WARREN, D. S. 1999. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming 38,* 1, 31–55.

REECE, J., URRY, L., CAIN, M., WASSERMAN, S., MINORSKY, P., AND JACKSON, R. 2010. *Campbell Biology*. B. Cummings. 9th Edition.

RIGUZZI, F. AND SWIFT, T. 2013. Well-definedness and efficient inference for probabilistic logic programming under the distribution semantics. *Theory and Practice of Logic Programming 13,* 2, 279–302.

SAGONAS, K. AND SWIFT, T. 1998. An abstract machine for tabled execution of fixed-order stratified logic programs. *ACM TOPLAS 20,* 3 (May), 586 – 635.

SAGONAS, K., SWIFT, T., AND WARREN, D. S. 2000. An abstract machine for efficiently computing queries to well-founded models. *Journal of Logic Programming 45,* 1-3, 1–41.

SAHA, D. 2006. Incremental evaluation of tabled logic programs. Ph.D. thesis, SUNY Stony Brook.

SAHA, D. AND RAMAKRISHNAN, C. 2005. Incemental and demand-driven points-to analysis using logic programming. In *Principles and Practice of Declarative Programming*. 117–128.

SANTOS COSTA, V., DAMAS, L., AND ROCHA, R. 2012. The YAP prolog system. *Theory and Practice of Logic Programming 12,* 1-2, 5–34.

SWIFT, T. AND WARREN, D. 2012. XSB: Extending the power of Prolog using tabling. *Theory and Practice of Logic Programming 12,* 1-2, 157–187.

YANG, G., KIFER, M., WAN, H., AND ZHAO, C. 2013. *FLORA-2: User's Manual Version 0.99.3*. http://flora.sourceforge.net.

ZHOU, N. AND HAVE, C. 2012. Efficient tabling of structured data with enhanced hash-consing. *Theory and Practice of Logic Programming 12,* 4-5, 547–563.