Appendix for the paper of 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

Acknowledgements

The research in this paper was partially funded by Vulcan, Inc. and Coherent Knowledge Systems and FCT Project ERRO PTDC/ EIACCO/121823/2010. The author would like to thank Paulo Moura for latex-related help, Fabrizio Riguzzi for making the University of Ferrara server available for benchmarks, and anonymous reviewers for their careful comments. Finally, the author would like to thank Michael Kifer for finding and reporting many, many bugs in transparent incremental tabling.

Appendix A View Consistency and Table Updates

As discussed in Section 3.2 the approach to maintaining view consistency for transparent incremental tabling has three main parts. (1) a count of the OCCPs for an incremental table T is always maintained. (2) when an update affects an incremental table, the view of an OCCP is preserved by copying its unconsumed answers onto the heap and altering the OCCP to use the copied answers. (3) a new instruction returns answers from the preserved views upon backtracking.

More than other aspects of transparent incremental tabling, the details of view consistency support rely on tabling data structures and algorithms used by XSB, some background for which is presented here.

- *Answer Tries* Steps 1 and 2 use the sequence of choice points set up when backtracking through an answer trie, the default data structure used by XSB to represent answers (Ramakrishnan et al. 1999) Answer tries are constructed to support *substitution factoring*, so that they contain only the information used to bind variables in the associated subgoals, i.e., the answer substitution introduced in Section 4. Each trie node contains an SLG-WAM instruction, so that returning an answer substitution directly corresponds to traversing a path from the root of a trie to a leaf, and backtracking through the trie corresponds to traversing the trie in a fixed depth-first order. A choice point is created whenever traversing a new node that has multiple children and is removed when all children have been traversed (through a trust-style instruction).
- *Freeze Registers* Steps 2 and 3 make use of the SLG-WAM's **HF** (heap freeze) register, which is used to protect terms in the heap from being over-written when tabled computations are repeatedly suspended and resumed.

While these data structures are not unique to XSB, other engines that differ from XSB in their representation of answers or in their implementation of suspension and resumption may implement this approach with suitable modifications.

A.1 Maintaining a Count of OCCPs for a Completed Incremental Table

To maintain a count of OCCPs, a field called *occp_num* is added to subgoal frames. In addition, the first choice point created in backtracking through an answer trie, CP_{first} is modified so that it increases the OCCP number when CP_{first} is created, and decreases the number when CP_{first} is removed. Finally, any routines that remove choice points must also be modified to reset the *occp_num*, including code that removes choice points upon executing a cut, and when executing a throw operation.

A.2 Preserving Views and Altering OCCPs

In order to preserve the views of the current OCCPs for a table T, incremental_reeval() of Fig. **??** is modified to check whether the *occp_num* in the subgoal frame S_T of T is non-zero. If so, preserve_occp_views() is called using S_T (described at a highly schematic level in Fig. A 1). This routine traverses the choice point stack from top downwards until all OCCPs for T have been located[1]. When a choice point CP is encountered whose failure continuation points to (the instruction field of) a node in the answer trie for T, the process begins of copying the answers that have not yet been consumed by CP. First, an associated choice point CP_{root} must be found. The process of backtracking through an answer trie can create a series of trie choice points of which CP will be the last in the segment due to the order of the choice point stack traversal. However this series of choice points will always form a connected segment in the choice point stack, so that finding the first choice point of the series CP_{root}, is relatively simple. Next, using CP and CP_{root}, the unconsumed portion of the answer trie for T is traversed; each time the traversal encounters a leaf, a pointer to the leaf is added to a list, $Unconsumed$ [2]. Next, a $prservedList$ is constructed on the heap, by traversing the elements of $Unconsumed$. Each element of the $prservedList$ contains a binary term $ret_2(Substitution, Condition)$. $Substitution$ represents a given answer substitution consisting of $AnsSubstSize$ terms, one for each distinct variable in the associated subgoal. It is repreented as a term $ret_{AnswerSubstSize}(Args)$ where each argument corresponds to an element of the answer substitution. $Condition$ is $null$ for unconditional answers, and points to a special answer *undef* whose truth value is *undefined*, and whose use is explained below.

