Efficient access mechanisms for tabled logic programs

I.V. Ramakrishnan a,1, Prasad Rao b,2, Konstantinos Sagonas c,*, Terrance Swift d,3, David S. Warren a,4

a Department of Computer Science, State University of New York at Stony Brook, NY 11794-4400, USA
b Bellcore, 445 South Street, Morristown, NJ 07960-6438, USA
c Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001, Heverlee, Belgium
d Department of Computer Science, University of Maryland at College Park, A.V. Williams Building, College Park, MD 20742, USA

Received 17 April 1997; received in revised form 26 February 1998; accepted 17 April 1998

Abstract

The use of tabling in logic programming allows bottom-up evaluation to be incorporated in a top-down framework, combining advantages of both. At the engine level, tabling also introduces issues not present in pure top-down evaluation, due to the need for subgoals and answers to access tables during resolution. This article describes the design, implementation, and experimental evaluation of data structures and algorithms for high-performance table access. Our approach uses tries as the basis for tables. Tries, a variant of discrimination nets, provide complete discrimination for terms, and permit a lookup and possible insertion to be performed in a single pass through a term. In addition, a novel technique of substitution factoring is proposed. When substitution factoring is used, the access cost for answers is proportional to the size of the answer substitution, rather than to the size of the answer itself. Answer tries can be implemented both as interpreted structures and as compiled WAM-like code. When they are compiled, the speed of computing substitutions through answer tries is competitive with the speed of unit facts compiled or asserted as WAM code. Because answer tries can also be created an order of magnitude more quickly than asserted code, they form a promising alternative for representing certain types of dynamic code, even in Prolog systems without tabling. © 1999 Elsevier Science Inc. All rights reserved.

Keywords: Tabling; Indexing; Tries; Discrimination nets; SLG-WAM; Implementation

*Corresponding author. Tel.: +32 1632 7560; fax: +32 1632 7996; e-mail: kostis@cs.kuleuven.ac.be.
1 E-mail: ram@cs.sunysb.edu.
2 E-mail: prasadr@bellcore.com.
3 E-mail: tswift@cs.umd.edu.
4 E-mail: warren@cs.sunysb.edu.

0743-1066/99 - see front matter © 1999 Elsevier Science Inc. All rights reserved.
PII: S0743-1066(98)10013-4
1. Introduction

Bottom-up evaluation of logic programs offers well-known advantages over top-down: programs terminate if they have the finite term-depth property (as defined in [17]); redundant subcomputations are eliminated; and non-stratified programs can be evaluated according to the well-founded semantics without the possibly exponential number of negative contexts (see [3]). Of course direct bottom-up evaluation is unacceptable for general query evaluation, since it evaluates all possible queries to a program. As a result, a persistent theme of logic programming research has been to investigate how to combine the advantages of bottom-up evaluation with the goal-orientation of top-down techniques. This effort has led to many systems based on magic evaluation and related strategies (see [12] for a survey of such research).

The high speed of top-down engines, though, has sometimes been neglected in the literature. At least for loop-free, stratified programs with few redundant subcomputations, top-down engines, such as those based on the WAM [20], can be substantially faster than bottom-up engines. Thus, rather than adding goal-orientation to a bottom-up engine, a natural approach to evaluating in-memory queries is to add bottom-up capabilities, or tabling, to a Prolog engine. The XSB system [14] follows this latter approach. The goal of XSB is to evaluate tabled predicates (using SLG resolution [3]) in approximately the same time as non-tabled predicates (using SLDNF). Based on our experience, it appears that the greatest efficiency gains under present technology can be made at the level of engine design. This article reports on engine enhancements for tabling that yield substantial performance improvements. Specifically, we present results of experiments for reducing the time for an engine to access tabled information and more generally to access dynamically created facts.

Consider table access operations for definite programs:

Call CheckInsert. When a tabled subgoal is called, a check must be made to see whether the subgoal is redundant or not. In the current version of the XSB system, this amounts to a variant check of whether the new subgoal is a variant of one that already exists in the table. If it is, the subgoal is termed a consumer and answer clauses are resolved against it. If not, the subgoal is termed a generator, entered into the table, and program clause resolution is used instead. We associate with each tabled subgoal a set of answers which are stored in the answer table associated with the subgoal.

Answer CheckInsert. When an answer is derived for a particular subgoal, a check is made to determine whether it has already been entered into the answer table for the subgoal. If it has, the derivation path fails, a vital step for ensuring termination. If not, the computation continues, and the answer is scheduled for return to the applicable consumer subgoals.

Answer Backtracking. When a consumer subgoal is created, it backtracks through answers in the table in the course of its evaluation.

Observe that naive table lookups and inserts of calls and answers can result in repeatedly rescanning terms and thereby may degrade performance considerably. For in-memory computations, the goal of Prolog speed for tabled programs is only achievable if the above three operations are performed with very little overhead. Specifically, in the case of the call check/insert step, a call to a tabled predicate must take nearly the same time as a call to a non-tabled predicate. Similarly, the time of answer check/insert should be small relative to the time required to derive an answer, since
this operation occurs for each solution to a tabled predicate. And finally, backtracking through answer clauses must take roughly the same time as backtracking through unit program clauses. Needless to add, engine modifications to enable efficient storage and retrieval of subgoals and answers in tables cannot compromise the performance of the system for any class of problems.

While compile-time approaches can partially alleviate these problems – for instance, such approaches can indicate which predicates should be tabled and which should not – their ultimate solution must be dynamic. We may thus speak of the Table Access Problem as one of designing efficient algorithms and data structures for accessing tabled data at the level of an evaluation engine. This problem is addressed in this article.

Our results regarding the Table Access Problem are as follows: First, we devise a trie-based method for storing subgoals and their answers in tables. Tries eliminate repeated rescanning of tabled terms during lookups and inserts. Second, using tries in conjunction with substitution factoring, a technique developed in this article, further reduces the overheads of answer lookup and insert operations. Third, we devise a technique for dynamically compiling tries, leading to the ability to backtrack through answer clauses at speeds comparable to compiled WAM code. As a final result, we demonstrate the generality of these techniques by applying them to asserted facts and exhibiting significant speedups over existing methods. Trie-based tabling, substitution factoring and compiled tries have been present in the XSB system since Version 1.4.2, and the option of tries for asserted facts has been present since Version 1.7. XSB has been installed in about a thousand sites for educational, research and commercial use, and runs under a variety of platforms.

