Tabling Logic Programs in a Common Global Trie

Jorge Costa and Ricardo Rocha

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

Abstract. The performance of tabled evaluation largely depends on the implementation of the table space. Arguably, the most successful data structure for tabling is tries. However, while tries are efficient for variant based tabled evaluation, they are limited in their ability to recognize and represent repeated answers for different calls. In this paper, we propose a new design for the table space where terms in a tabled subgoal call or/and answer are stored in a common global trie instead of being spread over several different tries. Our preliminary experiments using the YapTab tabling system show very promising reductions on memory usage.

Keywords: Tabling, Table Space, Implementation.

1 Introduction

Tabling [1–3] 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* [4]. Tries meet the previously enumerated criteria of efficiency and compactness.

Used in applications that pose many queries, possibly with a large number of answers, tabling can build arbitrarily many and/or very large tables, quickly filling up memory. A possible solution for this problem is to dynamically abolish some of the tables. This can be done using explicit tabling primitives or using a memory management strategy that automatically recovers space among the least recently used tables when memory runs out [5]. An alternative approach is to store tables externally in a relational database management system and then reload them back only when necessary [6].

A complementary approach to the previous problem is to study how less redundant, more compact and more efficient data structures can be used to better represent the table space. While tries are efficient for variant based tabled evaluation, they are limited in their ability to recognize and represent repeated

answers for different calls. In [7], Rao *et al.* proposed a table organization using *Dynamic Threaded Sequential Automata* (DTSA) which recognizes reusable subcomputations for subsumption based tabling. In [8], 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 [9], 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 [10, 11], but our proposals can be easy generalized and applied to other tabling systems.

The remainder of the paper is organized as follows. First, we briefly introduce some background concepts about tries and the table space. Next, we describe YapTab's new design for the table space organization using the common global trie and then, we describe how we have extended YapTab to provide engine support for our approach. At last, we present some preliminary experimental results and we end by outlining some conclusions.

2 Table Space

The basic idea behind tabling is straightforward: programs are evaluated by storing answers for tabled subgoals in an appropriate data space, called the *table space*. Whenever a repeated tabled call is found, the subgoal's answers are recalled from the table space instead of being re-evaluated against the program clauses. The table space may be accessed in a number of ways: **(i)** to find out if a subgoal is in the table and, if not, insert it; **(ii)** to verify whether a newly found answer is already in the table and, if not, insert it; and **(iii)** to load answers to variant subgoals. With these requirements, YapTab implements its table space using *tries* [12] which is regarded a very efficient way to implement tables [4].

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.* [13], where each variable in a term is represented as a distinct constant. Formally, this corresponds to a function, $numbervar()$, from the set of variables in a term t to the sequence of constants $VAR_0, ..., VAR_N$, such that $numbervar(X) < numbervar(Y)$ if X is encountered before Y in the left-to-right traversal of t.

Internally, the trie nodes are 4-field data structures. The first field stores the node's token, the second field stores a pointer to the node's first child, the third field stores a pointer to the node's parent and the 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. To increase performance, YapTab enforces the *substitution factoring* [4] mechanism and 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. Oppositely to subgoal tries, answer trie paths hold just the substitution terms for the free variables which exist in the argument terms of the corresponding subgoal call.
- the leaf's child pointer of answers is used to point to the next available answer, a feature that enables answer recovery in insertion order. The subgoal frame has internal pointers that point respectively to the first and last answer on the trie. Whenever a variant subgoal starts consuming answers, it sets a pointer to the first leaf node. To consume the remaining answers, it must follow the leaf's linked list, setting the pointer as it consumes answers along the way. Answers are loaded by traversing the answer trie nodes bottom-up.

An example for a tabled predicate `t/2` is shown in Figure 1. Initially, the subgoal trie is empty. Then, the 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, the 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 Common Global Trie

We next describe YapTab's new design for the table space organization. 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.