Once the $preservedList$ is constructed, the choice points between CP and CP_{root} inclusive are coalesced via coalesce_choice_points() into a new choice point, $CP_{coalesced}$. This routine is easiest to illustrate by its results (Fig. A 2). The address of $CP_{coalesced}$ is that of CP, but when $CP_{coalesced}$ is backtracked into, it will restore the engine environment as it would be if

[1] Unlike some other Prologs, XSB has a choice point stack separate from the local stack. The traversal of the choice point stack uses the *previous_top* field of choice points; this field was not part of the SLG-WAM design presented in (Sagonas and Swift 1998), but was added to support various forms of garbage collection.

[2] As mentioned previously, (e.g., Section **??**) an answer list is preserved for incremental tables. While this answer list contains a pointer to each leaf of an answer trie, its ordering does not correspond to the traversal needed to obtain the unconsumed answers of an OCCP.

preserve_occp_views(subgoal_frame S_T)

Traverse the choice point stack from top until $S_T.occp\_num$ OCCPs have been located
For each choice point CP in the choice point stack
If $CP.failure\_continuation$ points into the answer trie for S_T
Determine the root choice point, CP_{root}, for CP
Construct a list of pointers, $Unconsumed$, to leaves of unconsumed answers
$preservedList$ = copy_answer_substitutions_to_heap($Unconsumed$,$CP_{root}.AnsSubstSize$)
coalesce_choice_points(CP,CP_{root},$preservedList$)
If (**HF** reg $\neq$ bottom of heap) $\mathbf{HF}reg = \mathbf{H}reg$
$S_T.occp\_num$ = 0

copy_answer_substitutions_to_heap(List_of_trie_leaves $Unconsumed$,int $AnsSubstSize$)

For each $leaf\_ptr$ in $Unconsumed$
Create a list element with the following information
Let Ans_{heap} be a skeleton with argument $ret_{AnsSubstsize}$ and $AnsSubstsize$ free variables
Instantiate each argument of Ans_{heap} with an element of the answer substitution
If $leaf\_ptr$ corresponds to a conditional answer
Create a non-trailed term on the heap $ret_2(Ans_{heap}, undefined\_ptr)$
Else create a non-trailed term on the heap $ret_2(Ans_{heap}, null)$
Return the head of the List

Fig. A 1. Schematic pseudo-code for preserving views and altering OCCPs

backtracking into CP_{root}, and when its choices are exhausted, it will backtrack into the choice point prior to CP_{root}. Of course, $CP_{coalesced}$ also contains a $preservedList$ field.

In Fig. A 2 the values of $CP_{coalesced}$ come from CP_{root} rather than from CP, with the exception of fields representing heap values. In the stack-oriented backtracking used by Prolog, the $preservedList$ can be protected by setting $CP_{coalesced}$ to the value of the **H** register after the construction of $preservedList$. If there is a possibility that tabling will suspend and resume computations, preserve_occp_views() needs to freeze the heap space containing these answers so that the heap cells containing them will not be overwritten. If (**HF** reg $\neq$ bottom of heap), then there is an active tabled computation, and the heap freeze register is set to the value of the **H** register after construction of $preservedList$. The previous value of the **HF** will be reset using $CP_{coalesced}$.*previous_hfreg* once backtracking through $preservedList$ is done.

$CP_{coalesced}$	preservedViewMember	/* Failure Continuation */
	$ereg_{root}$	/* Top environment in stack (**E** reg)*/
	$ebreg_{root}$	/* Environment of top choice point (**EB** reg) */
	$hreg$	/* Top of heap (**H** reg) */
	$trreg_{root}$	/* Top of trail (**TR** reg)*/
	$dreg_{root}$	/* SLG-WAM delay register */
	$rsreg_{root}$	/* SLG-WAM root subgoal register */
	$previous\_cp_{root}$	/* Pointer to previous choice point */
	$previous\_top_{root}$	/* Pointer to the previous top of CP stack */
	$answerSubstSize$ M_{root}+1	/* Number of Variables in Answer Substitution */
	$answerSubst$ [M_{root}]	
	:	
	$answerSubst$ [0]$_{root}$	
	$preservedList$	
	$previous\_hfreg$	/* Previous SLG-WAM heap freeze register */

