Global Storing Mechanisms for Tabled Evaluation

Jorge Costa and Ricardo Rocha *

DCC-FC & CRACS
University of Porto, Portugal
c0607002@alunos.dcc.fc.up.pt ricroc@dcc.fc.up.pt

Abstract. Arguably, the most successful data structure for tabling is tries. However, while tries are very efficient for variant based tabled evaluation, they are limited in their ability to recognize and represent repeated terms in different tabled calls or/and answers. In this paper, we propose a new design for the table space where tabled terms are stored in a common global trie instead of being spread over several different tries.

1 Introduction

Tabling is an implementation technique where intermediate answers for subgoals are stored and then reused whenever a repeated call appears. The performance of tabled evaluation largely depends on the implementation of the table space – being called very often, fast lookup and insertion capabilities are mandatory. Applications can make millions of different calls, hence compactness is also required. Arguably, the most successful data structure for tabling is *tries* [1].

However, while tries are very efficient for variant based tabled evaluation, they are limited in their ability to recognize and represent repeated terms in different tabled calls or/and answers. In [2], Rao *et al.* proposed a *Dynamic Threaded Sequential Automata* (DTSA) that recognizes reusable subcomputations for subsumption based tabling. In [3], Johnson *et al.* proposed an alternative to DTSA, called *Time-Stamped Trie* (TST), which not only maintains the time efficiency of the DTSA but has better space efficiency.

In this paper, we propose a different approach. We propose a new design for the table space where all terms in a tabled subgoal call or/and answer are stored in a *common global trie* instead of being spread over several different trie data structures. Our approach resembles the *hash-consing* technique [4], as it tries to share data that is structurally equal. An obvious goal is to save memory usage by reducing redundancy in term representation to a minimum. We will focus our discussion on a concrete implementation, the YapTab system [5], but our proposals can be easy generalized and applied to other tabling systems.

* This work has been partially supported by the research projects STAMPA (PTDC/EIA/67738/2006) and JEDI (PTDC/ EIA/66924/2006) and by Fundação para a Ciência e Tecnologia.

2 Table Space

A trie is a tree structure where each different path through the trie data units, the *trie nodes*, corresponds to a term. Each root-to-leaf path represents a term described by the tokens labelling the nodes traversed. Two terms with common prefixes will branch off from each other at the first distinguishing token. For example, the tokenized form of the term $p(X, q(Y, X), Z)$ is the stream of 6 tokens: $p/3, VAR_0, q/2, VAR_1, VAR_0, VAR_2$. Variables are represented using the formalism proposed by Bachmair *et al.* [6], where the set of variables in a term is mapped to the sequence of constants $VAR_0, ..., VAR_N$.

Internally, the trie nodes are 4-field data structures. One field stores the node's token, one second field stores a pointer to the node's first child, a third field stores a pointer to the node's parent and a fourth field stores a pointer to the node's next sibling. Each node's outgoing transitions may be determined by following the child pointer to the first child node and, from there, continuing through the list of sibling pointers. A threshold value controls whether to dynamically index the sibling nodes through a hash table. Further, hash collisions are reduced by dynamically expanding the hash tables. YapTab implements tables using two levels of tries - one for subgoal calls, the other for computed answers. More specifically, the table space of YapTab is organized in the following way:

- each tabled predicate has a *table entry* data structure assigned to it, acting as the entry point for the predicate's *subgoal trie*.
- each different subgoal call is represented as a unique path in the subgoal trie, starting at the predicate's table entry and ending in a *subgoal frame* data structure, with the argument terms being stored within the path's nodes.
- the *subgoal frame* data structure acts as an entry point to the *answer trie*.
- each different subgoal answer is represented as a unique path in the *answer trie*. To increase performance, answer trie paths enforce the *substitution factoring* mechanism [1] and hold just the substitution terms for the free variables which exist in the argument terms.
- the subgoal frame has internal pointers to the first and last answer on the trie and the leaf's child pointer of answers are used to point to the next available answer, a feature that enables answer recovery in insertion time order. Answers are loaded by traversing the answer trie nodes bottom-up.

