A Term-Based Global Trie for Tabled Logic Programs

Jorge Costa, João Raimundo, and Ricardo Rocha *

DCC-FC & CRACS
University of Porto, Portugal
{jcosta,jraimundo,ricroc}@dcc.fc.up.pt

Abstract. A critical component in the implementation of an efficient tabling system is the design of the data structures and algorithms to access and manipulate tabled data. Arguably, the most successful data structure for tabling is tries. However, when used in applications that pose many queries and/or have a large number of answers, tabling can build arbitrarily many and/or very large tables, quickly filling up memory. In this paper, we propose a new design for the table space organization where all terms in tabled subgoal calls and tabled answers are represented only once in a common global trie instead of being spread over several different trie data structures. Our initial experiments using the YapTab tabling system show significant reductions on memory usage without compromising running time.

Key words: Tabling Logic Programming, Table Space, Implementation.

1 Introduction

Tabling is an implementation technique that overcomes some limitations of traditional Prolog systems in dealing with redundant sub-computations and recursion. Tabling has become a popular and successful technique thanks to the groundbreaking work in the XSB Prolog system and in particular in the SLG-WAM engine [1]. The success of SLG-WAM led to several alternative implementations that differ in the execution rule, in the data-structures used to implement tabling, and in the changes to the underlying Prolog engine. Implementations of tabling are now widely available in systems like Yap Prolog, B-Prolog, ALS-Prolog, Mercury and more recently Ciao Prolog.

A critical component in the implementation of an efficient tabling system is the design of the data structures and algorithms to access and manipulate tabled data. Arguably, the most successful data structure for tabling is *tries* [2]. Tries are trees in which common prefixes are represented only once. The trie data structure provides complete discrimination for terms and permits look up and possibly insertion to be performed in a single pass through a term, hence resulting in a very efficient and compact data structure for term representation.

* This work has been partially supported by the FCT research projects STAMPA (PTDC/EIA/67738/2006) and JEDI (PTDC/EIA/66924/2006).

Despite the good properties of tries, when used in applications that pose many queries and/or have a large number of answers, tabling can build arbitrarily many and/or very large tables, quickly filling up memory [3]. A possible solution for this problem is to dynamically abolish some of the tables. This can be done by using explicit tabling primitives or by using a memory management strategy that automatically recovers space among the least recently used tables when memory runs out [4]. An alternative approach is to store tables externally in a relational database system and then reload them back only when necessary [5].

A complementary approach to the previous problem is to study how less redundant and more compact data structures can be used to better represent the table space. In this paper, we propose a new design for the table space organization where all terms in tabled subgoal calls and tabled answers are represented only once in a *common global trie* instead of being spread over several different trie data structures. Our approach resembles the *hash-consing* technique [6], as it shares data that is structurally equal, thus saving memory usage by reducing redundancy in term representation. We will focus our discussion on a concrete implementation, the YapTab system [7], 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 introduce 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 the new design. At last, we present some experimental results and we end by outlining some conclusions.

2 Tabling Tries

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*. Repeated calls to tabled subgoals[1] are not re-evaluated against the program clauses, instead they are resolved by consuming the answers already stored in their table entries. During this process, as further new answers are found, they are stored in their tables and later returned to all repeated calls.

Within this model, 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 repeated subgoals. With these requirements, a correct design of the table space is critical to achieve an efficient implementation. YapTab uses *tries* which is regarded as a very efficient way to implement the table space [2].

A trie is a tree structure where each different path through the trie data units, the *trie nodes*, corresponds to 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

[1] A subgoal repeats a previous subgoal if they are the same up to variable renaming.

$f(X, g(Y, X), Z)$ is the sequence of 6 tokens: $f/3$, VAR_0, $g/2$, VAR_1, VAR_0 and VAR_2, where each variable is represented as a distinct VAR_i constant [8].