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

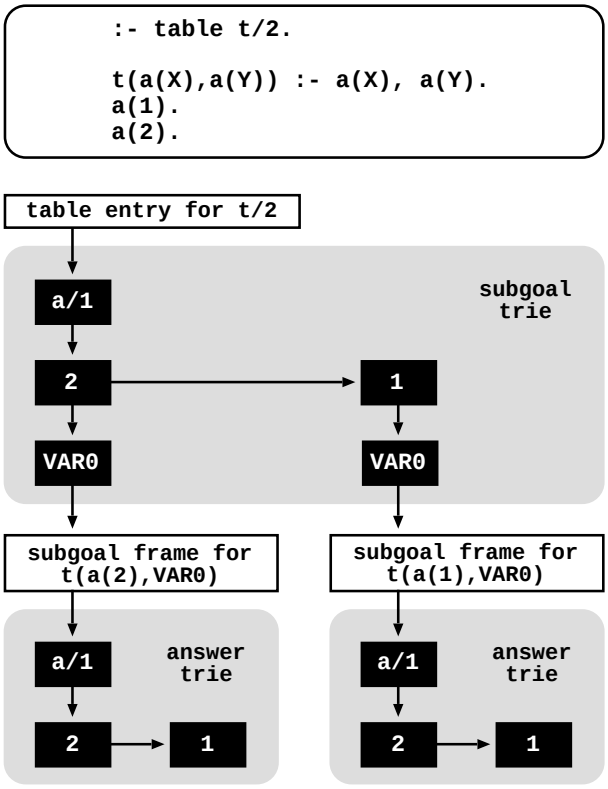

Fig. 1. YapTab's original table design

(see Figure 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 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.

On completion of a subgoal, a strategy exists that avoids answer recovery using bottom-up unification and performs instead what is called a *completed table optimization* [4]. This optimization implements answer recovery by top-down traversing the completed answer trie and by executing specific WAM-like code from the answer trie nodes. With our new design, the nodes in the GT can

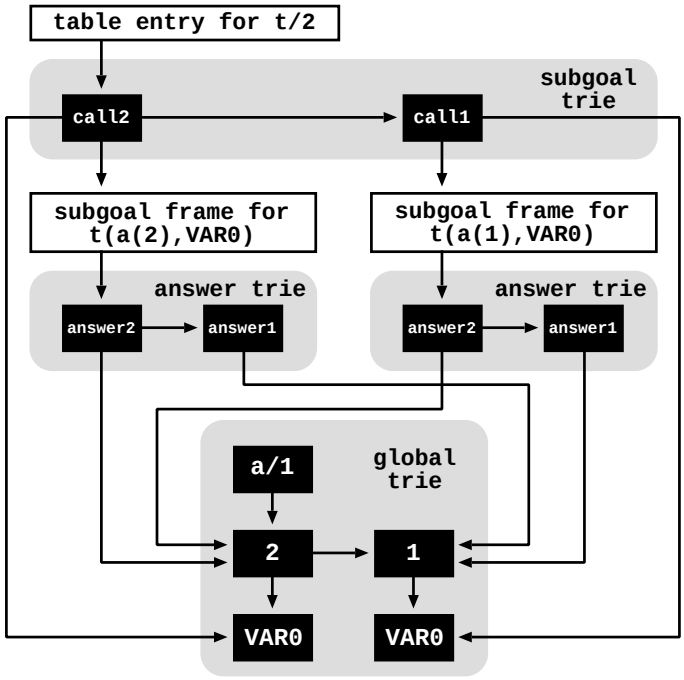

Fig. 2. YapTab's new table design

belong to several different subgoal/answer tries, and thus this optimization is no longer possible.

Figure 2 uses again the example from Figure 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.

As we saw in the previous example, 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.

4 Implementation Details

We then describe in more detail the data structures and algorithms for YapTab's new table design based on the GT. We start with Figure 3 showing in more detail the table organization previously presented in Figure 2.