The rest of this article is organized as follows. Section 2 describes trie-based methods for storing subgoal and answer tables. In Section 3 we present the concept of substitution factoring. Implementation aspects of trie-driven tabling are discussed in Section 4. The technique of dynamically compiling tries is described in Section 5. In Section 6 we present performance results which provide strong evidence that our techniques can indeed allow tabled logic programs to achieve speeds comparable to Prolog programs. We conclude with a discussion of the relevance of this work to in-memory query optimization techniques. We assume knowledge of the WAM [20].

2. Tabling tries

We assume the standard definitions of terms and the notions of substitution and subsumption of terms. A position in a term is either the empty string $\lambda$ that reaches the root of the term, or $p \cdot i$, where $p$ is a position and $i$ is an integer, that reaches the ith child of the term reached by $p$. The symbol $t$ (possibly subscripted) denotes terms; $f$, $g$ denote function symbols; and all capital letters (possibly subscripted) denote variables. We use the terms call and subgoal interchangeably, as well as the terms answer and return.

The trie data structure was originally invented to index dictionaries [7] and has since been generalized (as discrimination nets) to index terms (see [2] for use of tries in indexing logic programs and [1,6,8,10,15,19] for automated theorem proving and term rewriting). We will use a variant of the discrimination net in [1] as the data structure for tabling calls and their answers. We refer to it as the tabling trie.
The essential idea underlying a tabling trie is to partition a set $T$ of terms based upon their structure so that looking up and inserting these terms will be efficiently done. The tabling trie is a tree-structured automaton whose root represents a start state, and whose leaves each corresponds to a term in $T$. Each internal state specifies a position to be inspected in the input term when reaching that state. The outgoing transitions specify the function symbols expected at that position. A transition is taken if the symbol in the input term at that position matches the symbol on the transition. On reaching a leaf state we say that the input term matches the term associated with the leaf state. The root-to-leaf path taken to reach the leaf state corresponds to a left-to-right preorder traversal of the matching term. When no outgoing transition from a state can be taken, a lookup operation fails. On the other hand, for an insert operation we add an outgoing transition for the symbol and a new destination state for this transition. The position that will be associated with the new state is the next position in the preorder traversal of the input term. We illustrate the operations on a tabling trie using the example in Fig. 1.

To look up the term $rt(a, f(a, b), a)$ we begin at state $s_1$. Since position 1 in the input term is $a$, we make a transition to state $s_2$. In this state we inspect the next preorder position of the input (position 2) and make a transition to state $s_3$. Transition from this state on seeing $a$ (in position 2.1) leads to the state $s_4$. Continuing thus we finally reach the leaf state $s_6$ and declare a match. To insert the term $rt(a, g(b, c), c)$ we again start at state $s_1$ and make a transition to state $s_2$. Since there is no outgoing transition labeled $g/1$ we create a new transition for $g/1$ and a new destination state for this transition. In this new state we will inspect position 2.1 which is next in preorder traversal of the input term. Continuing in this fashion we will create three more new states ($s_{13}, s_{14},$ and $s_{15}$) yielding the trie shown in Fig. 2.

Our point of departure from the trie formalism described in [1] is in our treatment of variables. Recall that our lookup/insert operations perform a variant check, i.e., two terms match if they are identical up to variable renaming. Performing a variant check for calls and answers, has advantages for a tabling system. A detailed discus-

---

**Fig. 1.** (b) is a trie for the terms in (a).
sion of the various issues involved can be found in [3], but we mention the two main advantages here. First, tabling based on variance can support Prolog-style meta-programming using built-in predicates such as \texttt{vart/1}, and second, variant checks can be implemented very efficiently, as shown below.

To realize variant checks in our tabling trie we \textit{standardize} the representation of a term to treat each variable as a distinct constant. Formally this can be done through a bijection, \texttt{numbervar}, from the set of variables in \( t \), denoted by \( \text{vars}(t) \), to the sequence of constants \( \langle v_1, v_2, \ldots, v_n \rangle \) such that \( \text{numbervar}(V) < \text{numbervar}(W) \) if \( V \) is encountered before \( W \) in the left-to-right preorder traversal of \( t \). For example in the term \( f(g(Y,Z),X,Z) \), \( \text{numbervar}(X) \), \( \text{numbervar}(Y) \) and \( \text{numbervar}(Z) \) are \( v_3 \), \( v_1 \) and \( v_2 \) respectively. Let \( \text{numbervar}(t) \) denote the term \( t_0 \), where \( \theta(V) = \text{numbervar}(V) \) for every variable \( V \) in \( \text{vars}(t) \). Thus, \( \text{numbervar}(f(g(Y,Z),X,Z)) = f(g(v_1, v_2), v_3, v_2) \). Consequently, two terms are variants of each other if and only if \( \text{numbervar}(t_1) = \text{numbervar}(t_2) \).

Converting a term to standard form can be done concurrently with the process of lookup and insertion (Implementation details are in Section 4). As an example of this process, consider the addition of the term \( rt(a, g(b, c), c) \) to the trie of Fig. 2. Starting at state \( s_1 \) we make matching transitions till state \( s_4 \). The next position in the preorder traversal of the input term contains a variable whose standardization is \( v_1 \) and matches the label of the outgoing transition to state \( s_7 \). At this state there is no outgoing transition for a variable whose standardization is \( v_1 \) (the only transition is labeled with \( v_2 \)), so a new transition labeled with \( v_1 \) is created together with a new destination state, \( s_{16} \), for this transition yielding the trie shown in Fig. 3.

From these examples and the description above it is easy to see that

\textbf{Proposition 2.1.} For the subgoal and answer check/insert steps, each element of the input term is examined only once.

In summary, we claim that trie-based tabling has two major advantages over hash-based tabling.
1. **Complete discrimination.** Tries completely discriminate between terms no matter where in a term the discriminating element lies. In contrast, if hashing is based on a limited prefix of the term, it will suffer when the discriminating element is deeply nested.

2. **Single pass check/insert.** For the subgoal and answer check/insert, a single traversal of the term is necessary, regardless of whether the term needs to be copied into the table. In hash-based tabling lookup alone may require multiple passes over the term due to hash collisions. Furthermore, insertion will require a separate pass. Given the prevalence of these operations in a tabling system, the savings in time over a two-pass operation can be substantial.

The above claims are substantiated in the performance results presented in Section 6, where it is also shown that use of the trie-based approach can save space over hash-based methods. The approach just described provides a useful optimization for lookup and insert operations in the call and answer tables. However, because it treats the two tables as independent entities, it does not exploit sharing of bindings between a specific call and its answers. This sharing can be exploited by **substitution factoring** described in Section 3.