To increase performance, YapTab implements tables using two levels of tries: one for subgoal calls; the other for computed answers. More specifically:

- 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. Contrary 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 [2]. Repeated calls to tabled subgoals load answers 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, the subgoal `t(f(1),Y)` is called and three trie nodes are inserted: one for functor `f/1`, a second for constant `1` and one last for variable `Y` (`VAR0`). The subgoal frame is inserted as a leaf, waiting for the answers. Next, the subgoal `t(X,Y)` is also called. The two calls differ in the first argument, so tries bring no benefit here. Two new trie nodes, for variables `X` (`VAR0`) and `Y` (`VAR1`), 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. Subgoal `t(f(1),Y)` has two answers, `Y=f(1)` and `Y=f(2)`, so we need three trie

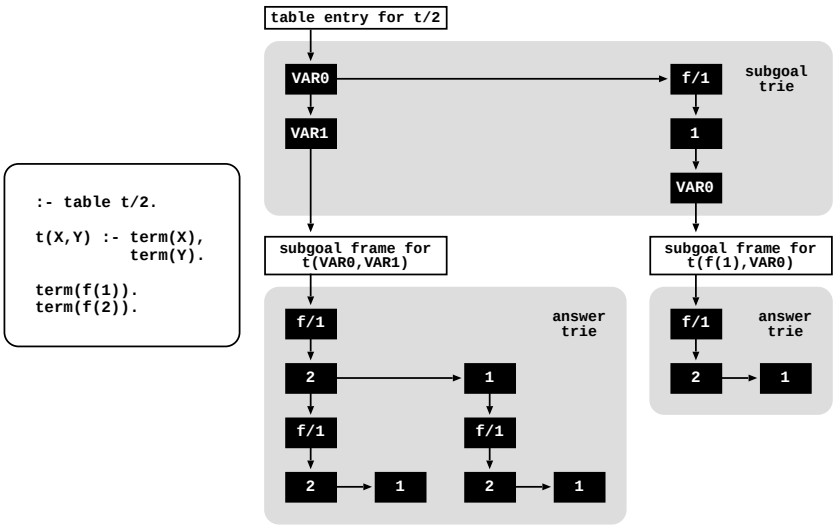

Fig. 1. Original table organization

nodes to represent both: a common node for functor `f/1` and two nodes for constants `1` and `2`. For subgoal `t(X,Y)` we have four answers, resulting from the combination of the answers `f(1)` and `f(2)` for variables `X` and `Y`, which requires nine trie nodes to represent them. Note that, for this particular example, the completed answer trie for `t(X,Y)` includes in its representation the completed answer trie for `t(f(1),Y)`.

3 Common Global Trie

In this section, we describe YapTab's new design for the table space organization. Our new design can be seen as an extension of a previous approach [9], where we first introduced the idea of using a common global trie. In what follows, we will refer to our previous approach as the *Global Trie for Calls and Answers* (GT-CA), and to our new design as the *Global Trie for Terms* (GT-T). Next, we start by briefly introducing the GT-CA approach and then we discuss in more detail how we have extended and optimized it to our new GT-T design.

3.1 Global Trie for Calls and Answers

In the GT-CA approach, all tabled subgoal calls and answers are stored in a common global trie instead of being spread over several different trie data structures. The GT-CA still is a tree structure where each different path through the trie nodes corresponds to a subgoal call and/or answer. However, here a path can end at any internal trie node and not necessarily at a leaf trie node.

The original subgoal trie and answer trie data structures are now represented by a unique level of nodes that point to the corresponding paths in the GT-CA (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-CA that represents the argument terms for the subgoal call. 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-CA that represents the substitution terms for the free variables which exist in the argument terms. With this organization, answers are now loaded by following the pointer in the node's token and then by traversing bottom-up the corresponding GT-CA's nodes.

Figure 2 uses again the example from Fig. 1 to illustrate how the GT-CA design works. Initially, the subgoal trie and the GT-CA are empty. Then, the first subgoal `t(f(1),Y)` is called and three nodes are inserted in the GT-CA: one to represent the functor `f/1`, another for the constant `1` and a last one for variable `Y` (`VAR0`). Next, a node representing the path inserted in the GT-CA is stored in the subgoal trie (node labeled `call1`). For the second subgoal call, `t(X,Y)`, we start again by inserting the call in the GT-CA and then we store a node in the subgoal trie (node labeled `call2`) to represent the path inserted in the GT-CA. Each answer is also inserted first in the GT-CA and then we store a node in the corresponding answer trie (nodes labeled `answer1`, `answer2`, `answer3` and `answer4`) to represent the path inserted in the GT-CA.