Fig. A 2. Choice point stack after coalescing

A.3 Backtracking through Preserved Views

Fig. A 3 shows the new SLG-WAM instruction that returns an answer through a preserved OCCP view when a coalesced choice point is backtracked into. The instruction reconstructs the SLG-WAM state at the time of its call (except for the heap register which was adjusted to protect the $preservedList$). Each answer substitution cell of the coalesced choice point is dereferenced to a heap or local stack cell, the dereferenced cell is bound to an element of the answer substitution, and the binding trailed. Afterwards, the $preservedList$ field is reset to point to the next list element if one is present; otherwise the **HF** register is set to its before the view was preserved, and the **B** register is set to the previous choice point.

```
Instruction preservedViewMember
   undo_bindings(B register)                  /* does not affect answers that were copied to heap */
   Restore SLG-WAM program registers
   Set up pointers to access ret_2(Substitution, Condition)
   If (Condition ≠ null) delay_negatively(Condition)
   For each cell, answerSubst[i], of B.answerSubst
      Bind argument i of Substitution to the dereferenced value of answerSubst[i]
            and trail the binding
   If B.preservedList has been consumed
      B = B.previous_cp
      HF reg = B.previous_hfreg
   Else make B.preservedList point to the next list element
```

Fig. A 3. Schematic pseudo-code for backtracking through preserved views

A.4 Discussion of View Consistency in Transparent incremental tabling

Of course, other approaches to view consistency are possible besides the one just presented. Before the above was implemented, answer tries were extended to include timestamps indicating when a given answer was valid (analogous to that of (Lindholm and O'Keefe 1987) for dynamic Prolog code). However, the time and space overhead of this approach was deemed to be too high. The actual implementation of the heap copying approach presented here uses XSB's general tabling code as much as possible, so that the cost to traverse tries and copy answers is generally very low.

It should be noted that the approach to view consistency is more closely linked to the data structures of the XSB engine than are other features of transparent incremental tabling, as view consistency interfaces with XSB's heap and stack freezing mechanisms.

Appendix B Performance Results

In the benchmarks that follow, all times are measured in seconds, and all space is measured in bytes unless otherwise specified [3].

B.1 Transparent Incremental Tabling and Linear Left 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. B 3) with and without IDG abstraction on the *edge/2* predicate. As discussed in Section 5 and shown in Fig. 6, the IDG created for such a query may differ greatly depending on whether abstraction is used. In the benchmarks, *edge/2* consists of ground facts representing a randomly generated graph $G(N/M)$ where N is the number of possible nodes in the graph, while M is the number of directed edges. Because of the left recursive form of *reach/2* together with its query form, the IDG nodes for *edge/2* are associated with subgoals $edge(\langle free \rangle, \langle free \rangle)$ from clause 1 of *reach/2*, and $edge(\langle ground \rangle, \langle free \rangle)$ where argument 1 is instantiated by different values of Z in clause 2 of *reach/2*. Using the re-evaluation strategies described in previous sections, any update to *edge/2* will cause a re-evaluation of the subgoal *reach*($\langle free \rangle, \langle free \rangle$) so that (in this program fragment) maintaining nodes of the form $edge(\langle ground \rangle, \langle free \rangle)$ provides no benefit, as their dependencies will be captured by $edge(\langle free \rangle, \langle free \rangle)$.

Nodes	No incr. tabling		Incr. tabling		Incr. tabling	+ abstraction
	CPU time	Table space	CPU time	Table space	CPU time.	Table space
100,000	0.12	7,663,728	0.21	21,671,136	0.13	10,273,672
1,000,000	2.19	72,121,240	3.43	211,184,888	2.34	92,746,112
10,000,000	40.9	701,364,952	59.7	2,070,845,368	41.2	902,048,352

Fig. B 1. Overhead for transparent incremental tabling on query evaluation of *reach(⟨free⟩,⟨free⟩)* over randomly generated graphs $G(Nodes/\frac{Nodes}{2})$

Nbr of asserts	Incr. tabling		Incr. tabling	+ abstraction
	Time to read/assert/inval.	Query time	Time to read/assert/inval.	Re-query time
100	0.004	3.53	0.003	2.29
1,000	0.023	3.67	0.022	2.29
10,000	0.19	4.20	0.17	2.38