### 3. Substitution factoring

As mentioned in the introduction, we associate an answer table with every subgoal in the subgoal table. Given a subgoal $G$, any answer $A$ for the subgoal is subsumed by $G$, and can be represented as $G\theta_A$. We call $\theta_A$ an **answer substitution** for $G$. Note that the sum of the sizes of terms in $\theta_A$ is less than the size of $G\theta_A$. The core idea of substitution factoring is to store only the answer substitutions, and to create a mechanism of returning answers to consuming subgoals that takes time linear in the size of $\theta_A$ rather than the size of $G\theta_A$. In other words, substitution factoring ensures that answer tables contain no information that also exists in their associated subgoal table. Operationally this means that the non-variable symbols in the subgoal need not be examined again during either answer check/insert or answer backtracking.
Let $G$ denote a subgoal and $\{V_1, V_2, \ldots, V_n\}$ denote the set of variables in $G$ such that $\text{numbervar}(V_i) = v_i$. An answer substitution $\theta_A$ for $G$ is of the form $\{V_1 \leftarrow t_1, V_2 \leftarrow t_2, \ldots, V_m \leftarrow t_m\}$. We call the sequence $\langle t_1, t_2, \ldots, t_m \rangle$ the answer tuple corresponding to the answer substitution $\theta_A$. Observe that we can reconstruct the answers given the subgoal, the variable sequence $\langle V_1, V_2, \ldots, V_n \rangle$ and the answer tuples. Hence if we store the variable sequence with the subgoal, then we need only store the answer tuples in the answer table. Because the variable sequence is determined when a subgoal $G$ is standardized for insertion into the subgoal table, the storage requirement for an answer $G\theta_A$ depends only on the size of $\theta_A$. If answer tables are implemented as tries then the following proposition will hold.

**Proposition 3.1.** Let $G$ be a subgoal and $A$ be an answer for $G$. Using substitution factoring both answer check/insert and answer backtracking can be performed in time proportional to the size of the answer substitution of $A$.

To illustrate this, consider a subgoal $p(f(X, Y), g(X))$ with an answer $p(f(a, b), g(a))$. In this case the answer has six symbols, whereas the substitution $\theta$ has only two symbols. For an access operation on an answer, either check/insert or return, using substitution factoring only two symbols are traversed as opposed to six.

In terms of related work, substitution factoring bears a certain resemblance to the factoring of [9] (hereafter termed NRSU-factoring) in that both reduce the number of arguments copied into or out of a table. However, substitution factoring has different characteristics than NRSU-factoring, mainly because it is a dynamic rather than static technique. Whether a predicate is NRSU-factorable is undecidable in general; hence NRSU-factoring is applicable only to certain classes of Datalog programs. Consequently, substitution factoring may reduce arguments of predicates that are not reduced by NRSU-factoring. Furthermore, contrary to NRSU-factoring, substitution factoring is applicable to and can be very effective for non-Datalog programs. On the other hand, [9] introduces additional optimizations based on the factored program which are not performed by substitution factoring. These optimizations can transform certain right and double recursions into left recursions, an important transformation not performed by substitution factoring.

### 4. Implementation aspects of tabling tries

Tabling tries are implemented by representing each state by a node, and transitions by pointers to nodes. The structures of subgoal and answer trie nodes are shown in Fig. 4 and are explained throughout this section.

The label on a transition is placed in the symbol field of the node representing the destination state. The outgoing transitions from a node are traced using its first child pointer and by following the list of sibling pointers of this child. Recall that in order to lookup or insert a term into a tabling trie, the term is traversed in preorder. If the symbol inspected in this traversal is the label of an outgoing transition from the current state, that transition is taken. Otherwise, a new destination state is created, and the transition to this state is taken. In the current implementation of tries in XSB, the matching outgoing transition is found using sequential search whenever the number
of outgoing transitions from this state is small, otherwise hashing is used. Note that in this case hashing is always on a single symbol so that it is easy to achieve good discrimination. Hash collisions are reduced by dynamically expanding the hash tables.

Recall that terms inserted in the trie are standardized. This standardization process is performed while a term is inserted in the trie. The variables in the term are replaced by their numbervar values, by binding the dereferenced variable cell to a unique number, and tagging the cell with a type tag that is not otherwise used by the SLG-WAM. Using this single binding, non-linearity (i.e., repeated occurrences of the same variable) is handled without the need to check whether a variable has been previously encountered. The bindings are undone as soon as the insertion of the term in the trie is complete. In this manner, the numbervar bijection can be performed in a single pass of the input term.

4.1. Implementation of substitution factoring

In order to explain the implementation of substitution factoring, we briefly consider the creation of SLG-WAM choice points for calls to a tabled predicate; full details can be found in [13]. As does the WAM, the SLG-WAM creates a choice point by copying the program registers at the time of the call, including registers containing each argument of the subgoal (argument registers). If the subgoal is new to the evaluation, a generator choice point (Fig. 5(a)) is created which will backtrack through program clauses. However, together with the arguments $A_1, \ldots, A_n$ of the tabled subgoal, a generator choice point also contains a substitution factor consisting of dereferenced pointers to unbound variables $V_1, \ldots, V_m$ of the subgoal (see Fig. 5(a)). These pointers are obtained during the call check/insert operation; after this operation is completed other choice point cells are placed above the substitution factor. If the subgoal has already been encountered during the evaluation, answer resolution will be used instead of program clause resolution. In this case, a consumer choice point (Fig. 5(b)) is created, which serves as an environment into which answers can be returned by means of an answer-return operation. Like the generator choice points, these consumer choice points contain a substitution factor. However because they are not used for program clause resolution, consumer choice points have no need for argument registers.

The referents of the substitution factor reside either in the local or global stack. Bindings to these referents are trailed through forward execution, whether they
are caused by program clause resolution (for a generator choice point) or answer clause resolution (for a consumer choice point). The values in the substitution factor variables are untrailing through backtracking just as argument cells would be in WAM execution. Trailing and untrailing in the SLG-WAM is beyond the scope of this paper and is explained in detail in [13]. When a new answer to a tabled subgoal is detected, the dereferenced values of the cells of the substitution factor from the generator choice point are copied directly into the table. Later, they will be loaded directly into the consumer choice points to return the answers. Fig. 6 shows an example of a tabling trie incorporating substitution factoring for answers to the subgoal \( p(f(X), g(Y)) \).