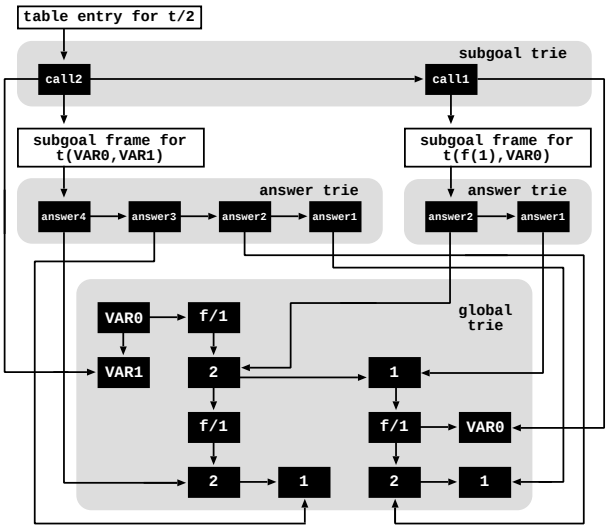

Fig. 2. GT-CA table organization

This example shows us that with the GT-CA we cannot share the representation of common terms appearing at different argument or substitution positions. For example, the terms `f(1)`, `f(2)` and `VAR0` appear more than once represented in the global trie. Moreover, with this example, we can see also that terms in the GT-CA can end at any internal trie node and not necessarily at a leaf trie node. This happens because tabled subgoals calls and answers are not always necessarily *pure* terms. A subgoal call is, in fact, represented by a sequence of argument terms and an answer is, in fact, represented by a sequence of substitution terms. Thus, when the number of argument or substitution terms is greater than one, then we may have situations where a subgoal call or answer can end at internal nodes of other subgoal calls and/or answers. This raises a problem when supporting table abolish operations because the nodes representing an individual subgoal call or answer may not be removed if they belong to other different paths. This problem can be solved by introducing an extra field in each trie node to count the number of paths it belongs to and only allow deletion when it reaches zero, but this solution is contradictory with our goal of saving memory.

Another problem with the GT-CA design is that, 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* [2]. 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 the GT-CA design, the nodes in the global trie can belong to several different subgoal/answer tries, and thus this optimization is no longer possible.

We next discuss how we have extended and optimized this table organization to the new GT-T design in order to solve these problems.

3.2 Global Trie for Terms

The GT-T was designed in order to maximize the sharing of tabled data that is structurally equal. In the GT-T design, all argument and substitution terms appearing in tabled subgoal calls and/or answers are represented *only once* in the common global trie. The GT-T still is a tree structure where each different path through the trie nodes represents a *unique* argument and/or substitution term, therefore always ending at a leaf trie node. Each path in a subgoal or answer trie is now composed of a fixed number of trie nodes representing the argument or substitution terms in the corresponding tabled subgoal call or answer. For the subgoal tries, each node now represents an argument term where the node's token is the pointer to the unique path in the GT-T representing the term. For the answer tries, each node now represents a substitution term where the node's token is the pointer to the unique path in the GT-T representing the term.

Figure 3 uses again the example from Fig. 1 to illustrate how the GT-T design works. Initially, the subgoal trie and the GT-T are empty. Next, the first subgoal `t(f(1),Y)` is called and the two argument terms, `f(1)` and `Y` (`VAR0`), are first inserted in the GT-T. Then, the argument terms are represented in the subgoal trie by two nodes (nodes labeled `arg1` and `arg2`), each one pointing to the leaf node of the corresponding term inserted in the GT-T. For the second subgoal call, `t(X,Y)`, the argument terms `VAR0` and `VAR1` are also inserted first in the GT-T and then we store also two nodes in the subgoal trie, each one pointing to the corresponding representation in the GT-T.

