An Engine for Computing Well-Founded Models

Terrance Swift

CENTRIA — Universidade Nova de Lisboa
tswift@cs.sunysb.edu

Abstract. The seemingly simple choice of whether to use call variance or call subsumption in a tabled evaluation deeply affects an evaluation's properties. Most tabling implementations have supported only call variance or, in the case of XSB Prolog, supported call subsumption only for stratified programs. However, call subsumption has proven critical for (sub-)model generation as required for some kinds of program analysis (e.g. type analysis) and for semantic web applications such as RDF inference. At the same time, the lack of well-founded negation has prevented the use of call subsumption in producing residual programs, and has limited its use in semantic web applications that require negation (e.g. evaluation of OWL ontologies). This paper describes an engine for evaluating normal programs under the well-founded semantics (WFS) in which the evaluation method can be based on a mixture of call subsumption and call variance, chosen at the predicate level. The implementation has been thoroughly tested for both local and batched evaluation and is available in version 3.2 of XSB.
Keywords: Tabling, WAM

Tabled evaluations can differ in their subgoal reuse strategy. Given a selected tabled subgoal G, answers may be resolved when there is a subgoal in the table that subsumes G, in which case *call subsumption* is used, or only when a variant of G is in the table, where *call variance* is used. Call variance preserves the instantiation patterns of selected subgoals in an evaluation, making it efficient for query-oriented applications and suitable for tabling meta-interpreters. Call subsumption, on the other hand, often evaluates only the most general subgoals, giving it a more bottom-up flavor. Call subsumption is therefore suitable for (sub-)model generation as required for stable model generation, for certain type analyses, or for some semantic web applications. Call subsumption is harder to implement than call variance and is usually not supported in tabling implementations. Previous versions of XSB implemented call subsumption for stratified programs [3], and allowed the strategy of subsumption or variance to be declared on a predicate basis. This paper describes how XSB's implementation of call subsumption is extended to support full well-founded semantics The extended implementation has been thoroughly tested, and is available in XSB version 3.2.

A Motivating Example Computing A-box entailment from a standard OWL wine ontology (`www.w3.org/TR/2003/CR-owl-guide-20030818/wine`) provides a striking example of a use of call subsumption for a semantic web application. When translated into datalog by KAON2 (`http://kaon2.semanticweb.org`) the ontology is a highly recursive datalog program. Computing a query using call subsumption in XSB terminated correctly in 10 seconds, while call variance terminated with memory errors after 100+ seconds. The difference was due to the lack of relevancy

in query evaluation: essentially the entire (sub-)model of the wine program needed to be constructed. Because call subsumption avoided recomputing subsumed calls, it saved space and time, and is quite competitive with special-purpose ontology tools [4]. While this is a single example, two conclusions are clear. Call subsumption can be critical for "bottom-up" computations that do not benefit from relevancy. Also, since ontologies in general require negation when they are translated into datalog, evaluating WFS using call subsumption is an important problem.

1. Implementation

We briefly describe the main ideas of the engine as implemented in the SLG-WAM of XSB. Because of space limitations, we must assume a general knowledge of tabling and its implementation. As background, [3] describes call subsumption in the SLG-WAM, while [5] describes the overall SLG-WAM architecture for WFS and [2] the data structures used for well-founded residual programs.

We begin by describing actions of the SLG-WAM on definite programs. When call variance is used, if a tabled subgoal G is new to an evaluation, it is associated with a generator choice point to backtrack through program clauses. If G was previously selected and completed, the engine simply backtracks through answers in an answer trie. If G was previouslyselected but is not completed a consumer choice point is created that will backtrack through answers using an *answer return list* for G. The answer return list is needed to backtrack efficiently through the dynamically changing answer trie.

Call subsumption extends the cases an implementation must support. Let subgoal G subsume a subgoal $G\theta$. If $G\theta$ is selected before G no special action is taken – $G\theta$ and G are evaluated just as with call variance, However if $G\theta$ is selected *after* G, two subcases arise. If G is completed, $G\theta$ simply backtracks through answers for G failing on those that do not unify with $G\theta$. If G is not completed, then a consumer choice point is created for $G\theta$ (as with call variance) along with a *subsumed subgoal frame* in the table. In addition, a special answer return list is created for $G\theta$ pointing to each answer in the answer trie for G that unifies with $G\theta$. When $G\theta$ consumes the final answer in its answer return list, the engine traverses the answer trie for G to generate a new answer return list for $G\theta$, consisting of answers that unify with $G\theta$ *and* were added after the previous list was generated for $G\theta$. Figure 1 schematically illustrates the relation of G and $G\theta$. [3] described an important optimization where nodes in an answer trie are associated with time stamps, and the time stamps manipulated to avoid unnecessary search through the trie when regenerating answer lists for subsumed subgoals. Note that this mechanism supports those stratified programs where no ground negative subgoal $G\theta$ occurs in the same SCC as G (i.e. the same set of mutually dependent subgoals).