Fig. B 2. Updates of *edge/2* for the query *reach(⟨free⟩,⟨free⟩)* over a randomly generated graph $G(1,000,000/500,000)$

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%. By using IDG abstractionat depth 0, the table space overhead becomes approximately 30%, and the time overhead 5-10%. Regardless of whether abstraction is used, Fig. B 1 demonstrates scalability for 2 orders of magnitude; the time scales log-linearly due to the need to maintain indices. Fig. B 2 shows that for

[3] Except for those reported in Section B.2.1, the benchmarks below were performed on a MacBook Pro, with a dual core 2.53 Ghz Intel i5 chip and 4 Gbytes of RAM. The benchmarks for Section B.2.1 were performed on a server at the University of Ferrara with 3 Intel dual-core 3.47 GHz CPUs and 188 megabytes of RAM running under Fedora Linux. The default 64-bit, single-threaded SVN repository version of XSB was used for all tests. Benchmark programs can be obtained at www.cs.sunysb.edu/~tswift/interpreters.html.

a batch updates (0.02%-2% of EDB), the overhead of re-evaluation is negligible, particularly if abstraction is used.

B.1.1 Non-Stratified Linear Left Recursion

Similar tests were made using the predicate *ureach/2* (Fig. B 3). The query *ureach(⟨free⟩,⟨free⟩)* was evaluated on the $G(1000000, 500000)$ graph of *edge/2* facts, so that all answers to the query had the truth value *undefined*. Overhead results for the initial query (Fig. B 4) are similar to those for *reach(⟨free⟩,⟨free⟩)* in terms of time; however the space overhead for incremental tabling is proportionally less as storing conditional answers requires its own space overhead (Sagonas et al . 2000). Fig. B 5 shows the time to add various numbers of *edge_1* facts, which causes new answers to be added to the table for *ureach(⟨free⟩,⟨free⟩)*, and also changes the truth value of some known answers from *undefined* to *true* as discussed in Section 4. From Fig. B 5 it can be seen that updating conditional answers imposes essentially no overhead compared to updating unconditional answers.

```
:- table ureach/2 as incremental.
:- dynamic edge/2, edge_1/2 as incremental.
ureach(X,Y):- reach(X,Z),edge(Z,Y).
ureach(X,Y):- edge(X,Y),undefined.
ureach(X,Y):- edge_1(X,Y).
```

Fig. B 3. Benchmark program for non-stratified left linear recursion

Nodes	No incr. tabling		Incr. tabling		Incr. tabling	+ abstraction
	CPU time	Table space	CPU time	Table space	CPU time.	Table space
100,000	0.14	21,333,304	0.24	35,540,760	0.15	24,143,168
1,000,000	2.30	208,352,144	3.61	347,416,664	2.42	228,977,672

Fig. B 4. Overhead for transparent incremental tabling on query evaluation of the non-stratified program *ureach(⟨free⟩,⟨free⟩)* over randomly generated graphs $G(Nodes/\frac{Nodes}{2})$

Nbr of asserts	Incr. tabling		Incr. tabling + abstr.	
	Time to read/assert/inval.	Query time	Time to read/assert/inval.	Re-query time
100	0.005	3.78	0.004	2.591
1,000	0.025	3.83	0.25	2.57
10,000	0.21	3.86	0.22	2.58

Fig. B 5. Updates of *edge_1/2* for the query *ureach(⟨free⟩,⟨free⟩)* over a randomly generated graph $G(1000000, 500000)$