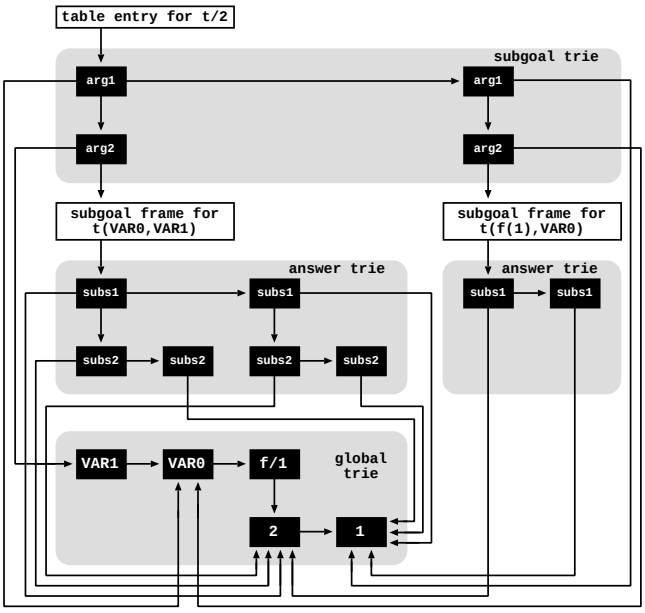

Fig. 3. GT-T table organization

For the answers, each substitution term is also inserted first in the GT-T and then we store a node in the corresponding answer trie to represent its path in the GT-T (nodes labeled `subs1` and `subs2`). The substitution terms for the complete set of answers for the two subgoal calls only include the terms `f(1)` and `f(2)`. Thus, as `f(1)` is already stored in the global trie, we only need to insert `f(2)` in order to be able to represent the full set of answers. As we are maximizing the sharing of common terms appearing at different argument or substitution positions, for this particular example, this results in a very compact representation of the global trie.

Regarding space reclamation, as each different path in the GT-T always ends at a leaf node, we can use the child field (that is always NULL in a leaf node) to count the number of references to the path it represents and only allow deletion when it reaches zero. This solves the previous problem of supporting table abolish operations without introducing extra memory overheads.

Regarding compiled tries, the idea is to keep the global trie only with the term representation and store the WAM-like instructions in the answer tries, as in the original design [2]. The difference is that for the GT-T approach, the WAM-like instructions are more *high-level*, i.e., instead of working at the level of atoms/terms/functors/lists as in [2], each instruction works at the level of the substitution terms. For example, consider again the four answers for the call `t(X,Y)`. When loading these answers, we have two choices for `X` and, for each `X`, we have two choices for `Y`. In the GT-T design, the answer trie nodes representing the choices for `X` and for `Y` (nodes labeled respectively `subs1` and `subs2`) are compiled with a WAM-like sequence such as `try_subs_term` (for the first choices) and `trust_subs_term` (for the second/last choices). GT-T's compiled tries also include a `retry_subs_term` instruction (for intermediate choices) and a `do_subs_term` instruction (for single choices).

4 Implementation Details

We then describe in more detail the data structures and algorithms for YapTab's new table design. We start with Fig. 4 showing in more detail the table organization previously presented in Fig. 3 for the subgoal call `t(X,Y)`.

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 trie nodes, 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 next sibling node. Remember that for the global trie, the leaf node's `child` field is used to count the number of references to the path it represents. For the answer tries, an