Subgoals:

<table>
<thead>
<tr>
<th>Subgoal</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p(X,X) )</td>
</tr>
<tr>
<td>( p(f(X),a) )</td>
</tr>
<tr>
<td>( p(f(X),g(Y)) )</td>
</tr>
<tr>
<td>( p(f(X),g(X)) )</td>
</tr>
</tbody>
</table>

Answer substitutions for \( p(f(X),g(Y)) \):

- \( X = a, Y = a \)
- \( X = a, Y = b \)
- \( X = b, Y = a \)
- \( X = b, Y = b \)

Fig. 6. Substitution factoring illustrated for the subgoal \( p(f(X), g(Y)) \).
4.2. Returning answers to consumer subgoals

Recall that in a tabling framework answers need to be returned to applicable consumer subgoals. Answer tries of subgoals for which new answers may be derived are termed incomplete (see [13]). Since answer insert and answer return operations can be interleaved, and new answers can be inserted anywhere in the trie, it is not possible to perform the answer-return operation by sequentially backtracking through such a trie starting from its root. Therefore, an explicit list of answers (uniquely identified by leaf nodes of the answer trie), has to be maintained. Alternatively, the list can be implemented by having the first child field of leaf answer nodes point to the next answer (as shown in Fig. 6). The order of this list reflects the creation times of its members. For example, in Fig. 6 the answers are created in the order \{X = b, Y = b\}, \{X = a, Y = b\}, and \{X = b, Y = a\}. Answers are returned for incomplete answer tries by traversing this list, and, with the help of two stacks, a term stack and a unification stack, constructing the answers by a leaf-to-root traversal. To efficiently perform this latter traversal, every node of the answer trie maintains a back pointer to its parent node (denoted as parent pointer in Fig. 4). The answer return operation starts by pushing the substitution factor variables (in reversed order) into the unification stack. Then starting from the leaf node and following the parent pointers, the symbols in the branch from the leaf to the root are pushed into the term stack. On reaching the root of the answer trie, the substitution factor variables in the unification stack are unified with the terms constructed on the term stack.

Note, however, that answers can be returned from a completed answer trie by sequentially backtracking from its root. Indeed, the WAM is a highly optimized engine for performing backtracking. To exploit this power of the WAM, we dynamically compile answer tries into WAM code as presented in Section 5. The idea of compiling dynamically created terms has been around for quite some time in logic programming languages; for example it is used in some implementations of Prolog's assert/1. Recently, this idea has also been used in the context of general theorem proving to efficiently perform forward subsumption (i.e. pattern matching) of terms that are dynamically created [19].

5. Dynamic compilation of tries

We describe how answer tries are dynamically compiled into WAM-like instructions, called trie instructions. We refer to the tries that consist of these instructions as compiled tries, and to those described in Section 4.2 as interpreted tries. 5

To motivate the new WAM instructions, we first show how an answer trie can be represented as regular Prolog clauses. We then consider how WAM-style instructions might implement those clauses, and finally we create "mega"-instructions that constitute a space-efficient representation of answer tries. We use the following example throughout the development. Assuming that substitution factoring is employed, consider the answer trie of a subgoal that contains three variables shown in Fig. 7.

---

5 The instructions presented in this section are slightly more general than needed for answer tries under variant tabling, and can be used to implement an alternative to assert/1 (see Section 6.5).
Fig. 7. An answer trie.

(a) As facts.

(b) As "prefix-factored" clauses into a full-trie.

The four answers in this trie could be represented by the Prolog facts of Fig. 8(a). Alternatively these facts could be prefix-factored into a full-trie [4] represented as a set of Prolog clauses shown in Fig. 8(b). Note that each clause in Fig. 8(b) corresponds to a single edge in the answer trie of Fig. 7, and that the order of the clauses reflects a breadth-first, left-to-right traversal of the edges of the trie.

---

6 For simplicity of presentation we consider linear answers first, and describe later on how non-linearity is handled.
We assume the existence of an array of registers and base the following discussion on two premises of Section 4. First, we assume that backtracking is only performed on completed tables, so that no answers will be added to a trie through which we are backtracking. Also, we assume that substitution factoring is performed. Operationally this means that when answer resolution is to be used for a subgoal with $n$ distinct variables, the first $n$ registers have been initialized to hold these variables. This initialization can be easily performed while traversing the subgoal in the subgoal trie.

Fig. 9 shows WAM-like code segments for the first three clauses of Fig. 8(b). The code for the first two clauses starts with a choice-point instruction, here a `try_me_else` or a `trust_me_else` instruction. The second instruction is a `get_type` instruction. The `shift_left` instruction is not contained in the WAM. Its function is to shift all the registers to the left by some number of positions (here one). This function is needed to set up the arguments for the final instruction, the execute, which branches to the next clause. Now consider the code for the first clause of `rt_a/2`, whose first argument is a structure. Here again the first instruction is a choice-point instruction, and the second instruction is a `get_structure` instruction. Now however, the `get_structure` is followed by a `shift_right` instruction which shifts the registers right to make room for the arguments of the structure symbol (the required number of positions is always one less than the arity of the structure symbol). The `shift_right` instruction is followed by an argument-construction instruction, `unify_variable` for each argument. Finally, there is again an execute instruction to branch to the next clause.

Code segments like the ones in Fig. 9 will construct the answer one-at-a-time on the stack through backtracking. For efficient implementation we coalesce these sequences of instructions into a single WAM-like instruction termed a `trie instruction`. The main reason for this coalescing is to reduce the space needed for the representation of these instructions in trie nodes (only one extra field needs to be added to the format of the answer trie nodes of Fig. 4); this action however also has a small time performance improvement. Note that there are following five major parameters to a code segment.

1. The `choice` alternative.
2. The `get_type` instruction, 7
3. The constant, structure symbol, or variable to match (Symbol). 8
4. The address of the next code segment down the trie (`ContLabel`),

---

7 If the instructions are intended to support tabling in which a variant-check is used for subgoals, `get_type` instructions can be replaced by `build_style` instructions.
8 Note that the argument register involved in the `get_type` instructions is always register 1 ($A_1$).
5. The address of the alternative to try on failure (FailLabel).

So, the general form of the trie instructions is:

\[
\text{trie\_choice\_type Symbol, ContLabel, FailLabel.}
\]

Note how the three arguments of this instruction naturally correspond to the three fields (Symbol, First Child, and Sibling) of the answer trie nodes of Fig. 4.

Since each of the trie instructions may appear as the first, an intermediate, the last, or the only instruction in a sequence of alternatives, we denote the choice possibilities as try, retry, trust, and do respectively. As for types, constants, structures, lists as well as uninstantiated (first occurrence) and instantiated (consequent occurrences of) variables should be handled. Fig. 10 presents the set of the trie instructions thus created. In the special case of answer tries whose subgoals have no variables, another trie instruction needs to be introduced, named trie_proceed which has exactly the functionality of the WAM’s proceed, namely setting the program register to the continuation register. The functionality of a proceed is also needed after the leaf of a trie is reached.