B.2 Analysis of Transparent Incremental Tabling on a Program with KRR-style 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. Although the program is simplified and idealized in its content, computationally it requires the use of some sophisticated reasoning features. While the program contains stratified negation, its main computational challenge arises from its use of equality, which provides a reasoning capability similar in flavor to some description logics. The predicate *equals/2* allows terms using the

function symbol *parent_of/1* (formed from the EDB predicate *parent_of_edb/2*) to be considered as equal to constants representing individuals.

```
good_influence(P1,P2):- influences(P1,P2),
    sk_not(high_risk(P1)),sk_not(possible_risk(P1)),
    (high_risk(P2) ; possible_risk(P2)).

:- table high_risk_association/2 as incremental.
high_risk_association(Per1,Per2):- high_risk_contact(Per1,Per2),has_disease(Per2).
high_risk_association(Per1,Per2):- high_risk_association(Per1,Per3),high_risk_contact(Per3,Per2).

high_risk_contact(Per1,Per2):- may_share_needle(Per1,Per2).
high_risk_contact(Per1,Per2):- may_have_unprotected_sex(Per1,Per2).

:- table high_risk/1 as incremental.
high_risk(Per):- high_risk_association(Per,_),!.

:- table possible_risk_association/2 as incremental, answer_abstract(3).
possible_risk_association(Per1,Per2):- might_be_sexual_partner(Per1,Per2),
    high_risk_contact(Per2,_).
possible_risk_association(Per1,Per2):- possible_risk_association(Per1,Per3),
    might_be_sexual_partner(Per3,Per2).

:- table possible_risk/1 as incremental.
possible_risk(Per):- possible_risk_association(Per,_),!.

influences(Per1,Per2):- loves(Per2,Per1).
influences(Per1,Per2):- works_for(Per2,Per1).
influences(Per1,Per2):- attends_church(Per2,Church),pastor(Church,Per1).
influences(Per1,Per2):- lives_at(Per1,Loc),lives_at(Per2,Loc).

may_share_needle(Per1,Per2):- obtained_needle(Per1,Needle,_Loc1), returned_needle(Per2,Needle,_Loc2),Per1 Per2.
may_share_needle(Per1,Per2):- share_needle_report(Per1,Per2,_Per3).

might_be_sexual_partner(Per1,Per2):- loves(Per1,Per2),sk_not(related(Per1,Per2)).
might_be_sexual_partner(Per1,Per2):- sexual_partner_report(Per1,Per2,_Per3).

:- table related/2 as incremental.
related(Per1,Per2):- equals(Per1,parent_of(Per2)).
related(Per1,Per2):- equals(Per1,parent_of(parent_of(Per2))).

:- table loves/2 as incremental.
loves(X,Y):- loves(Y,X).
loves(X,Y):- friend(X,Y).
loves(X,Y):- equals(parent_of(X),Y).
loves(X,Y):- grandparent_of(X,Y).

:- table equals/2 as incremental, subgoal_abstract(3).
equals(X,Y):- equals(Y,X).
equals(parent_of(X),parent_of(X)).
equals(parent_of(X),Y):- parent_of_edb(X,Y).
equals(parent_of(parent_of(X)),Y):- parent_of(X,Z),equals(parent_of(Z),Y).

father_of(X,Y):- equals(parent_of(X),Y),male(Y).
mother_of(X,Y):- equals(parent_of(X),Y),female(Y).
grandparent_of(X,Y):- equals(parent_of(parent_of(X)),Y).

:- dynamic friend/2, returned_needle/3, obtained_needle/3, share_needle_report/3, sexual_partner_report/3 as incremental.
:- dynamic has_disease/1, works_for/2, may_have_unprotected_sex/2, pastor/2, parent_of_edb/2, lives_at/2,attends_church/2
                                                                    as incremental,abstract(0).
```

Fig. B 6. A social network example showing KRR features

The EDB for this program consists of 12 different dynamic predicates as seen at the bottom of the program [4]. The use of the *parent_of/1* function within *equals/2* quickly leads to non-termination and unsafe negative subgoals during query evaluation. Unsafe negative subgoals are soundly addressed by XSB's *sk_not/1* which skolemizes non-ground variables in an atomic subgoal for the purpose of calling a negative subgoal. Non-termination is addressed in two ways. The use of subgoal abstraction in *equals/2* ensures that there will be only a finite number of tabled queries to this predicate, and in general ensures termination for programs with finite models (Riguzzi and Swift 2013). However, the predicate *possible_risk_association/2*, produces an infinite number of answers for the benchmark data set. The use of answer abstraction (or restraint) for this predicate, ensures sound (but not complete) terminating query evaluation (Riguzzi and Swift 2013) [5].

