Interfacing a Tabled-WAM Engine to a Tabling Subsystem Supporting Both Variant and Subsumption Checks

Ernie Johnson

Abstract

In [4] we devised and implemented a mechanism for efficiently evaluating tabled logic programs based on call subsumption. Since then, the addition of answer subsumption has yielded a fully-subsumptive mechanism for tabled evaluation. Its subsequent incorporation into the XSB logic programming system [9] provides users with the ability to evaluate a tabled predicate using *either* variant or subsumption checks. In this article we describe how this dual-mode tabling subsystem can be incorporated into any WAM-style [1] tabling engine. The interaction required between the tabling engine and the tabling subsystem is detailed, as well as the interface provided by our subsystem to support this interaction. The advantages of utilizing this interface for supporting future planned modifications are also discussed.

1 Introduction

Abstractly, the act of performing a tabled evaluation requires the cooperation between two components of a tabling engine. The *core engine* contains the machinery to perform both top-down and table-supported fixpoint evaluations. Such machinery for supporting tabled resolution includes, for example, mechanisms for suspending and resuming subcomputations, and maintaining dependencies between calls. The second component of a tabling engine is the *tabling subsystem* which maintains and controls access to the table. The interaction between these two components was described in [4] through a series of abstract operations. These operations are sufficient for capturing this interaction independent of the implementation of the table and whether it employs variant or subsumption checks.

In a *variant-based tabled evaluation*, proven instances, or answers, are only shared between variant subgoals.[1] Although variant-based tabling has proven to be greatly beneficial, this approach permits only limited sharing of answer computations. By

Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York 11794−4400. E-mail: `ejohnson@cs.sunysb.edu`

This work was supported in part by NSF grants C-CR 9711386, C-CR 9876242, and EIA 9705998.

[1] Two terms are *variants* if they can be made identical through variable renaming.

contrast, a *subsumption-based tabled evaluation* allows for the use of answers by subsumed subgoals,[2] thereby permitting greater reuse of computed results. Tabling by subsumption, however, is more complex to engineer as answer resolution must support indexing into dynamically expanding answer sets. The condition under which answers are recorded in the table is similarly provided by these relations. In the remainder of this article, we shall assume that "variant-based tabling" or "subsumption-based tabling" refers to the use of the *same* relation for the condition on *both* calls and answers.

In [4] we presented a table organization and algorithms based on Time-Stamped Tries for efficiently computing tabled evaluations on definite programs using *call subsumption*. This is a partial implementation of subsumption-based tabling whereby a subsumptive condition is applied toward calls, but a variant condition toward answers. An implementation of this design proved to be efficient in both time and space as compared to two prior engine designs: one utilizing call subsumption based on Dynamic Threaded Sequential Automata (DTSA) [6], and the other a variant design based on tries [5]. Of particular importance is the space utilization of TSTs which was shown (from both a theoretical and a practical perspective) to be superior to DTSA during evaluations in which calls were subsumed, and no more than twice that of the trie-based variant design irrespective of the existence of subsumed calls during an evaluation.

The practicality of this approach merited its incorporation into the open-source XSB logic programming system [9]. XSB, based upon SLG resolution [2], provides variant-based evaluation of tabled predicates. This strategy also serves as the basis for evaluations of normal logic programs according to the well-founded semantics [8]. Furthermore, variant-based tabling interacts well with most Prolog constructs, making XSB an attractive alternative to Prolog. The recent addition of *answer subsumption* to the TST design provides XSB with a second, fully-subsumptive, approach to tabling, and the ability to provide superior performance on "purer" (more declarative) programs. In fact, the decision of which tabling strategy to use can be made on a *per predicate* basis, affording the user a finer grain of control over the evaluation.

XSB is implemented via the SLG-WAM [7], a WAM-style abstract machine [1] suitably augmented for performing tabled evaluations. As such, interaction between the core engine and the tabling subsystem occurs within special abstract machine *tabling instructions*. This interaction is facilitated through the sharing of data structures and the invocation of table-manipulation routines. The subsequent integration of subsumption, however, has led to a proliferation of these constructs. Indeed, as each tabling strategy is efficiently implemented based on its own separate design, it is desirable to maintain them both. Furthermore, the tabling instructions of the engine require generalizing to support subsumptive as well as variant evaluations, necessitating the coexistence of these variant- and subsumption-supporting constructs within the same tabling instruction.