As a concrete example of how the trie instructions are used, Fig. 11 shows the WAM code generated for the clauses of Fig. 8(b), and, as a consequence, for the answer trie of Fig. 7. Since no choice points are laid down for the trie_do_? instructions, their FailLabel fields are not used. Notice the correspondence between the labels of the instructions in Fig. 11 and the names of the atoms in the clauses of Fig. 7. For facts, the trie instruction has a ContLabel of proceed to indicate that the final operation of the trie instruction should be that of a proceed WAM instruc-
tion rather than that of an execute. A slight optimization is to create specialized versions of trie instructions that encode the last operation as that of a proceed. Such instructions would be needed only for constants and variables.

The actions of the trie instructions are easily understandable if one thinks of them as macros that define WAM code segments like those of Fig. 9. For example, the three code segments of Fig. 9 present the operations performed (for some values of the parameters) by the trie_trv_constant, trie_trrust_constant, and trie_trv_structure instructions, respectively. We note that the shift_left and shift_right operations could be implemented efficiently in an engine that stores the registers as an array, simply by modifying the base of that array. Alternatively, a separate array of pseudo-registers could be used for the trie instructions only, which would allow it to perform efficiently as a register stack. The latter is the implementation scheme chosen by XSB. Non-linearity is handled by adding another array to the WAM, called the var-array. The trie_?_variable instructions initialize the indicated var-array entry on the heap, setting the top element of the register stack to point to it. The trie_?_value instructions then unify the top register of the register stack with the indicated var-array variable.

The trie instructions presented are used in XSB not only for answer tries but for asserted facts. If trie instructions were used only for tabling with variant-checks for subgoals, substitution factoring would allow all uses of the get_type subinstructions to directly bind their values, i.e. to run in write mode, rather than to perform unification. We also note that indexing is needed for the answer check/insert step as well as for asserted code. Accordingly, the set of trie instructions described in this section has been extended with two more hashing instructions to perform this indexing. While useful for not slowing down the answer check/insert step, the hashes do not provide any extra efficiency in answer backtracking once a subgoal is completed. Both the use of indexing and the provision of unification in get_type subinstructions slightly complicate the dynamic compilation, and impose a small performance overhead which would be avoidable if answer tries and asserted facts did not use the same compilation mechanisms.

We end this section by stating a useful property of compiled tries. This property is based on the observation that all common prefixes of the terms in a trie are shared during execution of trie instructions.

Property 5.1. When backtracking through the terms of a trie that is represented using the trie instructions, each edge of the trie is traversed only once.

6. Performance results

Several optimization methods have been presented so far: the use of tries, of substitution factoring, and of dynamically compiling tabled terms into WAM-like code. We first discuss the performance on tabled evaluations of each of these optimizations, and then the advantages of using trie-like code in creating facts dynamically through a mechanism similar to Prolog's assert/1.  

---

9 All benchmarks were run on a SparcStation 2 with 64 MB of main memory running SunOS 4.1.3. Sizes of the benchmark programs do not reflect limitations in any of the systems evaluated.
6.1. Trie-based vs. hash-based table structures in XSB

We first compare alternative tabling methods as they have been implemented in XSB. A hash-based method of XSB Version 1.4.0, and two different trie-based methods. The first trie-based method does not compile tries into instructions and was used in Version 1.4.0; the second method compiles tries, and is found in Versions 1.4.2 and later. Hash-based table structures have a simple form. Each tabled predicate has its own subgoal hash table. For the subgoal check/insert step, the subgoal is hashed and compared against any other subgoal in the hash bucket, using a variant-check. If the subgoal is not present, it is entered into the chain of the proper hash bucket. Each subgoal has its own answer hash table which resembles the subgoal hash tables in its essential details, and also requires a variant-check in the case of hash collisions. Subgoals are hashed on the outer functor symbol of their first argument, while answers are hashed on the combination of the outer functor symbols of all their arguments. Note that this latter method gives full indexing for Datalog terms. As a result, hash-based tabling consists of a quick insert, but a slow check if hash collisions occur. On the other hand, trie-based tabling consists of a relatively slower insert than the hash-based – it must set parent and sibling pointers – but combines the check and insert steps, and thereby may need to copy less information for answers. Substitution factoring has been implemented only in the trie-based methods, but its effect will be isolated in Section 6.2.

We begin by comparing the hash-based methods to the interpreted tries. The first set of tests use standard left (Fig. 12(a)), and right (Fig. 12(b)) recursive transitive closures. A Datalog binary tree was used as the EDB relation (shown in Fig. 13(a)). As an additional test, the tree was nested in a unary structure (Fig. 13(b)). Unification factoring [5] was used to compile the structured EDB. Unification factoring processes the heads of the p/2 clauses into a non-deterministic net which, in this case, provides perfect indexing. The graph of Fig. 14 shows times for 25 iterations of the queries \(- a(1, X) \) to non-structured EDB and that of Fig. 15 of queries \(- a(f(1), X) \) to structured EDB. In the graphs, Height refers to the height of the tree, while Trie and Hash indicate the use of trie and hash-based methods respectively. Left and Right stand for left and right-recursive definitions of transitive closure. We note that for queries of the form \(- a(\text{bound, free}) \) over complete binary trees, the left-recursive definition of transitive closure encounters (and generates answers for) only one distinct call, and thus has a better complexity than the right-recursive one where the number of calls encountered is equal to the size of the tree.

\[ a(X, Y) \leftarrow p(X, Y). \]
\[ a(X, Y) \leftarrow a(l, Z), p(Z, Y). \]
\[ a(X, Y) \leftarrow a(l, Z), p(Z, Y). \]

(a) & (b) & (c)

Fig. 12. (a) Left. (b) right. and (c) NRSU-factored recursive transitive closures.

\[ (a) \quad p(1, 2), p(1, 3), \ldots, p(2^n - 1, 2^{n+1} - 1) \]
\[ (b) \quad p(f(1), f(2)), p(f(1), f(3)), \ldots, p(f(2^n - 1), f(2^{n+1} - 1)) \]

Fig. 13. (a) Datalog, and (b) structured binary trees for the programs of Fig. 12.
The graphs in Figs. 14 and 15 indicate the power of tries. For the Datalog cases, and especially for left recursion, times for hash and tries are generally similar, with tries having a slight advantage for large data sets where the effect of hash collisions is more noticeable. However, as soon as discriminating information is nested within structures, the times for tries become far more efficient than those for hashing. This divergence is due to the trie's ability to effectively index subgoals and answers on constants within the symbol \( f/1 \) in the structured data, an ability not shared by hash-based tabling. This point is further substantiated in Section 6.2.