Thus, the ability to incrementally maintain tables for queries to this program requires the ability to update three-valued models that arise from answer abstraction, and to combine with tabled negation and subgoal abstraction. As a first benchmark test, a small EDB of about 10,000 facts about a population of 10,000 persons was generated, and *good_infuence*$(\langle bound \rangle, \langle free \rangle)$ was queried for 200 randomly chosen values for its first argument. If no incremental tabling was used, the combined CPU time for these queries averaged 1.14 seconds and table space was about 233 megabytes — as discussed further below the relatively large cost for this query was almost entirely due to the use of equality. When transparent incremental tabling was used with no abstraction, the cost rose to 3.02 seconds, and 865 megabytes. By applying IDG abstraction the initial query time dropped to 2.73 seconds and 655 megabytes. The purpose of this sets of declarations was only to test the overhead of transparent incremental tabling for queries and updates: they should not necessarily be considered to be "optimal" for these tests

Fig. B 7 shows times to re-evaluate the queries to *good_influence/2* mentioned above after inserting N randomly generated facts for a given predicate (the "Asserts" column); and then after retracting these inserted facts (the "Retracts" column). Most of the times in Fig. B 7 are near the level of noise, however recomputation of several of the predicates timed out [6]. Analysis of these timeouts showed that they arose because the additional facts caused a large number of new (sub-)tables to be created for the 200 queries. Usually, this only occurred after 12,500 facts were added, but for *parent_of_edb/2* which strongly affects goals to *equals/2*, the addition of 500 facts led to a timeout, while the addition of 100 facts led to a 5.57 second recomputation time. Although the program is not wholly monotonic, it is largely so, and computations after retractions were always fast. Fig. B 8 shows the times to assert or retract plus the time taken to invalidate affected subgoals via traverse_affected_nodes(). Except for updates to *parent_of_edb/2* invalidation did not take a significant amount of time

[4] The social network programs and supporting data can be found at http://www.cs.sunysb.edu/~tswift.

[5] Briefly, if an answer has an argument A with depth greater than a given bound, A is rewritten so that terms with depth equal to the bound are replaced by new variables; then the answer A is assigned the truth value *undefined*

[6] Timeouts, denoted Tout in Fig. B 7, were triggered after one minute. The short timeout period was to avoid excessive memory consumption on the laptop benchmarking machine. Retracts of bulk inserts could not be measured, and are designated as n/a. As the population size was 10,000, 12,500 distinct facts could not be generated for the unary EDB predicate *has_disease/1*.

Predicate	Asserts 100	500	2500	12500	Retracts 2500	12500
friend/2	0.08	0.37	2.36	Tout	0.02	n/a
returned_needle/3	0.01	0.01	0.01	0.01	0.01	0.01
obtained_needle/3	0.01	0.01	0.01	0.02	0.01	0.01
share_needle_report/3	0.03	0.03	0.13	0.55	0.01	0.01
sexual_partner_report/3	0.01	0.02	0.12	Tout	0.01	n/a
has_disease/1	0.01	0.01	0.01	n/a	0.01	n/a
works_for/2	0.01	0.04	0.42	1.76	0.01	0.01
may_have_unprotected_sex/2	0.03	0.08	0.12	0.56	0.02	0.02
pastor/2	0.01	0.01	0.01	0.01	0.01	0.01
parent_of_edb/2	5.57	Tout	Tout	Tout	n/q	n/a
lives_at/2	0.01	0.01	0.07	2.11	0.01	0.01
attends_church/2	0.01	0.01	0.01	0.01	0.01	0.01

Fig. B 7. CPU times to re-evaluate *good_influence/2* for 200 first-argument bindings after batch updates. The program uses non-specialized equality, and the EDB size is $\mathcal{O}(10^4)$. The top group of predicates use depth-0 IDG abstraction; the bottom group has no IDG abstraction.

