Incremental answer completion in the SLG-WAM

Terrance Swift, Alexandre Miguel Pinto, and Luís Moniz Pereira *

Centro de Inteligência Artificial, Universidade Nova de Lisboa

Abstract. The SLG-WAM of XSB Prolog soundly implements the Well-Founded Semantics (WFS) for logic programs, but in a few pathological cases its engine treats atoms as undefined that are true or false in WFS. The reason for this is that the XSB does not implement the SLG ANSWER COMPLETION operation in its engine, the SLG-WAM – rather ANSWER COMPLETION must be performed by post-processing the table. This engine-level omission has not proven significant for applications so far, but the need for ANSWER COMPLETION is becoming important as XSB is more often used to produce well-founded residues of highly non-stratified programs. However, due to its complexity, care must be taken when adding ANSWER COMPLETION to an engine. In the worst case, the cost of each ANSWER COMPLETION operation is proportional to the size of a program P, so that the operation must be invoked as rarely as possible, and when invoked the operation must traverse as small a fragment as possible of P. We examine the complexity of ANSWER COMPLETION; and then describe its implementation and performance in XSB's SLG-WAM such that the invocations of the operation are restricted, and which is limited in scope to Strongly Connected Components within a tabled evaluation's Subgoal Dependency Graph.

Designers of logic programming engines must weigh the usefulness of operations against the burden of complexity they require. Perhaps the best known example is the *occurs check* in unification. Prologs derived from the WAM do not perform occurs check between two terms, since its cost may be exponential in the size of the terms. Rather, the occurs check must be explicitly invoked through the ISO predicate `unify_with_occurs_check/2` or a similar mechanism. For evaluating normal programs using tabling, checking for certain positive loops involves similar considerations. While most positive loops can be efficiently checked, positive subloops within larger negative loops are more difficult to detect, and account for the complexity of evaluating a program P according to WFS, which is $atoms(P) \times size(P)$, where $atoms(P)$ is the number of atoms of P and $size(P)$ is the number of rules of P.

As implemented in XSB, the SLG-WAM detects positive loops between tabled subgoals so that answers are not added to a table unless they are true, or are involved in a loop through negation and so are undefined at the time of their addition (termed *conditional answers*). As shown in Theorem 1 below, this sort of evaluation can be done in time linear in $size(P)$. However, a situation can arise where certain conditional answers are later determined to be true or false. This determination may break a negative loop, which uncovers a positive loop and makes the answers false. Within SLG, this situation is addressed by the ANSWER COMPLETION operation, which is not implemented within the currently available version of the SLG-WAM. So far, the lack of ANSWER

* tswift@cs.sunysb.edu, amp@di.fct.unl.pt, lmp@di.fct.unl.pt

COMPLETION has not proven a problem for most programs. However, the SLG-WAM is increasingly being used to produce well-founded residues for highly non-stratified programs for applications involving intelligent agents (e.g. [2]), where the need for ANSWER COMPLETION is greater.

This paper examines issues involved in adding ANSWER COMPLETION to the SLG-WAM. We illustrate the situation of a positive loop begin uncovered when a negative loop is resolved through a concrete example, and then we provide a formal result on the contribution ANSWER COMPLETION makes to the complexity of computing WFS. We introduce an algorithm for efficiently performing ANSWER COMPLETION (subject to its complexity), and discuss performance results obtained by implementing it in the SLG-WAM. Due to space requirements, we must assume knowledge of tabled evaluation of WFS through SLG resolution [1] as well as certain data structures of the SLG-WAM [3].

Example 1. The following program is soundly, but not completely, evaluated by the SLG-WAM, where `tnot/1` indicates that tabled negation is used:

```
:- table p/1,r/0,s/0.
p(X):- tnot(s).      p(X):- p(X).
   s:- tnot(r).         s:- p(X).          r:- tnot(s),r.
```