6.2. Measuring the effects of substitution factoring

In order to isolate the effect of substitution factoring, we statically factor a left recursive program (shown in Fig. 12(c)) in a manner similar to \( NRSU\)-factoring. Note
that given a query `?- a(free)`, the program of Fig. 12(c) will perform exactly the same subgoal check/insert, answer check/insert and answer backtracking operations as the program in Fig. 12(a) when substitution factoring is performed. Given the same \( p/2 \) relation as in the previous section, we would expect the trie-based engine with substitution factoring to exhibit no speedup, while the hash-based engine to exhibit a speedup due to substitution factoring. As expected, static factoring shows no speedups over dynamic factoring for the trie-based emulators in Tables 1 and 2 (rows labeled Trie Speedup). For the hash-based emulators, the effects are substantial, especially for the non-Datalog program (rows Hash Speedup). The effect of substitution factoring causes the times for the hash-based emulator to become identical to that of the trie-based emulator for Datalog programs (last row of Table 1). However, for non-Datalog programs the trie-based emulator is linear in the size of the binary tree while the hash-based emulator shows a marked quadratic factor (as shown by their comparison in last row of Table 2). Thus, with substitution factoring, the hash-based emulator is comparable to the trie-based emulator for the Datalog programs, but the ability of tries to discriminate information nested within a term is clearly important for structured data.

6.3. Compiled vs. interpreted tries

The preceding performance sections compare a hash-based implementation to a trie-based implementation without dynamic compilation. We now compare interpreted tries to compiled tries. Dynamic compilation of tries can be expected to improve the speed of answer backtracking, but to slow down the answer check/insert operation. (Since backtracking through tabled subgoals is never done in a pure tabled evaluation, the subgoal trie is never dynamically compiled.)

The effects of dynamic compilation on answer backtracking. The first two columns of Table 3 show times required to backtrack through various sets of dynamically compiled and interpreted tries.
Table 3
Times for accessing dynamically created terms of various forms

<table>
<thead>
<tr>
<th>Form and number of terms</th>
<th>Interpreted tries</th>
<th>Compiled tries</th>
<th>Asserted (XSB)</th>
<th>Asserted (Quintus)</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(i) 0 ≤ i ≤ 4K</td>
<td>4.95 3.71 3.71 7.10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p(a, i) 0 ≤ i ≤ 4K</td>
<td>6.5 3.76 4.91 7.85</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p(i, a), p(i, b) 0 ≤ i ≤ 2K</td>
<td>6.63 4.78 4.88 7.84</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p(i, 2..., 10) 0 ≤ i ≤ 4K</td>
<td>18.75 18.91 15.44 14.55</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p(i, 2..., 100) 0 ≤ i ≤ 200</td>
<td>7.68 8.39 6.41 4.35</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary tree Level 6</td>
<td>9.67 4.88 7.38 8.58</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary tree Level 7</td>
<td>24.07 9.07 16.23 19.10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p(f(i)) 0 ≤ i ≤ 4K</td>
<td>6.13 3.75 4.31 7.40</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>p(f(i, 2..., 10)) 0 ≤ i ≤ 4K</td>
<td>14.78 18.93 9.04 9.60</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We may define a common prefix measure for a set, S, of terms as

\[ 1 - \frac{\text{number of transitions in the trie for } S}{\text{sum of the sizes of the terms in } S}. \]

For Table 3 the common prefix measure ranges from about 91% (for the structured data in the second to last row), to no sharing at all (the unary, 10-ary, and 100-ary Datalog terms in rows 1, 4, and 5).

As expected, the performance of compiled tries increases with the common prefix measure. In the admittedly extreme case of \( p(f(...(f(i))) \) in the second to last row, compiled tries achieve speed-ups of 4 times over tries without code. However, when the common prefix measure is low, the performance of compiled tries is slightly slower than that of interpreted tries, especially for terms that contain structures. In the case of \( p(f(i, 2..., 10)) \) the slowdown is due to the fact that compiled tries effectively perform the transformation \( p(f(i, 2..., 10)) : = p(i, 2..., 10). \) So that variables within the \( f/10 \) structure lie below the last choice point. These variables must be present as cells within the choice point and must also be trailed. In contrast, the other methods recreate the \( f/10 \) structures on the heap. A second point is

---

10 All times in Table 3 represent 100 iterations except for the “Binary Tree” cases which represent 10,000 iterations.
that for a binary tree, compiled tries will execute about twice the number of choice point instructions as the other methods. (Compiled tries will execute a choice point instruction for every edge of the tree, while the other methods will execute an instruction for every leaf of the tree.) However this trade-off of choice points for binding generally seems to be beneficial, according to results in [5] for static code.

*The effects of dynamic compilation on answer check/insert.* Having compared the performance of accessing compiled and interpreted answers we next measure the time required for creating the trie data structures. Clearly creation time is a critical factor since the code generation phase is performed during query evaluation. Tables 4 and 5 present times for completing tables with and without the code generation phase using the left (Fig. 12(a)) and right recursive (Fig. 12(b)) transitive closure predicates on Datalog chains. As the times show, the extra code generation phase incurs only a minimal overhead (less than 5%) to the table creation process. We note that in these benchmarks no answers from completed tables are ever used; they thus provide an upper-bound of the actual cost of code generation. In cases where the derivation of answers for a table involves resolution with answers from other already completed tables, the overhead from code generation is usually balanced by the speedup in the time to access these answers.

### 6.4. Analysis of space requirements

In this section we analyze space usage on a practical example. In [11] it was shown that model checking of concurrent systems can be implemented using XSB’s tabling. Furthermore, it was shown that the resulting system is comparable in both time and space to systems that have been specially designed for model checking.

Table 6 compares either the number of trie nodes (in trie-based methods) or the summed term size of calls and answers (in the hash-based methods) using various table access methods. In particular, hash-based tables are compared to trie-based tables, both with and without substitution factoring. The programs analyzed are sieve, which traverses the states for a concurrent system in which a generator process and six tester processes communicate along a linear chain; and leader which verifies that a leader election algorithm will always choose a unique leader in a two process system. The information in Table 6 was obtained in two steps. The first step evaluated the queries in order to construct completed tables for leader and sieve.