Predicate	Asserts 100	500	2500	12500	Retracts 2500	12500
friend/2	0.01	0.01	0.02		0.03	
returned_needle/3	0.01	0.01	0.03	0.14	0.03	0.14
obtained_needle/3	0.01	0.01	0.03	0.15	0.03	0.20
share_needle_report/3	0.01	0.01	0.02	0.14	0.03	0.17
sexual_partner_report/3	0.01	0.01	0.03		0.03	
has_disease/1	0.01	0.01	0.02	n/a	0.02	n/a
works_for/2	0.01	0.01	0.02	0.10	0.04	0.16
may_have_unprotected_sex/2	0.03	0.08	0.11	0.52	0.04	0.16
pastor/2	0.01	0.01	0.02	0.11	0.03	0.15
parent_of_edb/2	27.8	Tout	Tout	Tout	37.2	Tout
lives_at/2	0.01	0.01	0.02	0.11	0.03	0.17
attends_church/2	0.01	0.01	0.02	0.11	0.03	0.16

Fig. B 8. CPU times to apply updates and to invalidate subgoals created by queries to *good_influence/2* for 200 first-argument bindings. The program uses non-specialized equality, and the EDB size is $\mathcal{O}(10^4)$. The top group of predicates use depth-0 IDG abstraction; the bottom group has no IDG abstraction.

B.2.1 Scalability Analysis on a Program with KRR Features

As a next step, the equality relation in the previously mentioned program of Fig. B 6 was specialized so that it had the form:

```
:- table equals/2 as incremental, subgoal_abstract(3).
equals(X,Y):- atomic(X),Y = parent_of(_),equals(Y,X).
equals(parent_of(X),parent_of(X)).
equals(parent_of(X),Y):- parent_of_edb(X,Y).
equals(parent_of(parent_of(X)),Y):- parent_of_edb(X,Z),equals(parent_of_(Z),Y1),Y1 = Y.
```

In this form, the first clause of *equals/2* is changed so that symmetry is applied only if the first argument corresponds to a nominal individual (constant), and the second argument has a functional form. The fourth clause is changed so that subgoals of the form *equals*$(\langle bound \rangle, \langle bound \rangle)$ are not called by this clause, but instead subgoals of the form *equals*$(\langle bound \rangle, \langle free \rangle)$ are called. These changes, which do not affect the semantics of the program, significantly reduce the time and space required for query evaluation, although goals to *equals/2* are still computationally expensive to update.

With this change, a series of 200 queries as described above were tested on EDBs ranging 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, various batch updates were timed along with time to re-evaluate queries (Figs. B 10 and B 11). Specifically for $N = 2500, 12500, 62500$ and 312500, N asserts of each EDB predicate were performed and timed; and then the N asserted facts were retracted and timed. Except for updates to *parent_of_edb/2*, re-evaluation time was low compared to initial query time (even compared to the initial query time for non-incremental tabling). These benchmarks illustrate the scalability of this implementation of transparent incremental tabling even for very large IDGs. In Figs. B 10 and B 11, the IDG contained over 750 million edges; after the update sequences mentioned above were applied, it contained more than 1 billion edges.

EDB Size	Query Time	Table Space	IDG Nodes	IDG Edges	Non-incr Query Time
$\mathcal{O}(10^5)$	3.9	0.51 Gbytes	22,374	7,362,284	1.7
$\mathcal{O}(10^6)$	62.1	5.33 Gbytes	67,106	78,612,966	24.5
$\mathcal{O}(10^7)$	679.8	51.56 Gbytes	505,972	753,798,584	391.9

Fig. B 9. CPU times to initially evaluate *good_influence/2* for 200 first-argument bindings for EDBs of various sizes. The program uses specialized equality.

Predicate	Asserts				Retracts			
	2500	12500	62500	312500	2500	12500	62500	312500
friend/2	3.11	3.16	2.63	3.51	3.11	3.16	2.58	2.91
returned_needle/3	3.11	6.59	2.57	2.96	3.11	3.21	2.57	2.87
obtained_needle/3	3.11	3.16	2.59	2.65	3.11	3.16	2.52	2.52
share_needle_report/3	3.12	3.16	2.52	2.54	3.11	3.16	2.52	2.54
sexual_partner_report/3	3.12	3.16	2.52	2.54	3.11	3.16	2.52	2.55
has_disease/1	3.46	3.51	2.81	2.80	3.46	3.50	2.80	2.81
works_for/2	3.14	3.25	3.34	4.81	3.11	3.16	2.52	2.52
may_have_unprotected_sex/2	4.34	4.37	3.51	3.51	4.33	4.37	3.51	3.51
pastor/2	3.12	3.16	3.34	2.51	3.11	3.16	2.51	2.52
lives_at/2	3.12	3.16	2.52	2.58	3.11	3.16	2.52	2.52
attends_church/2	3.12	3.16	2.52	2.52	3.16	3.16	2.52	2.52