The well founded model for this program has true atoms $\{s\}$ and false atoms $\{r, p(X)\}$. Recall that literals that do not have a proof and that are involved in loops over default negation are considered *undefined* in WFS. Unproved literals involved only in positive loops, i.e., without negations, are *unsupported* and, hence, *false* in WFS. Accordingly, `p(X)`, whose second clause fails, is *false*; however, a query to $p(X)$ in XSB indicates that $p(X)$ is *undefined*. The reason is that during evaluation the engine detects a strongly connected component (SCC) of mutually dependent goals containing $p(X)$, r and s, along with negative dependencies, and no answers for any of these goals. In such a situation, the SLG-WAM delays negative literals and continues execution. Here, the literal `tnot(s)` in the rule `p(X):- tnot(s)` is delayed, producing an answer `p(X):- tnot(s)|`, indicating that `p(X)` is conditional on a *delay list*, here `tnot(s)`. That answer is returned to the goal `p(X)` in the clause `p(X):- p(X)` and a conditional answer `p(X):- p(X)|` is derived. Later, a positive loop is detected for `r`, causing its truth value to become *false*. The failure of `r` causes `s` to become *true*, and SIMPLIFICATION removes the answer `p(X):- tnot(s)|`. At this stage, however, no further simplification is possible for `p(X) :- p(X)|`, which is now unsupported.

The ANSWER COMPLETION operation addresses such cases by detecting positive loops in dependencies among conditional answers. More precisely, ANSWER COMPLETION marks *false* sets of answers that are not *supported*: i.e. conditional answers for completed subgoals that contain only positive, and no negative dependencies in their delay lists. The creation of unsupported answers are uncommon in the SLG-WAM because its evaluation is *delay minimal* – that is, the engine performs no unnecessary DELAYING operations [4]. Delay minimality reduces the need for simplification of dependencies among answers, and thereby the chances of uncovering positive loops among answers, as with the answer `p(X):- p(X)|` above.

1. Complexity

We begin by showing that queries to programs that do not need ANSWER COMPLETION can be evaluated in $\mathcal{O}(size(P))$. Such programs include stratified ones, and also non-stratified programs that contain no positive loops within negative SCCs in their dynamic dependency graphs [1].

Theorem 1. *Let Q be a query to a finite ground normal program P. Under a cost model with constant time access to all subgoals, nodes, and delay elements of each SLG forest in an evaluation, and constant time access to each clause in P, a partial SLG evaluation that does not perform* ANSWER COMPLETION *can be constructed that is linear in the size of P.*

The algorithm ITERATE ANSWER COMPLETION below iteratively applies ANSWER COMPLETION operations, calling Check Supported Answers() to perform a check for positive loops. Check Supported Answers() is an adaptation of Tarjan's algorithm for SCC detection (cf. `http://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm`), which is linear in $size(P)$. Note that in the worst case, ANSWER COMPLETION operations iteratively need to be applied, and that each time it is applied, a single atom would be found *false*. In that case, program evaluation would have a cost proportional to $atoms(P) \times size(P)$, which is equivalent to the known complexity for WFS.

2. Implementation of ANSWER COMPLETION

Within an SLG evaluation, a tabled subgoal can be marked as *complete*, which indicates that all possible answers have been produced for the subgoal, although SIMPLIFICATION and ANSWER COMPLETION operations may remain to simplify or delete conditional answers. Completed subgoals do not require execution stack space, but only table space, so that completing subgoals as early as possible is a critical step for engine efficiency. Accordingly the SLG-WAM performs *incremental completion* via a completion instruction, which maintains information about mutually dependent sets of subgoals (SCCs), and completes each SCC when all applicable operations have been performed. In addition to marking each subgoal S in an SCC as complete, if S failed (has no answers) the completion instruction may initiate SIMPLIFICATION for conditional answers that depend negatively on S. In terms of ANSWER COMPLETION, observe that any positive loop among the delayed literals of conditional answers must be contained within the SCC being completed, as each delayed literal was a selected literal before it was delayed. This incremental approach has several benefits. Performing ANSWER COMPLETION operation within the completion instruction restricts the space that any such operation needs to search. In addition, performing ANSWER COMPLETION after all other SIMPLIFICATION operations have been performed on answers within the SCC similarly reduces search space. As a final optimization, ANSWER COMPLETION is not required unless delaying has been performed within the SCC, a fact that is easily maintained using data structures in the SLG-WAM's *Completion Stack*, which maintains information about SCCs. The pseudo code for Iterate Answer Completion(), which traverses all subgoals in the SCC using the *Completion Stack*, and checks each

[1] The proof of Theorem 1 is contained in an appendix of a fuller version of this paper available on request.

answer for support, deleting unsupported answers from the table and invoking SIMPLIFICATION operations, is presented in Figure 1. SIMPLIFICATION may remove further negative loops, and uncover new unsupported other answers as a side-effect. In such case, the ANSWER COMPLETION procedure should be executed once more, and this is guaranteed by the use of the $reached\_fixed\_point$ flag. A fixed-point is reached when all answers within the scope of the SCC are known to be supported.