An example for a tabled predicate `t/2` is shown in Fig. 1. Initially, the subgoal trie is empty. Then, subgoal `t(a(1),X)` is called and three trie nodes are inserted: one for the functor `a/1`, a second for the constant `1` and one last for variable `X`. The subgoal frame is inserted as a leaf, waiting for the answers. Next, subgoal `t(a(2),X)` is also called. It shares one common node with `t(a(1),X)` but, having `a/1` a different argument, two new trie nodes and a new subgoal frame are inserted. At the end, the answers for each subgoal are stored in the corresponding answer trie as their values are computed. Note that, for this particular example, the completed answer trie for both subgoal calls is exactly the same.

3 Global Trie

We next describe the YapTab's new design for the table space. In this new design, all terms in a tabled subgoal call or/and answer are now stored in a common global trie (GT) instead of being spread over several different trie data structures. The GT data structure still is a tree structure where each different path through the trie nodes corresponds to a term. However, here a term can end at any internal trie node and not necessarily at a leaf trie node.

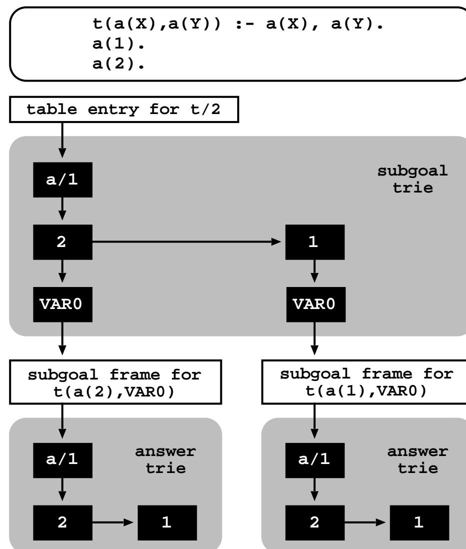

Fig. 1. YapTab's original table organization

The previous subgoal trie and answer trie data structures are now represented by a unique level of trie nodes that point to the corresponding terms in the GT (see Fig. 2 for details). For the subgoal tries, each node now represents a different subgoal call where the node's token is the pointer to the unique path in the GT that represents the argument terms for the subgoal call. The organization used in the subgoal tries to maintain the list of sibling nodes and to access the corresponding subgoal frames remains unaltered. For the answer tries, each node now represents a different subgoal answer where the node's token is the pointer to the unique path in the GT that represents the substitution terms for the free variables which exist in the argument terms. The organization used in the answer tries to maintain the list of sibling nodes and to enable answer recovery in insertion time order remains unaltered. With this organization, answers are now loaded by following the pointer in the node's token and then by traversing the corresponding GT's nodes bottom-up.

Figure 2 uses again the example from Fig. 1 to illustrate how the GT's design works. Initially, the subgoal trie and the GT are empty. Then, the first subgoal `t(a(1),X)` is called and three nodes are inserted on the GT: one to represent the functor `a/1`, another for the constant `1` and a last one for variable `X`. Next, a node representing the path inserted on the GT is stored in the subgoal trie (node labeled `call1`). The token field for the `call1` node is made to point to the leaf node of the GT's inserted path and the child field is made to point to a new subgoal frame. For the second subgoal call, `t(a(2),X)`, we start again by inserting the call in the GT and then we store a node in the subgoal trie (node labeled `call2`) to represent the path inserted on the GT.

For each subgoal call we have two answers: the terms `a(1)` and `a(2)`. However, as these terms are already represented on the GT, we need to store only two nodes, in each answer trie, to represent them (nodes labeled `answer1` and `answer2`). The token field for these answer trie nodes are made to point to the corresponding term representation on the GT. With this example we can see that terms in the GT can end at any internal trie node (and not necessarily at a leaf trie node) and that a common path on the GT can simultaneously represent different subgoal and answer terms.