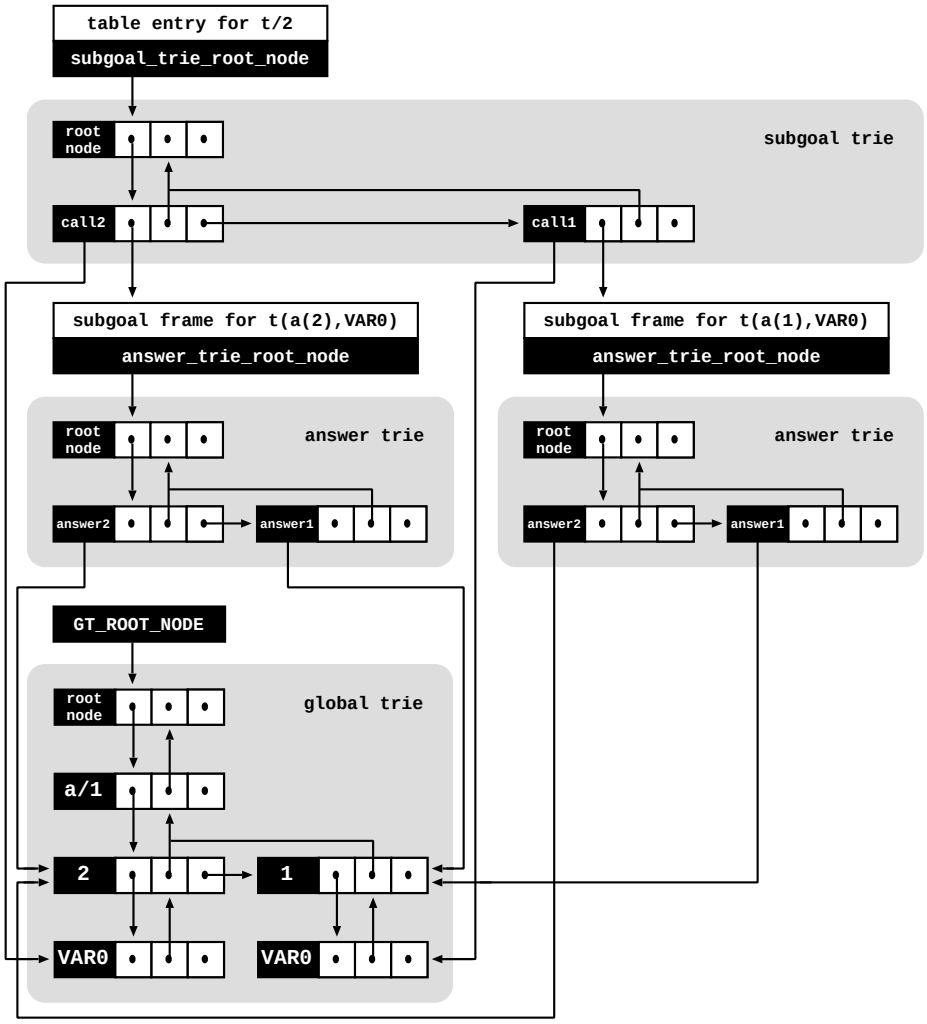

Fig. 3. Implementation details for YapTab's new table design

Internally, tries are represented by a top *root node*, acting as the entry point for the corresponding subgoal, answer or global trie data structure. For the subgoal tries, the root node is stored in the corresponding table entry's

subgoal_trie_root_node data field. For the answer tries, the root node is stored in the corresponding subgoal frame's answer_trie_root_node data field. For the global trie, the root node is stored in the GT_ROOT_NODE global variable.

Regarding the trie nodes, remember that they are internally implemented as 4-field data structures. The first field (token) stores the token for the node and the second (child), third (parent) and fourth (sibling) fields store pointers, respectively, to the first child node, to the parent node, and to the sibling node.