<table>
<thead>
<tr>
<th>Length of chain</th>
<th>1K</th>
<th>2K</th>
<th>4K</th>
<th>8K</th>
<th>16K</th>
<th>32K</th>
<th>64K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreted tries</td>
<td>0.05</td>
<td>0.10</td>
<td>0.19</td>
<td>0.37</td>
<td>0.74</td>
<td>1.50</td>
<td>3.11</td>
</tr>
<tr>
<td>Compiled tries</td>
<td>0.05</td>
<td>0.11</td>
<td>0.21</td>
<td>0.40</td>
<td>0.81</td>
<td>1.56</td>
<td>3.18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Length of chain</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>768</th>
<th>1K</th>
<th>1.5K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreted tries</td>
<td>0.13</td>
<td>0.62</td>
<td>2.70</td>
<td>6.34</td>
<td>11.25</td>
<td>26.16</td>
</tr>
<tr>
<td>Compiled tries</td>
<td>0.18</td>
<td>0.66</td>
<td>2.73</td>
<td>6.36</td>
<td>11.31</td>
<td>27.19</td>
</tr>
</tbody>
</table>
As presented in [11], a state of a concurrent system can be represented as a logical term. Such a term may be lengthy, but "similar" states may share a common prefix when represented as terms. Table 6 reflects this sharing through the size reduction of the trie-based methods over the hash-based methods. In leader, highly instantiated tabled subgoals are called, so that substitution factoring provides a significant reduction in space requirements for hashing. Much of the instantiated portion of these subgoals, however, occurs in their leftmost prefix. As a result, substitution factoring leads to smaller space savings for the tries, since the leftmost prefix is factored into the top of a trie. However, if substitution factoring is not used, the top of a trie will need to be traversed at each answer check/insert operation and each answer backtracking operation, so that substitution factoring has a beneficial effect on the execution time of leader (this effect is not measured in this section).

Table 6 measures the sizes of hashed terms and of tries, but does not indicate how much space the tables will use in a functioning system. To obtain this information, indexing must be taken into account, along with the actual space requirements for terms which may vary according to whether the terms are compiled or interpreted. Disregarding index sizes for a moment, the actual space requirements of the terms themselves can be easily approximated using the following assumptions. We assume that each constant, variable or function symbol of hashed term requires 1 word when interpreted. When hashed answer tables are compiled, we assume that two words are required per symbol (as in the WAM). We further note that interpreted tries require four words per node and that compiled answer tries require five words per node. Table 7 indicates the approximate space requirements, in words, for the various tabling methods on model-checking examples.

Table 7 indicates that (interpreted) tries with substitution factoring give the best space utilization for storage of tabled subgoals and answers, disregarding indexing. Somewhat surprisingly, however, the tries require almost no space for indexing as measured via hashing instructions (as defined in Section 5) – in XSB only 16 words are required over both examples. It can be expected that hash-based methods will require far more index space for even moderate discrimination of terms, so at least for this example, tries outperform hash-based methods in terms of space.

<table>
<thead>
<tr>
<th>Table access method</th>
<th>Sieve</th>
<th>Leader</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of calls</td>
<td>1</td>
<td>2 022</td>
</tr>
<tr>
<td>Number of returns</td>
<td>3 089</td>
<td>3 083</td>
</tr>
<tr>
<td>Size of calls (hashing)</td>
<td>4</td>
<td>214 873</td>
</tr>
<tr>
<td>Size of returns (hashing, no substitution factoring)</td>
<td>235 224</td>
<td>641 818</td>
</tr>
<tr>
<td>Size of returns (hashing, substitution factoring)</td>
<td>225 957</td>
<td>324 648</td>
</tr>
<tr>
<td>Size of calls (tries)</td>
<td>4</td>
<td>62 216</td>
</tr>
<tr>
<td>Size of returns (tries, no substitution factoring)</td>
<td>63 347</td>
<td>62 625</td>
</tr>
<tr>
<td>Size of returns (tries, substitution factoring)</td>
<td>63 343</td>
<td>58 740</td>
</tr>
</tbody>
</table>
Table 7
Approximate space requirements in words, for various table access configurations (not including indexing space)

<table>
<thead>
<tr>
<th>Table access method</th>
<th>sieve</th>
<th>leader</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreted hashing, no substitution factoring</td>
<td>235 228</td>
<td>856 691</td>
</tr>
<tr>
<td>Interpreted hashing, substitution factoring</td>
<td>225 961</td>
<td>539 521</td>
</tr>
<tr>
<td>Compiled hashing, no substitution factoring</td>
<td>468 452</td>
<td>1 498 509</td>
</tr>
<tr>
<td>Compiled hashing, substitution factoring</td>
<td>451 918</td>
<td>864 169</td>
</tr>
<tr>
<td>Interpreted tries, no substitution factoring</td>
<td>253 404</td>
<td>499 364</td>
</tr>
<tr>
<td>Interpreted tries, substitution factoring</td>
<td>253 388</td>
<td>483 824</td>
</tr>
<tr>
<td>Compiled tries, no substitution factoring</td>
<td>316 755</td>
<td>619 205</td>
</tr>
<tr>
<td>Compiled tries, substitution factoring</td>
<td>316 735</td>
<td>542 564</td>
</tr>
</tbody>
</table>

6.5. Tries for asserted terms

Compared to asserted code, compiled tries provide good speed for answer backtracking as presented in Section 6.3. They can also utilize space well compared to compiled hash-based methods as shown in the previous section. When unit clauses are dynamically compiled and asserted, their internal representation resembles that of hashed, compiled, answer clauses. It is thus natural to explore the use of tries to store dynamically created facts outside of tabling.

As a last set of benchmarks, we compare the time needed to assert a set of terms (using Prolog’s `assert/1`) with the time needed to create them as compiled tries.