In this article is described a new interface architecture designed to facilitate the interaction between XSB's core engine and its dual-strategy tabling subsystem. This description includes the data structures shared by the two system components as

[2]A term t_1 *subsumes* a term t_2 if t_2 is an instance of t_1. Further, t_1 *properly subsumes* t_2 if t_2 is not a variant of t_1.

well as the access routines which allow the engine to interact with the table in either a variant or subsumptive mode. These routines facilitate this interaction by implementing the aforementioned abstract tabling operations. Invocation of one of these routines dispatches the correct underlying tabling-strategy-specific operation, depending on the evaluation strategy utilized by the predicate. This framework provides a single, consistent interface to the core engine, clarifying the division between it and the tabling subsystem. Specific aspects of the tabling strategy are hidden behind this interface, allowing the core engine to focus on the details of the evaluation.

2 Interaction During a Tabled Evaluation

In this section, we revisit the abstract tabling operations of [4] while describing the interaction between the core engine and the tabling subsystem.

Upon encountering a call to a tabled predicate, the engine must first determine the manner in which to satisfy it. This determination, made through the abstract tabling operation **call-check-insert**, is based upon the contents of the table at the time the call is issued and the method of tabling, either variant- or subsumption-based, used by the predicate of the call. If a subgoal in the table exists whose answers can be used to satisfy this call, then the call will be resolved a tuple-at-a-time via these answers. We refer to this process as *answer resolution* and to the call as a *consumer*.

The abstract operation of selecting an answer from this set and resolving it with the consumer is called **retrieve-answer**. Performing this operation requires a handle to the call, a reference to the answer set from which the answers will be drawn, and an abstract structure called an *answer continuation*. This latter structure represents the set of all answers remaining to be retrieved. When an answer is selected and unified with the call by the tabling subsystem, an updated answer continuation is returned to the engine for use during the call's subsequent **retrieve-answer** operation. If no such answer is available, then the consumer must be *suspended* by the engine until more answers are available. Such consumers are *resumed* by the engine as part of the fixpoint operation, discussed below.

On the other hand, if no subgoal exists whose answers can be used to satisfy the call, then the call is copied into the table and it is evaluated via resolution with program clauses. When an answer is derived, an attempt is made to insert it into the table. We refer to this abstract operation as **answer-check-insert** and to the call as a *producer*, as it produces the answers stored in the table. Several pieces of information are required to perform this operation: the method of tabling used by the predicate (as this determines how the table will interpret redundant answers), the answer term itself to be inserted, and a reference to the set into which the answer is to be inserted.

Once program resolution has been exhausted, the producer takes part in a fixpoint check. For the purposes of this article, how this operation is performed is viewed as an engine-side issue. However, we note that the fixpoint operation requires access to all the call instances representing consumers of the answer set produced by this call.

TablingStrategy	`VARIANT` and `SUBSUMPTION` based evaluations
TabledCallType	`PRODUCER` and `CONSUMER` tabled calls

(a)

Table	Representation of the table for a predicate
EntryDesc	Table entry (subgoal and answer set pair) descriptor
AnsTmplt	Answer template; references subterms of the call
AnsCont	Answer continuation; tracks consumed/unconsumed answers

(b)

table_new(TablingStrategy) **returns** ⟨ Table ⟩
table_call_search(Arity, Args, Table)
    returns ⟨ TabledCallType, EntryDesc, AnsTmplt ⟩
table_answer_search(EntryDesc, AnsTmplt) **returns** ⟨ Boolean ⟩
table_pending_answer(EntryDesc, AnsTmplt, AnsCont)
    returns ⟨ Answer, AnsCont ⟩
table_unify_answer(AnsTmplt, Answer) **returns** ⟨ ⟩
table_complete_entry(EntryDesc) **returns** ⟨ ⟩

(c)

Figure 1: Interface to the tabling subsystem: (a) enumerated types, (b) data structures, and (c) routines.

The engine invokes the table operation **pending-answers** on each such consumer to check whether it has consumed each and every answer from this set. Those calls which have not yet consumed all unifying answers must be rescheduled by the engine for further resolution. An invocation of this operation requires the same information as **retrieve-answer**.

3 Facilitating the Interaction

We have developed a flexible yet concise interface between XSB's tabling subsystem, which supports both variant-based and subsumption-based tabling, and its core engine. A summary of this interface is shown in Figure 1. We assume, as in XSB, that a table is created for each individual tabled predicate. Creation of a table for a predicate in the subsystem is accomplished through the `table_new()` routine. The tabling strategy to be used for evaluating the predicate is supplied as an argument. Because this information is stored as part of the table, the engine is relieved from tracking this information explicitly.