Traversing a trie to check/insert for new calls or for new answers is implemented by repeatedly invoking a trie_node_check_insert() procedure for each token that represents the call/answer being checked. Given a trie node parent and a token t, the trie_node_check_insert() procedure returns the child node of parent that represents the given token t. Figure 4 shows the pseudo-code for this procedure.

```
trie_node_check_insert(TRIE_NODE parent, TOKEN t) {
  child = parent->child
  if (child == NULL) {              // the list of sibling nodes is empty
    child = new_trie_node(t, NULL, parent, NULL)
    parent->child = child
  } if (is_not_a_hash_table(child)) {    // sibling nodes without hashing
    sibling_nodes = 0             // to count the number of sibling nodes
    do {           // check if token t is already in the list of siblings
      if (child->token == t)
        return child
      sibling_nodes++
      child = child->sibling
    } while (child)
    child = new_trie_node(t, NULL, parent, parent->child)
    if (sibling_nodes > MAX_SIBLING_NODES_PER_LEVEL) {  // alloc new hash
      hash = new_hash_table(child)
      parent->child = hash
    } else
      parent->child = child
  } else {                                    // sibling nodes with hashing
    hash = child
    bucket = hash_function(hash, t)    // get the hash bucket for token t
    child = bucket
    sibling_nodes = 0
    while (child) {     // check if token t is already in the hash bucket
      if (child->token == t)
        return child
      sibling_nodes++
      child = child->sibling
    }
    child = new_trie_node(t, NULL, parent, bucket)
    if (sibling_nodes > MAX_SIBLING_NODES_PER_BUCKET)      // expand hash
      expand_hash_table(hash)
  }
  return child
}
```

Fig. 4. Pseudo-code for the trie_node_check_insert() procedure

Initially, the procedure checks if the list of sibling nodes is empty. If this is the case, a new trie node representing the given token `t` is initialized and inserted as the first child of the given `parent` node. To initialize new trie nodes, we use a `new_trie_node()` procedure with four arguments, each one corresponding to the initial values to be stored respectively in the `token`, `child`, `parent` and `sibling` fields of the new trie node.

Otherwise, if the list of sibling nodes is not empty, the procedure checks if they are being indexed through a hash table. Searching through a list of sibling nodes is initially done sequentially. This could be too expensive if we have hundreds of siblings. A threshold value (`MAX_SIBLING_NODES_PER_LEVEL`) controls whether to dynamically index the nodes through a hash table, hence providing direct node access and optimizing search. Further hash collisions are reduced by dynamically expanding the hash tables when a second threshold value (`MAX_SIBLING_NODES_PER_BUCKET`) is reached for a particular hash bucket.

If not using hashing, the procedure then traverses sequentially the list of sibling nodes and checks for one representing the given token `t`. If such a node is found then execution is stopped and the node returned. Otherwise, a new trie node is initialized and inserted in the beginning of the list. If reaching the threshold value `MAX_SIBLING_NODES_PER_LEVEL`, a new hash table is initialized and inserted as the first child of the given `parent` node.

If using hashing, the procedure first calculates the hash bucket for the given token `t` and then, it traverses sequentially the list of sibling nodes in the bucket checking for one representing `t`. Again, if such a node is found then execution is stopped and the node returned. Otherwise, a new trie node is initialized and inserted in the beginning of the bucket list. If reaching the threshold value `MAX_SIBLING_NODES_PER_BUCKET`, the current hash table is expanded.

To manipulate tries we use two interface procedures. For traversing a trie to check/insert for new calls or for new answers we use the

```
trie_check_insert(TRIE_NODE root, TERM term)
```

procedure, where `root` is the root node of the trie to be used and `term` is the call/answer term to be inserted. The `trie_check_insert()` procedure invokes repeatedly the previous `trie_node_check_insert()` procedure for each token that represents the given `term` and returns the reference to the leaf node representing its path. Note that inserting a term requires in the worst case allocating as many nodes as necessary to represent its complete path. On the other hand, inserting repeated terms requires traversing the trie structure until reaching the corresponding leaf node, without allocating any new node.

To load a term from a trie back to the Prolog engine we use the

```
trie_load(TRIE_NODE leaf)
```

procedure, where `leaf` is the reference to the leaf node of the term to be returned. When loading a term, the trie nodes are traversed in bottom-up order.