Fig. B 10. CPU times to re-evaluate *good_influence/2* for 200 first-argument bindings after batch updates. The program uses specialized equality, and the EDB size is $\mathcal{O}(10^7)$. The top group of predicates use depth-0 IDG abstraction; the bottom group has no IDG abstraction.

Predicate	Asserts 2500	12500	62500	312500	Retracts 2500	12500	62500	312500
friend/2	0.12	0.60	3.01	15.9	0.13	0.67	3.43	18.1
returned_needle/3	0.12	0.60	3.01	16.0	0.13	0.69	3.51	18.4
obtained_needle/3	0.15	0.74	3.74	19.5	0.17	0.83	4.21	22.0
share_needle_report/3	0.12	0.61	2.99	15.9	0.11	0.01	2.98	15.8
sexual_partner_report/3	0.12	0.61	2.99	16.0	0.11	0.59	2.98	15.9
has_disease/1	0.07	0.33	1.67	8.6	0.08	0.39	1.87	10.5
works_for/2	0.12	0.57	0.42	16.1	0.0	0.65	3.34	18.2
may_have_unprotected_sex/2	0.34	1.68	8.45	43.3	0.13	1.75	8.87	45.3
pastor/2	0.07	0.33	1.71	19.5	0.08	0.39	2.04	10.9
parent_of_edb/2	380.9	Tout	Tout	Tout	222.6	Tout	Tout	Tout
lives_at/2	0.11	0.56	2.82	14.7	0.14	0.68	3.45	18.0
attends_church/2	0.07	0.34	1.71	8.9	0.08	0.43	2.15	11.3

Fig. B 11. CPU times to apply updates and to invalidate subgoals created by queries to *good_influence/2* for 200 first-argument bindings. The program uses specialized equality, and the EDB size is $\mathcal{O}(10^7)$. The top group of predicates use depth-0 IDG abstraction; the bottom group has no IDG abstraction.

Appendix C A Note on Usability

The XSB manual contains information on how transparent incremental tabling may be used in practice; however to make this paper self-contained, we provide an outline of some usability and system aspects.

XSB has a variety of tabling mechanisms that are used for different purposes. As seen from Fig. B 6, transparent incremental tabling works properly with subgoal abstraction and with answer abstraction; as discussed in Section 4, transparent incremental tabling works properly with well-founded negation regardless of the tabled negation operator: for instance with *sk_not/1* in Fig. B 6, or with other XSB operators such as *tnot/1*. It also works properly with tabled attributed variables (supporting tabled constraints). A variety of dynamic code may be used as a basis for transparent incremental tabling including not only regular facts and rules, but also facts that are interned as XSB tries. Incremental tables, of whatever form, may be used alongside non-incremental tables, although special declarations must be made if an incremental table depends on a non-incremental table.

Within the current version of XSB, transparent incremental tabling does not yet work properly with call subsumption, answer subsumption, hash-consed tables, or multi-threaded tables; also, predicates that are tabled as incremental must use static code rather than dynamic code. Attempts to declare a predicate using an unsupported mixture of tabling features causes a compile-time permission error.

There are situations where it is convenient or necessary to abolish an incremental table rather than updating it. An example of this occurs when an exception is thrown. If an exception is thrown over a choice point to a completed table no action need be taken; however if an exception is thrown over a choice point to an incomplete tabled subgoal (including one that is being recomputed), XSB abolishes the table as its computation has become compromised. In transparent incremental tabling, abolishing an incremental table is not problematic. If a table T is to be abolished, tables that depend on T must be invalidated before actually abolishing T itself. When a call is made to a subgoal with an invalidated affected node, portions of the IDG that were removed through abolishing will be reconstructed during the calls made by *incremental_reeval()*, due to the actions of lazy recomputation.