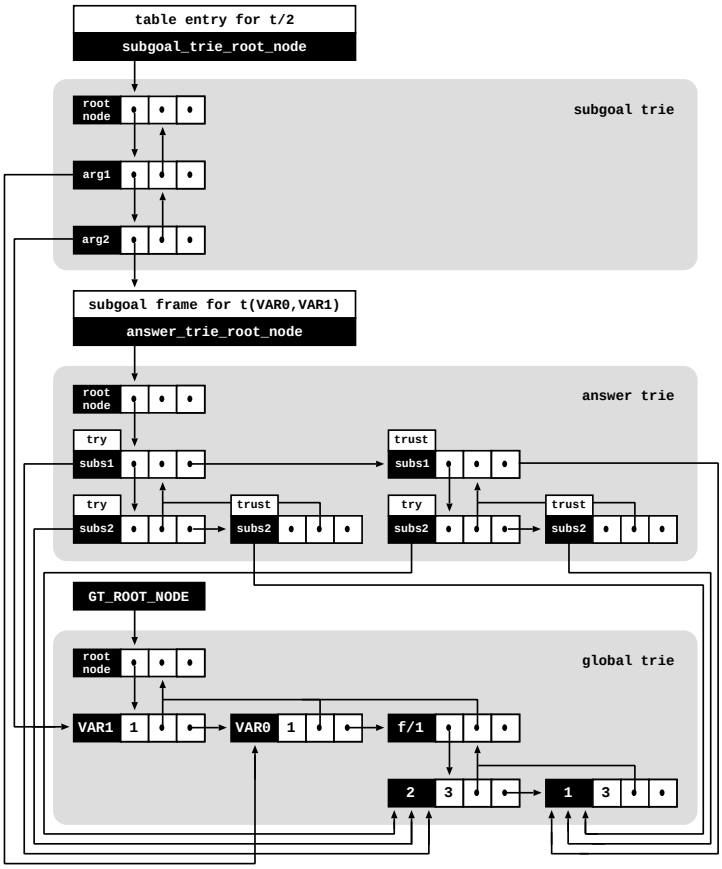

Fig. 4. Implementation details for the GT-T table organization

additional field (`code`) is used to support compiled tries. 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 5 shows the pseudo-code for this 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. For answer trie nodes, the `code` field is computed later when completion takes place.

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

```
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
  } else if (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. 5. Pseudo-code for the `trie_node_check_insert()` procedure

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 execu-

tion 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:

```
trie_load(TRIE_NODE leaf)
trie_check_insert(TRIE_NODE root, TERM t)
```

The `trie_load()` is used to load a term from a trie back to the Prolog engine, where `leaf` is the reference to the leaf node of the term to be loaded.

The `trie_check_insert()` is used for traversing a trie to check/insert for new terms, where `root` is the root node of the trie to be used and `t` is the term to be inserted. It invokes repeatedly the previous `trie_node_check_insert()` procedure for each token that represents the given term `t` 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 path. On the other hand, inserting repeated terms requires traversing the trie structure until reaching the corresponding leaf node, without allocating any new node.

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 Fig. 6 and the latter situation is handled by the `answer_check_insert()` procedure as shown in Fig. 7.

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 GT-T design, for each argument term `t`, it first checks/inserts the term `t` in the GT-T and, then, it uses the reference to the leaf node representing `t` in the GT-T (`leaf_gt_node` in Fig. 6) 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_check_insert(TABLE_ENTRY te, SUBGOAL_CALL call, ARGS_ARITY a) {
  if (GT_ROOT_NODE) {                                  // GT-T table design
    st_node = te->subgoal_trie_root_node
    for (i = 1; i <= a; i++) {
      t = get_argument_term(call, i)
      leaf_gt_node = trie_check_insert(GT_ROOT_NODE, t)
      leaf_gt_node->child++     // increase number of paths it represents
      st_node = trie_node_check_insert(st_node, leaf_gt_node)
    }
    leaf_st_node = st_node
  } else                                             // original table design
    leaf_st_node = trie_check_insert(te->subgoal_trie_root_node, call)
  return leaf_st_node
}
```

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

```
answer_check_insert(SUBGOAL_FRAME sf, ANSWER answer, SUBS_ARITY a) {
  if (GT_ROOT_NODE) {                                  // GT-T table design
    at_node = sf->answer_trie_root_node
    for (i = 1; i <= a; i++) {
      t = get_substitution_term(answer, i)
      leaf_gt_node = trie_check_insert(GT_ROOT_NODE, t)
      leaf_gt_node->child++     // increase number of paths it represents
      at_node = trie_node_check_insert(at_node, leaf_gt_node)
    }
    leaf_at_node = at_node
  } else                                            // original table design
    leaf_at_node = trie_check_insert(sf->answer_trie_root_node, answer)
  return leaf_at_node
}
```

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

subgoal trie exceeds the `MAX_SIBLING_NODES_PER_LEVEL` threshold value, then a new hash table is initialized as described before.