When inserting terms in the table space we need to distinguish two situations: **(i)** inserting tabled calls in a subgoal trie structure; and **(ii)** inserting

answers in a particular answer trie structure. The former situation is handled by the `subgoal_check_insert()` procedure as shown in Figure 5 and the latter situation is handled by the `answer_check_insert()` procedure as shown in Figure 6.

```
subgoal_check_insert(TABLE_ENTRY te, SUBGOAL_CALL call) {
  st_root_node = te->subgoal_trie_root_node
  if (GT_ROOT_NODE) {                                  // new table design
    leaf_gt_node = trie_check_insert(GT_ROOT_NODE, call)
    leaf_st_node = trie_node_check_insert(st_root_node, leaf_gt_node)
  } else {                                           // original table design
    leaf_st_node = trie_check_insert(st_root_node, call)
  }
  return leaf_st_node
}
```

Fig. 5. Pseudo-code for the `subgoal_check_insert()` procedure

In the original table design, the `subgoal_check_insert()` procedure simply uses the `trie_check_insert()` procedure to check/insert the given `call` in the subgoal trie corresponding to the given table entry `te`. In the new design based on the GT, the `subgoal_check_insert()` procedure now first checks/inserts the given `call` in the GT. Then, it uses the reference to the GT's leaf node representing `call` (`leaf_gt_node` in Figure 5) as the token to be checked/inserted in the subgoal trie corresponding to the given table entry `te`. Note that this is done by calling the `trie_node_check_insert()` procedure, thus if the list of sibling nodes in the subgoal trie exceeds the `MAX_SIBLING_NODES_PER_LEVEL` threshold value, then a new hash table is initialized as described before.

```
answer_check_insert(SUBGOAL_FRAME sf, ANSWER answer) {
  at_root_node = sf->answer_trie_root_node
  if (GT_ROOT_NODE) {                                  // new table design
    leaf_gt_node = trie_check_insert(GT_ROOT_NODE, answer)
    leaf_at_node = trie_node_check_insert(at_root_node, leaf_gt_node)
  } else {                                           // original table design
    leaf_at_node = trie_check_insert(at_root_node, answer)
  }
  return leaf_at_node
}
```

Fig. 6. Pseudo-code for the `answer_check_insert()` procedure

The `answer_check_insert()` procedure works similarly. In the original table design, it checks/inserts the given `answer` in the answer trie corresponding to the given subgoal frame `sf`. In the new design based on the GT, it first checks/inserts the given `answer` in the GT and, then, it uses the reference to the GT's leaf node representing `answer` (`leaf_at_node` in Figure 6) as the token to be checked/inserted in the answer trie corresponding to the given sub-

goal frame `sf`. Again, if the list of sibling nodes in the answer trie exceeds the `MAX_SIBLING_NODES_PER_LEVEL` threshold value, a new hash table is initialized.

Finally, the `answer_load()` procedure is used to consume answers. Figure 7 shows the pseudo-code for it. In the original table design, it simply uses the `trie_load()` procedure to load from the answer trie the answer given by the trie node `leaf_at_node`. In the new design based on the GT, the `answer_load()` procedure first accesses the GT's leaf node represented in the `token` field of the given trie node `leaf_at_node` (`leaf_gt_node` in Figure 7). Then, it uses the `trie_load()` procedure to load from the GT back to the Prolog engine the answer represented by the obtained GT's leaf node.

```
answer_load(ANSWER_TRIE_NODE leaf_at_node) {
  if (GT_ROOT_NODE) {                                  // new table design
    leaf_gt_node = leaf_at_node->token
    answer = trie_load(leaf_gt_node)
  } else {                                          // original table design
    answer = trie_load(leaf_at_node)
  }
  return answer
}
```

Fig. 7. Pseudo-code for the `answer_load()` procedure

5 Preliminary Experimental Results

We next present some preliminary experimental results comparing YapTab with and without support for the common global trie data structure. 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.