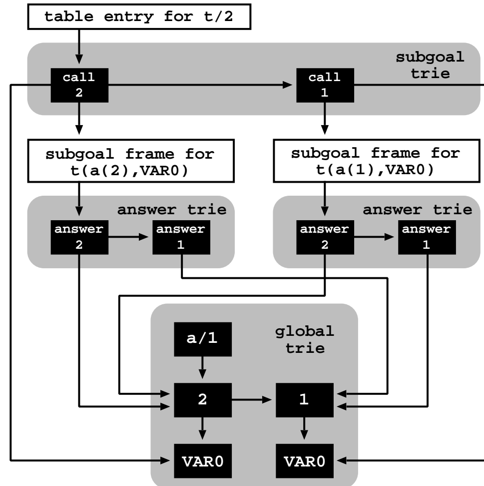

Fig. 2. YapTab's new table organization

4 Preliminary Experimental Results

To evaluate the impact of our proposal, we have defined a tabled predicate `t/5` that stores in the table space terms of a certain kind, and then we use a top query goal `test/0` that recursively calls `t/5` with all combinations of one and two free variables in the arguments. We next show the code example used in the experiments for functor terms of arity 1 (500 terms in total).

```
t(A,B,C,D,E) :- term(A), term(B), term(C), term(D), term(E).

test :- t(A,f(1),f(1),f(1),f(1)), fail.        term(f(1)).
...                                             term(f(2)).
test :- t(A,B,f(1),f(1),f(1)), fail.           ...
...                                             term(f(499)).
test.                                           term(f(500)).
```

The environment for our experiments was an AMD Athlon XP 2800+ with 1 GByte of main memory and running the Linux kernel 2.6.24-19. Table 1 shows the memory usage and the running times to store to the tables (first execution) and to load from the tables (second execution) the complete set of subgoals/answers for YapTab with and without support for the global trie data structure. We tested 5 different programs with functor terms of arity 1 to 5.

The results show that GT support can significantly reduce memory usage proportionally to the depth and redundancy of the terms stored in the GT. On the other hand, the results indicate that this reduction comes at a price in execution time. With GT support, we need to navigate in two tries when

Table 1. Memory usage (in KBytes) and store/load times (in milliseconds) for YapTab with and without support for the global trie data structure

Terms	***YapTab (a)***			***YapTab+GT (b)***			***Ratio (b)/(a)***		
	Mem	***Store***	***Load***	***Mem***	***Store***	***Load***	***Mem***	***Store***	***Load***
500 f/1	49172	693	242	52811	1029	243	1.07	1.48	1.00
500 f/2	98147	842	314	56725	1298	310	**0.58**	1.54	**0.99**
500 f/3	147122	1098	377	60640	1562	378	**0.41**	1.42	1.00
500 f/4	196097	1258	512	64554	1794	435	**0.33**	1.43	**0.85**
500 f/5	245072	1418	691	68469	2051	619	**0.28**	1.45	**0.90**

checking/inserting a term. Moreover, in some situations, the cost of inserting a new term in an empty/small trie can be less than the cost of navigating in the GT, even when the term is already stored in the GT. However, our results seem to suggest that this cost also decreases proportionally to the depth and redundancy of the terms stored in the GT. The results obtained for loading terms do not suggest significant differences. However and surprisingly, the GT approach showed to outperform the original YapTab design in some experiments.

5 Conclusions

We have presented a new design for the table space that uses a common global trie to store terms in tabled subgoal calls and answers. Our preliminary experiments showed very significant reductions on memory usage. This is an important result that we plan to apply to real-world applications that pose many subgoal queries with a large number of redundant answers, such as ILP applications.

References

1. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38**(1) (1999) 31–54
2. Rao, P., Ramakrishnan, C.R., Ramakrishnan, I.V.: A Thread in Time Saves Tabling Time. In: Joint International Conference and Symposium on Logic Programming, The MIT Press (1996) 112–126
3. Johnson, E., Ramakrishnan, C.R., Ramakrishnan, I.V., Rao, P.: A Space Efficient Engine for Subsumption-Based Tabled Evaluation of Logic Programs. In: Fuji International Symposium on Functional and Logic Programming. Number 1722 in LNCS, Springer-Verlag (1999) 284–300
4. Goto, E.: Monocopy and Associative Algorithms in Extended Lisp. Technical Report TR 74-03, University of Tokyo (1974)
5. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction. (2000) 77–87
6. Bachmair, L., Chen, T., Ramakrishnan, I.V.: Associative Commutative Discrimination Nets. In: International Joint Conference on Theory and Practice of Software Development. Number 668 in LNCS, Springer-Verlag (1993) 61–74