To evaluate WFS, situations must be handled that arise when G and $not(G\theta)$ are mutually dependent (and neither is completed). The program

```
:- table win/1 as subsumptive.
win(X):- move(X,Y),tnot(win(Y)).
```

and query `win(X)` under varying extensions of `move/2` serves as a running example. (`tnot/1` is XSB's predicate for tabled negation).

Changes to DELAYING**.** Consider the extension `move(a,b) move(b,a)`. Evaluation of the goal `win(X)` creates the goals `win(a)` and `win(b)`. The two sub-

sumed goals are created after `win(X)` is called and before it is completed, so for each subsumed subgoal a consumer choice point is created along with a subsumed subgoal frame which is inserted in a *consumer subgoal chain* in the subgoal frame for `win(X)`. However, since the program is non-stratified, both `win(a)` and `win(b)` are undefined in WFS and are treated as *conditional answers*, denoted `win(a):- tnot(win(b))|` and `win(b):- tnot(win(a))|` — where the "`|`" symbol indicates that the preceding literal is delayed. To represent delay lists when call subsumption is used, `tnot/1` must be changed to determine whether a negated goal is subsumed by another goal, so that the delay list will properly contain a ground literal, rather than the subsuming literal `tnot(win(X))`. To effect this, the goal *tnot(G)* determines whether there is a subsuming goal for *G*, and if not associates the delay element with *G*'s (producer) subgoal frame. Otherwise, *tnot(G)* determines whether the subsuming goal is completed or not, creates the subsumed subgoal frame if necessary, and associates the delay element with the subsumed subgoal frame for *G* – in this case `win(a)` or `win(b)`.

Changes to SIMPLIFICATION**.** Next, consider the extension `move(a,b) move(b,a) move(b,c)` and the evaluation of the query `win(a)`. In this case, no delaying is necessary for either call variance or call subsumption: `win(a)` does not subsume `win(b)` or `win(c)`. `win(c)` is determined to be false, and this false value causes `win(b)` to be determined to be true and `win(a)` false before any conditional answers are created. However, if call subsumption were used for the goal `win(X)`, all goals would be in the same mutually dependent SCC, so that DELAYING and SIMPLIFICATION must both occur.

SIMPLIFICATION operations are initiated in two cases [5]. When a subgoal G is completed with no answers (fails), any answers conditional on G must be deleted; and any answers conditional on *tnot(G)* must have *tnot(G)* removed from their delay lists. Similarly, when an unconditional answer A is derived, any answers conditional on A must be simplified. Each case can initiate a chain of SIMPLIFICATION operations, since removing an answer can cause a goal to fail; while removing a literal from a delay list can cause an answer (or a ground subgoal) to become unconditionally true. Figure 1 shows the supporting data structures. A subgoal G has an answer Ans in its answer trie along with a subsumed goal $G\theta$. Ans contains $tnot(Gdep)$ in its delay list. To perform simplification, the subgoal $Gdep$ contains a list of backpointers to each answer (such as Ans) containing $tnot(Gdep)$ in its delay list (conditional answers have backpointers similar to those for subgoals). In addition, pointers from delay lists to answers (through Delayinfo structures) and from answers to (producing) subgoals are used to traverse table space and propagate SIMPLIFICATION operations.

Both cases in which SIMPLIFICATION is initiated are affected by call subsumption. To handle the case initiated by a failing subgoal G, the engine checks the subgoal frame to see whether G has been declared to use call subsumption. If so, the engine must check whether there are any subgoals subsumed by G that now fail. The chain of subsumed subgoals is traversed, and each subsumed subgoal frame checked for a non-null backpointers cell. The subsumed subgoal is checked for backpointers, and a SIMPLIFICATION operation executed aif the subgoal fails. In our running example, when the subsumed subgoal `win(c)` fails, a SIMPLIFICATION operation is performed us-