Table 8
Creation times for unary Datalog data \((p(i), 1 \leq i \leq \text{Size})\)

<table>
<thead>
<tr>
<th>Size</th>
<th>4K</th>
<th>5K</th>
<th>6K</th>
<th>7K</th>
<th>8K</th>
<th>9K</th>
<th>10K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asserted code (XSB)</td>
<td>1.51</td>
<td>1.98</td>
<td>2.35</td>
<td>2.84</td>
<td>3.18</td>
<td>3.64</td>
<td>3.96</td>
</tr>
<tr>
<td>Compiled tries (XSB)</td>
<td>0.10</td>
<td>0.12</td>
<td>0.15</td>
<td>0.17</td>
<td>0.21</td>
<td>0.25</td>
<td>0.28</td>
</tr>
<tr>
<td>Asserted code (Quintus)</td>
<td>1.73</td>
<td>2.15</td>
<td>2.58</td>
<td>3.01</td>
<td>3.50</td>
<td>3.86</td>
<td>4.35</td>
</tr>
</tbody>
</table>

Table 9
Creation times for 10-ary Datalog data \((p(i,2,\ldots,10), 1 \leq i \leq \text{Size})\)

<table>
<thead>
<tr>
<th>Size</th>
<th>4K</th>
<th>5K</th>
<th>6K</th>
<th>7K</th>
<th>8K</th>
<th>9K</th>
<th>10K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asserted code (XSB)</td>
<td>2.45</td>
<td>3.30</td>
<td>4.08</td>
<td>4.95</td>
<td>6.06</td>
<td>6.85</td>
<td>7.86</td>
</tr>
<tr>
<td>Compiled tries (XSB)</td>
<td>0.24</td>
<td>0.33</td>
<td>0.41</td>
<td>0.49</td>
<td>0.52</td>
<td>0.66</td>
<td>0.76</td>
</tr>
<tr>
<td>Asserted code (Quintus)</td>
<td>1.85</td>
<td>2.28</td>
<td>2.66</td>
<td>3.12</td>
<td>3.72</td>
<td>4.17</td>
<td>4.48</td>
</tr>
</tbody>
</table>

Table 10
Creation times for unary structured data \((p(f(i)), 1 \leq i \leq \text{Size})\)

<table>
<thead>
<tr>
<th>Size</th>
<th>4K</th>
<th>5K</th>
<th>6K</th>
<th>7K</th>
<th>8K</th>
<th>9K</th>
<th>10K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asserted code (XSB)</td>
<td>8.00</td>
<td>12.44</td>
<td>18.36</td>
<td>24.09</td>
<td>31.40</td>
<td>39.45</td>
<td>49.02</td>
</tr>
<tr>
<td>Compiled tries (XSB)</td>
<td>0.15</td>
<td>0.19</td>
<td>0.21</td>
<td>0.26</td>
<td>0.28</td>
<td>0.34</td>
<td>0.35</td>
</tr>
<tr>
<td>Asserted code (Quintus)</td>
<td>8.62</td>
<td>13.38</td>
<td>18.67</td>
<td>25.00</td>
<td>31.22</td>
<td>39.48</td>
<td>48.25</td>
</tr>
</tbody>
</table>
Tables 8 and 9 present times to create unary and 10-ary Datalog facts. In addition, Table 10 shows times to create a unary fact used in Table 8 when its argument is nested in a unary function symbol.

As shown in Tables 8 and 9, storing terms as code in trie-based answer tables is about 10–20 times faster than using Prolog's assert/1. Note that all these terms are perfectly indexed on their first argument. As soon as the discriminating information is nested within structures and hash collisions start to occur with the use of assert/1, storing the terms in the trie-based table structures exhibits an even bigger performance improvement. Table 10 shows that the use of tables for storing dynamic terms in the presence of hash collisions is faster than assert by two orders of magnitude. Similar results were obtained in BIMprolog release 4.1.0. Given the competitive retrieval speed of tries, their complete discrimination, and their superior creation time, they are a useful alternative to asserted code for sets of dynamic data when the order of the terms in the sets need not be preserved. Because of these advantages, dynamic unit clauses can be asserted in XSB (Version 1.7 and later) using either conventional assert/1 or assert/1 using trie-based data structures. The choice is specified on a predicate basis, by using a directive such as :- index(p/1, trie). Dynamic code asserted using trie-based data structures can be retracted or abolished just as with conventional dynamic code using Prolog's retract/1 or abolish/1. Execution of asserted code uses the same instructions as answer backtracking in completed tries.

7. Discussion

The trie-based approach with which we address the table access problem has important properties in its ability to index data of different forms, and in its single pass check/insert operation. When extended with substitution factoring this approach provides dynamic argument reduction, and indeed, reductions within complex terms. Further, when tries are dynamically compiled, their access time and space usage compares well with WAM code, and the amount of binding on backtracking can in some cases be greatly reduced.

This approach reflects the dynamic nature of subgoal and answer creation, a characteristic which distinguishes the results of this article from other recent work. Fundamentally, tabling tries must partition dynamically changing sets of terms. In contrast, the unification factoring automata of [5] compiled a static set of program clause heads into a trie-like structure for which optimality properties were proven. Finally, as mentioned in Section 3, both the dynamic nature of substitution factoring and its applicability to non-Datalog programs separates it from static methods such as NRSU factoring.

As mentioned earlier, our tabling tries are variants of discrimination nets. In particular, the call and incomplete answer tries can be viewed as discrimination nets over ground terms. However, the relationship between a completed answer trie and a discrimination net is a little subtle. First, our completed answer tries are compiled whereas traditionally discrimination nets have been interpreted. Secondly, our completed tries perform unification operations (in order to implement asserted code) whereas discrimination nets do match operations.
Our work is orthogonal to that reported in [13], which described the SLG-WAM as a whole, but did not examine table access mechanisms and substitution factoring in depth, or consider compiled tries. While our approach has been developed for the XSB system, we believe that tabling tries and substitution factoring may also prove useful to other systems that already have or will incorporate some sort of tabling.

The concept of trie data structures has been around for a while. In fact, it is the data structure of choice in high performance automated theorem provers and term rewriting systems. However seamless adaptation of tries to a WAM engine through development of techniques for a tight integration (such as substitution factoring, dynamic compilation) collectively distinguishes our implementation from those used in the above areas.

Little else has been published concerning algorithms for table access, although [16,18] describe structure-sharing algorithms for tabling in the context of an evaluation engine. While useful bounds can be derived for the amount of copying needed by a structure-sharing approach, such approaches may be subject to high constant overheads, and in any case do not appear suitable for a WAM-based implementation. In general, implementing logic as needed by deductive databases is a difficult task, and one for which a complete solution – that evaluates in-memory queries as well as a programming language, and queries to disk-resident data as well as a database system – is not yet at hand. Under various guises, the table access problem is central to deductive databases. The performance of the trie-based approach gives reason to expect that it will form a part of future tabled logic programming systems and deductive databases as it does in present versions of XSB.

Acknowledgements

The authors thank C.R. Ramakrishnan for his help in the preparation and proof-reading of this article, and the anonymous reviewers for their many helpful comments. This research was supported in part by NSF grants CCR 9711386, 9705998, 9702681, 9510072, 9404921, CDA 9303181, INT 9600598 and 9314412, and by a fellowship from the K.U. Leuven Research Council.

References