To evaluate the impact of our proposal, we have defined a tabled predicate `t/5` that simply stores in the table space terms defined by `term/1` facts, and then we used a top query goal `test/0` to recursively call `t/5` with all combinations of one and two free variables in the arguments. An example of such code for functor terms of arity 1 (500 terms in total) is shown next.

```
:- table t/5.
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(f(1),f(1),f(1),f(1),A), fail.        ...
test :- t(A,B,f(1),f(1),f(1)), fail.           term(f(499)).
...                                            term(f(500)).
test :- t(f(1),f(1),f(1),A,B), fail.
test.
```

We experimented the `test/0` predicate with 7 different kinds of 500 `term/1` facts: integers, atoms and functor terms of arity 1 to 5. Table 1 shows the memory

usage, in KBytes, and the running times, in milliseconds, to store to the tables (first execution) and to load from the tables (second execution) the complete set of subgoals/answers for YapTab with (column ***YapTab+GT***) and without (column ***YapTab***) support for the common global trie data structure.

Terms	*YapTab (a)*			*YapTab+GT (b)*			*Ratio (b)/(a)*		
	Mem	*Store*	*Load*	*Mem*	*Store*	*Load*	*Mem*	*Store*	*Load*
500 int	49074	490	155	52803	738	164	1.08	1.51	1.06
500 atom	49074	508	158	52803	770	167	1.08	1.52	1.06
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**

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

The results show that GT support can reduce memory usage proportionally to the depth and redundancy of the terms stored in the GT. In particular, for functor terms of arity 2 to 5, the results show an increasing and very significant reduction on memory usage. The results for integer and atoms terms are also very interesting as they show that the cost of representing only atomic terms in the GT (between 7% and 8% in these experiments) can be manageable when we increase redundancy. Note that integers and atoms terms are represented by a single node in the original YapTab design, and by an extra node (therefore requiring two nodes) if using the GT approach.

On the other hand, these results seem to indicate that memory reduction comes at a price in execution time. With GT support, we need to navigate in two tries when 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 decreases also 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.

6 Conclusions and Further Work

We have presented a new design for the table space organization that uses a common global trie to store terms in tabled subgoal calls and answers. Our goal is to reduce redundancy in term representation, thus saving memory by sharing data that is structurally equal. Our preliminary experiments showed very significant reductions on memory usage.

Further work will include exploring the impact of applying our proposal to real-world applications that pose many subgoal queries, possibly with a large number of redundant answers, such as ILP applications, seeking real-world experimental results allowing us to improve and expand our current implementation. In particular, we intend to study how alternative designs for the table space organization can further reduce redundancy in term representation.

Acknowledgements

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.

References

1. Michie, D.: Memo Functions and Machine Learning. Nature **218** (1968) 19–22
2. Tamaki, H., Sato, T.: OLDT Resolution with Tabulation. In: International Conference on Logic Programming. Number 225 in LNCS, Springer-Verlag (1986) 84–98
3. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43**(1) (1996) 20–74
4. 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
5. Rocha, R.: On Improving the Efficiency and Robustness of Table Storage Mechanisms for Tabled Evaluation. In: International Symposium on Practical Aspects of Declarative Languages. Number 4354 in LNCS, Springer-Verlag (2007) 155–169
6. Costa, P., Rocha, R., Ferreira, M.: Tabling Logic Programs in a Database. In: Workshop on (Constraint) Logic Programming. (2007) 125–135
7. 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
8. 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
9. Goto, E.: Monocopy and Associative Algorithms in Extended Lisp. Technical Report TR 74-03, University of Tokyo (1974)
10. 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
11. Rocha, R., Silva, F., Santos Costa, V.: On applying or-parallelism and tabling to logic programs. Theory and Practice of Logic Programming **5**(1 & 2) (2005) 161–205
12. Fredkin, E.: Trie Memory. Communications of the ACM **3** (1962) 490–499
13. 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