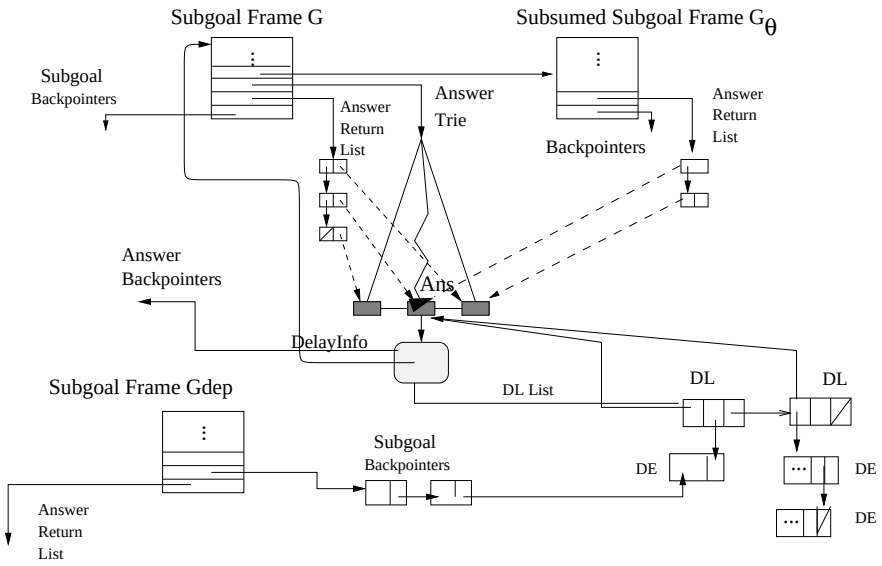

Fig. 1. Schematic Table Space for Call Subsumption with Conditional Answers

ing backpointers of `win(c)` to remove the answer `win(b):- tnot(win(c))|`, in turn propagating a SIMPLIFICATION operation to make the answer `win(a):- tnot(win(b)|` unconditional. Consider the case of the second simplification where an unconditional answer A is derived, other answers having either A or *tnot(A)* in their delay lists need to be simplified. Answers having A in their delay lists are obtained through the backpointers list off of the answer A itself, and so are not affected by call subsumption. On the other hand, obtaining answers having $tnot(A)$ in their delay list is more difficult using call subsumption, as they depend on the subgoal A which may be subsumed. As shown in Figure 1, there is a pointer from an answer to its producer (or variant) subgoal frame, but not to frames of any subsumed subgoals. However while A may not need to initiate simplification through its producing call, it may need to initiate simplification through a subsumed call. Again using our running example, when `win(b)` becomes true, the answer `win(a) :- tnot(win(b))|` must be deleted. This occurs through the backpointers of the subgoal `win(b)`, but as mentioned there is no pointer from the answer `win(b)` (for `win(X)`) to the subsumed subgoal frame for `win(b)`. Fortunately, the trie data structures make the check for possible subsumed subgoals efficient. Tabled subgoals are stored in a trie just as answers are, with subgoal frames (including those for subsumed subgoals) as leaves of the subgoal trie. A term is constructed from the answer substitution A, and trie indexing is used to determine whether A is also a subsumed subgoal frame: if so, simplification will be performed.

Performance Tests of `win/1` provide information about the speed of basic operations. The table below illustrates CPU times in seconds and table space in bytes for `win/1` using call variance and call subsumption. When `move/2` is a chain, call subsumption is significantly slower than call variance. For the goal `win(1)` this is due to the fact that `tnot/1`, currently written largely in Prolog, is more complicated for call subsumption.

In addition, call subsumption for the goal `win(X)` treats all subgoals as if they were in the same SCC, and must delay and simplify each answer. When `move/2` is a cycle, the time to add conditional answers dominates all strategies. For the goal `win(X)`, call subsumption must perform the same amount of delaying for the chain and for the cycle (and the same amount of delaying as call variance for the cycle). However, note that for the chain, 50,000 extra simplifications are performed in a negligible amount of time. Also, call subsumption requires slightly more table space than call variance, when there is no actual subsumption to exploit. With `win(X)` goals, call subsumption saves some table space, but unlike the wine example, the savings are limited in this example as there is at most a single answer per subsumed goal.

	call variance `win(1)`	call subsump. `win(1)`	call subsump. `win(X)`
50000 chain	0.19 / 5,582,396	0.53 / 5,982,596	1.0 / 3,653,620
50000 cycle	1.14 / 9,985,548	0.72 / 10,585,940	0.98 / 7,654,660

2. Discussion

In terms of related work, a global answer table has been implemented in YAP for definite programs [1], and allows the sharing of answers between different subgoals of a tabled predicate. Global tables and call subsumption can both reduce the size of tables, but each has advantages that the other does not. A global table can allow sharing of answers between subgoals that unify, even if neither subsumes the other. Call subsumption, on the other hand can reduce computation time as well as space.

The implementation described is intended to be robust. Accordingly, before distributing the described implementation in XSB, it was run on a suite of programs testing WFS, residual programs, and tabled constraints. The test suite contained over 12,000 lines of code, and was run on various platforms under local and batched evaluation for 32-bit and 64-bit compilations. The implementation described here makes minor changes to scheduling, relatively straightforward changes to table data structures and access routines, and more complex changes to simplification instructions. It is important to note that call subsumption does not affect a tabling system's mechanisms for suspending and resuming a computation – the aspect of tabling that is most intimately connected with the WAM data structures. This means that the approach described here is (in principle) applicable to systems such as Ciao, Mercury, B-Prolog and ALS that support tabling outside of the SLG-WAM.

References

1. J. Costa and R. Rocha. One table fits all. In *PADL*, pages 195–208, 2009.
2. B. Cui, T. Swift, and D. S. Warren. From tabling to transformation: Implementing non-ground residual programs. In *Implementations of Declarative Languages*, 1999.
3. E. Johnson, C.R. Ramakrishnan, I.V. Ramakrishnan, and P. Rao. A space-efficient engine for subsumption based tabled evaluation of logic programs. In *4th International Symposium on Functional and Logic Programming*, 1999.
4. S. Liang, P. Fodor, H. Wan, and M. Kifer. OpenRuleBench: An analysis of the performance of rule engines. In *WWW: Semantic Data Track*, 2009.
5. K. Sagonas, T. Swift, and D. S. Warren. An abstract machine for efficiently computing queries to well-founded models. *JLP*, 45(1-3):1–41, 2000.