The abstract tabling operation **call-check-insert** is implemented by the interface routine `table_call_search()`.[3] Communicated to this routine are the arity of the call, an array containing the arguments of the call, and the table for the predicate of the call. Returned from this function are several pieces of information crucial to

[3]In XSB parlance, the word "search" is applied to functions which perform a lookup with conditional insertion.

the evaluation of the call. The first is an indication as to how the call should be evaluated: either as a producer or a consumer. Also returned is a descriptor for the table entry with which **call-check-insert** has associated the call. This structure — depicted in Figure 2 — contains several fields, some of which are used directly by the engine to carry out the evaluation, while others are for internal use by the tabling subsystem. The particular form of this structure depends upon the type of the call, as indicated by the first returned value. Lastly, a structure containing references to subterms of the call, called an *answer template*, is supplied to the engine. This structure is intended for use when invoking other interface routines — in lieu of the call itself — and is not directly manipulated by the core engine. After `table_call_search()` returns, the engine begins preparations for the evaluation of the call in the manner indicated by this operation. In either case, a choice point frame (CPF) is created for the call, in which references to the entry descriptor and answer template are stored.

If the call is a producer, then the returned descriptor is a *producer entry descriptor* (see Figure 2(a)). Using the *ProducerCP* field of this structure, the engine creates a cross-reference between it and its choice point frame. The ability to locate the producer's CPF given an entry descriptor facilitates the engine's tracking of consumer calls. Recall from Section 2 that access to these consumers is required during the completion (fixpoint) operation. We assume that the engine organizes the consumers of a particular answer set as a chain of choice point frames whose head is stored in the CPF of the producer of this set. Program resolution proceeds, with answer derivation necessitating the invocation of an **answer-check-insert** operation, provided by the routine `table_answer_search()`. Supplying the entry descriptor and answer template returned from `table_call_search()` gives this routine access to the correct answer set, variable bindings, and tabling strategy used by the predicate. This function returns a boolean value indicating whether or not the answer was inserted into the table, allowing the engine to take appropriate measures commensurate with its *scheduling strategy* [3].

If the call is a consumer, then the descriptor returned from `table_call_search()` is a *consumer entry descriptor* (see Figure 2(b)). This structure references, through its *ProducerED* field, the producer descriptor which contains the answer set against which this call will be resolved. From there, the producer's CPF can be reached, allowing for the insertion of this consumer CPF into the consumer chain. Creation of a consumer CPF permits the call to backtrack through the relevant answers of the set. Resolution of the call with one such answer — hence, execution of a **retrieve-answer** operation — is performed through the combination of two interface routines. First, a relevant answer is sought using the function `table_pending_answer()` which implements the abstract operation **pending-answers**. Given the consumer entry descriptor, the answer template (which is needed for identifying a relevant answer), and the answer continuation (which distinguishes old from new answers with respect to consumption), this function returns a handle to an unconsumed answer stored in the table — `NULL` if none exists — plus an updated answer continuation. Supplying an initial answer continuation of `NULL` indicates to the tabling subsystem that the call is just beginning answer resolution, allowing the subsystem to make the necessary arrangements for its support. The answer returned from this routine can then be

Table	Reference to the table of which this entry is a part.
EntryPair	Ref. to internal representation of subgoal and answer set.
ProducerCP	Reference to the producer's choice point frame.
IsComplete	Flag indicating whether the answer set is complete.

(a)

Table	Reference to the table of which this entry is a part.
EntryPair	Ref. to internal representation of subgoal and relevant answer set.
ProducerED	Reference to entry descriptor from which answers are obtained.

(b)

Figure 2: Excerpt of contents of (a) producer and (b) consumer entry descriptors. Only fields which are central to interface support are shown.

unified with the call via `table_unify_answer()`. After the unification, the engine may proceed in a forward manner with the evaluation. If no answer exists, then the call is suspended by the engine.

After all program clauses are exhausted, the engine begins a fixpoint check on the producer and all of its consuming calls. Each call on the chain of consumers maintained in the producer CPF is checked for consumption of all generated answers in the set. This is accomplished by invoking `table_pending_answer()` on each such consumer. Similar to its use during answer resolution, the table entry, answer template, and answer continuation — all structures associated with the *consumer* — are passed to this function. In this case, however, the returned updated answer continuation is ignored, and the returned answer reference simply checked for non-`NULL`-ness. Such a value indicates that at least one unconsumed answer exists for that consuming call. If such is the case, then the consumer is scheduled for further answer resolution.

This brings up an important point. The engine must ensure that the environment of the consuming call (i.e., bindings to the variables) is resumed *before* calling `table_pending_answer()`. As these consumers are currently suspended during the fixpoint operation, their environments must be reconstituted to ensure that the call, and hence answer template, appears as it did when the call was issued.