Algorithm Iterate Answer Completion($CompletionStack$)

```
reached_fixed_point = FALSE;
while not reached_fixed_point
    reached_fixed_point = TRUE;
    foreach subgoal S in the Completion Stack
        foreach answer A for subgoal S
            if not Check_Supported_Answer(A)        /* A is unsupported */
                reached_fixed_point = FALSE;
                delete A;
                propagate A's deletion's simplifications;
```

Fig. 1. Algorithm ITERATE ANSWER COMPLETION.

Check Supported Answer This procedure (Figure 2) does the actual check of whether a (positive) answer is unsupported. It detects positive loops whenever it encounters an answer that has already been visited and which is in the SCC. In this case, the algorithm terminates returning $FALSE$ to indicate the answer is unsupported. On the other hand, if the answer has been visited but is not part of the SCC, it means such answer has been produced during some other branch of query-solving and was therefore, rightfully supported and stored in the table: the algorithm terminates returning $TRUE$. Checking a non-visited answer consists of 1) marking it as visited; 2) adding it to the state of the search (stored in the $Completion\ Stack$); and then 3) traversing all the Delay Elements (literals) of the Delay Lists for the answer recursively checking each in turn for supportedness. Whenever an answer is determined to be unsupported, all Delay Lists containing (Delay Elements that reference) it are deleted, which may cause further simplification and iterations of ANSWER COMPLETION.

The above algorithm has been implemented within the completion instruction of XSB. Full performance analysis is still underway. Preliminary results indicate advantages of our heuristics: traditional benchmarks like `win/1` either do not use SIMPLIFICATION or use it seldom so that there is no overhead for ANSWER COMPLETION. A stress test that performs a large number of repetitions of Example 1 shows an overhead of at most 18%. Example 1 is actually representative of the typical situation where ANSWER COMPLETION is needed. This is so because it contains (at least) two rules for some literal (in this case $p(X)$) where the first one depends on a loop through negation (rendering $p(X)$ *undefined*) and the second one depend on a positive loop, which is unsupported. The "undefinedness" coming from the first clause is passed on to the $p(X)$ in the body of the second one. Only ANSWER COMPLETION can then be used to clean away the delay list with $p(X)$ from the answer coming from the second clause for $p(X)$. The "pathological" nature of this example follows from the, until now, XSB's

```
Algorithm Check Supported Answer(Answer)

if Answer has already been visited
    if Answer is in the SupportCheckStack return FALSE;
    else return TRUE;
else
    mark Answer as visited;
    push Answer onto the SupportCheckStack;
    mark Answer as supported_unknown;
    foreach Delay List DL for Answer
        if Answer is supported_true exit loop;
        mark DL as supported_true;
        foreach Delay Element DE in the Delay List DL
            if DL is not supported_true exit loop;
            if DE is positive and it is in the SupportCheckStack
                recursively call Check Supported Answer(Answer of DE)
                if Answer of DE is not supported_true
                    mark DL as supported_false;
    if DL is supported_false
        remove DL from Answer's DLs list
        if Answer's DLs list is now empty
            delete Answer node;
            simplify away unsupported positives of Answer;
        else mark Answer as supported_true;
    if the Answer node was deleted return TRUE;
    else return FALSE;
```

Fig. 2. Algorithm CHECK SUPPORTED ANSWER.

SLG-WAM inability to rightfully detect and simplify away unsupported literals such as $p(X)$.

3. Conclusions

WFS is used in an increasing number of applications, from intelligent agents, to inheritance in object logics, to supply-chain analysis. However, the abstract complexity of WFS is a concern when embedding into the semantic core of a programming language like Prolog. Theorem 1 shows that the non-linearity of WFS can be separated from other parts of an engine for WFS; and the optimizations of the algorithm presented here, together with the preliminary performance results, underscore the suitability of WFS for general-purpose programming.

References

1. W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *JACM*, 43:20–74, 1996.
2. L. M. Pereira and G Lopes. Prospective logic agents. In *EPIA*, volume 4874 of *LNAI*, pages 73–86, 2007.
3. K. Sagonas and T. Swift. An abstract machine for tabled execution of fixed-order stratified logic programs. *TOPLAS*, 20(3):586 – 635, 1998.
4. K. Sagonas, T. Swift, and D. S. Warren. The limits of fixed-order computation. *Theoretical Computer Science*, 254(1-2):465–499, 2000.