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 GT-T design, for each substitution term `t`, it first checks/inserts the term `t` in the GT-T and, then, it uses the reference to the leaf node representing `t` in the GT-T (`leaf_gt_node` in Fig. 7) as the token to be checked/inserted in the answer trie corresponding to the given subgoal 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 8 shows the pseudo-code for it. In the original table design, it simply uses the `trie_load()` procedure to load from the answer trie back to the Prolog engine the answer given by the trie node `leaf_at_node`. In the new GT-T design, for each answer trie node `at_node`, now it uses the `trie_load()` procedure to load from the GT-T back to the Prolog engine the substitution term given by the reference (`leaf_gt_node` in Fig. 8) stored in the corresponding `token` field.

5 Experimental Results

We next present some experimental results comparing YapTab with and without support for the common global trie data structure. The environment for our experiments was an Intel(R) Core(TM)2 Quad 2.66GHz with 2 GBytes of main memory and running the Linux kernel 2.6.24.23 with YapTab 5.1.4.

To put the performance results in perspective and have a well-defined starting point comparing the GT-CA and GT-T approaches, first 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. We experimented the `test/0` predicate with 10 different kinds of 1000 `term/1` facts: integers,

```
answer_load(ANSWER_TRIE_NODE leaf_at_node, SUBS_ARITY a) {
  if (GT_ROOT_NODE) {                                  // GT-T table design
    at_node = leaf_at_node
    for (i = a; i >= 1; i--) {
      leaf_gt_node = at_node->token
      t = trie_load(leaf_gt_node)
      put_substitution_term(t, answer)
      at_node = at_node->parent
    }
  } else                                             // original table design
    answer = trie_load(leaf_at_node)
  return answer
}
```

Fig. 8. Pseudo-code for the `answer_load()` procedure

atoms, compound (with arities 1, 2, 4 and 6) and list (with lengths 1, 2, 4 and 6) terms. An example of such code for compound terms of arity 1 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.
...
test :- t(f(1),f(1),f(1),f(1),A), fail.
test :- t(A,B,f(1),f(1),f(1)), fail.
...
test :- t(f(1),f(1),f(1),A,B), fail.
test.
```