Once the fixpoint has been reached, a call to `table_complete_entry()` advises the tabling subsystem that computation of answers for a producer entry has ceased, allowing for the removal of auxiliary structures created for the consumption of answers from an incomplete answer set. For future reference, a flag is set in the producer's entry descriptor indicating completeness of the answer set. We note that reaching a fixpoint may involve the consideration of other interdependent producer calls.

4 Optimizing the Interface

The above description simplifies certain details for clarity. We now modify that description to bring it closer to the actual implementation found in XSB.

Although not specifically stated, all consuming calls which are variants of one another share the same entry descriptor. Indeed, the information contained within this structure is relevant for all such calls. Additionally, notice that the producer's entry descriptor would serve just as well in cases where the consumers were variants of the producer. We therefore modify the above interface to distinguish two types of consumers: those which are variants of a producer, and those which are properly subsumed by a producer. Appropriate modifications are made to `table_call_search()` to make this distinction and to return a description of the given call using this level of detail. Further, in cases where the consumer is a variant of a producer, a *producer entry descriptor* is returned by the interface. Notice, then, that consumer entry descriptors actually only represent *properly subsumed consumers*.

The operation **pending-answers** also deserves further treatment. We noted in the previous section that this operation should be invoked within the environment of the consuming call, thus assuring that the answer template is valid. During the completion operation, this would require many environment switches — one for each consumer. However, the answer template is in actuality needed only when the consumer is properly subsumed *and* the answer continuation is empty. Under these circumstances, the answer template is used to obtain additional relevant answers from the producer (we direct the reader to [4] for details). Therefore, we modify the interface such that the core engine provides `table_pending_answer()` with a routine for reconstituting the environment of the given consumer, and further require that `table_pending_answer()` invoke this routine when needed.

5 Discussion

We have introduced an architecture for interfacing a tabling subsystem, capable of maintaining a predicate's table using either variant- or subsumption-based checks, to the core engine of a tabled WAM-based system. This architecture provides for the cooperation between these two system components through a set of data structures and interface routines. Further, table-specific information is hidden within these structures, within the tabling subsystem itself, and behind the interface routines. By permitting the engine to interact with the table through a set of abstract operations, we allow it to focus on details of the evaluation rather than those of the tabling strategy and its implementation.

The value of this ability to hide implementation details from the core engine will become even more apparent when modifications begin for a multi-threaded version of the XSB engine. The architecture of this design calls for each thread to maintain a separate address space for a core engine, but all will share a global table. Such a design will require the addition of locking mechanisms on the table. A preliminary investigation suggests that the appropriate location for these mechanisms lie within the interface routines discussed in this article.

Finally we note the flexibility of this framework as it pertains to supporting other tabling strategies. There are certain situations for which it is desirable to perform evaluations utilizing variant-based call checks but subsumption-based answer checks, thereby reducing the number of "acceptable" answers. This strategy can currently be programmed through XSB's tabled aggregation facility. However, engine-level support could be added easily. Little more would be required than the addition of another tabling strategy type and corresponding checks within the interface routines for directing the invocation of the appropriate tabling operation.

Acknowledgements

The author would like to thank Terrance Swift for his comments on an earlier draft of this article, and also I. V. Ramakrishnan and C. R. Ramakrishnan for useful discussions during the initial preparation of this piece.

References

[1] H. Ait-Kaci. The WAM: A (real) tutorial. Technical report, DEC Paris Research Laboratory, 1990.

[2] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1), 1996.

[3] J. Freire, T. Swift, and D. S. Warren. Beyond depth-first: Improving tabled logic programs through alternative scheduling strategies. *Journal of Functional and Logic Programming*, 1998.

[4] E. Johnson, C. R. Ramakrishnan, I. V. Ramakrishnan, and P. Rao. A space efficient engine for subsumption-based tabled evaluation of logic programs. In A. Middeldorp and T. Sato, editors, *4th Fuji International Symposium on Functional and Logic Programming*, number 1722 in Lecture Notes in Computer Science, pages 284–299. Springer-Verlag, November 1999.

[5] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient access mechanisms for tabled logic programs. *Journal of Logic Programming*, January 1999.

[6] P. Rao, C. R. Ramakrishnan, and I. V. Ramakrishnan. A thread in time saves tabling time. In *Joint International Conference/Symposium on Logic Programming*. MIT Press, 1996.

[7] T. Swift and K. Sagonas. An abstract machine for tabled execution of fixed-order stratified logic programs. In *Transactions on Programming Languages and Systems*. ACM, May 1998.

[8] A. van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, July 1991.

[9] The XSB Group. The XSB programmer's manual, Version 2.2, April 2000. Available at `http://xsb.sourceforge.net`.