```
term(f(1)).
term(f(2)).
term(f(3)).
...
term(f(998)).
term(f(999)).
term(f(1000)).
```

Table 1 shows the table memory usage (columns ***Mem***), in MBytes, and the running times, in milliseconds, to store (columns ***Str***) the tables (first execution) and to load from the tables (second execution) the complete set of subgoals/answers without (columns ***Load***) and with (columns ***Cmp***) compiled tries for YapTab using the original table organization (column ***YapTab***), using the previous GT-CA approach (column ***GT-CA/YapTab***) and using the new GT-T design (column ***GT-T/YapTab***). For the GT-CA and GT-T approaches we only show the memory and running time ratios over YapTab's original table organization.

The results in Table 1 suggest that GT-T support is the best approach to reduce memory usage and that this reduction increases proportionally to the length and redundancy of the terms stored in the global trie. In particular, for compound and list terms, the results show an increasing and very significant reduction on memory usage, for both GT-CA and GT-T approaches. The results for integer and atoms terms are also very interesting as they show that the cost of representing only atomic terms in the global trie (around 8% for GT-CA and 0% for GT-T 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 a global trie.

Terms	*YapTab*				*GT-CA/YapTab*				*GT-T/YapTab*			
	Mem	*Str*	*Load*	*Cmp*	*Mem*	*Str*	*Load*	*Cmp*	*Mem*	*Str*	*Load*	*Cmp*
1000 ints	191	1009	358	207	1.08	1.56	1.30	n.a.	1.00	1.32	1.18	1.69
1000 atoms	191	1040	337	231	1.08	1.54	1.41	n.a.	1.00	1.26	1.24	1.54
1000 f/1	191	1474	548	239	1.08	1.35	1.33	n.a.	1.00	1.28	1.11	1.88
1000 f/2	382	1840	632	353	**0.58**	1.25	1.37	n.a.	**0.50**	1.11	1.18	1.58
1000 f/4	764	2581	786	631	**0.33**	1.21	1.35	n.a.	**0.25**	1.07	1.16	1.14
1000 f/6	1146	3379	1032	765	**0.25**	1.12	1.29	n.a.	**0.17**	1.01	1.05	1.08
1000 [ ]/1	382	1727	466	365	**0.58**	1.32	1.44	n.a.	**0.50**	1.17	1.21	1.29
1000 [ ]/2	764	2663	648	459	**0.33**	1.06	1.55	n.a.	**0.25**	**0.93**	1.20	1.48
1000 [ ]/4	1528	4461	1064	720	**0.20**	1.10	1.57	n.a.	**0.13**	**0.81**	1.01	1.28
1000 [ ]/6	2293	6439	2386	1636	**0.16**	1.02	1.05	n.a.	**0.08**	**0.71**	**0.58**	**0.68**
Average					**0.57**	**1.25**	**1.37**	**n.a.**	**0.49**	**1.07**	**1.09**	**1.36**

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

Regarding running time, these results seem to indicate that memory reduction comes at a price in storing time (around 25% for GT-CA and 7% for GT-T in these experiments). Note that with GT-CA and GT-T support, we pay the cost of navigating in two tries when checking/storing/loading a term. Moreover, in some situations, the cost of storing a new term in an empty/small trie can be less than the cost of navigating in the global trie, even when the term is already stored in the global trie. However, our results seem to suggest that this cost decreases proportionally to the length and redundancy of the terms stored in the global trie. In particular, for list terms, GT-T support showed to outperform the original YapTab design and, in particular, the reduction seems to decrease also proportionally to the length of the list terms stored in the global trie.

The results obtained for loading terms also show a cost on running time (around 37% for GT-CA and 9% and 36% for GT-T without and with compiled tries in these experiments). We think that this cost is smaller for GT-T as a result of a cache behaviour effect. With GT-T, as we need to navigate in the global trie for each substitution term, we kept accessing the same global trie nodes, thus reducing eventual cache misses. This seems also to be the reason why for list terms of length 6, GT-T clearly outperforms the original YapTab design, both without and with compiled tries. Note that, for this particular case, the GT-T support only consumes 8% of the memory used in the original YapTab.

Next, we tested our approach with two well-known Inductive Logic Programming (ILP) benchmarks: the *carcinogenesis* (***Carc***) and the *mutagenesis* (***Muta***) data sets. We used these data sets in a Prolog program that simulates the test phase of an ILP system. For that, first we ran the April ILP system [10] for the two data sets, each with two different configurations, in order to collect the set of clauses generated for each configuration. The simulator program then uses the corresponding set of generated clauses to run the positive and negative examples defined for each data set against them. To evaluate clauses, we used two different strategies: ***Pred*** denotes the tabling of individual predicates and

Data Sets	*YapTab*				*GT-CA/YapTab*				*GT-T/YapTab*			
	Mem	*Str*	*Load*	*Cmp*	*Mem*	*Str*	*Load*	*Cmp*	*Mem*	*Str*	*Load*	*Cmp*
Pred												
Carc_P1	1.6	70.72	71.26	72.95	**0.82**	1.35	1.34	n.a.	**0.62**	1.07	1.05	1.03
Carc_P2	2.1	51.19	50.44	55.97	**0.87**	1.42	1.44	n.a.	**0.51**	1.23	1.30	1.22
Muta_P1	0.6	98.93	5.57	5.86	**0.73**	1.20	1.19	n.a.	**0.63**	**0.91**	1.00	**0.94**
Muta_P2	0.6	93.01	2.01	2.40	**0.73**	1.26	1.47	n.a.	**0.63**	**0.96**	1.22	1.10
Average					**0.79**	**1.31**	**1.36**	**n.a.**	**0.60**	**1.04**	**1.14**	**1.07**
Conj												
Carc_C1	18.5	0.56	0.51	0.48	**0.53**	1.57	1.63	n.a.	**0.39**	1.20	1.22	1.08
Carc_C2	2802.8	93.85	70.16	36.44	**0.50**	1.50	1.50	n.a.	**0.14**	1.11	1.09	**0.82**
Muta_C1	84.7	97.02	7.36	6.14	**0.66**	1.30	1.65	n.a.	**0.53**	**0.99**	1.22	1.35
Muta_C2	675.6	92.76	1.36	1.53	**0.16**	1.25	1.42	n.a.	**0.16**	**0.98**	1.10	**0.78**
Average					**0.46**	**1.41**	**1.55**	**n.a.**	**0.31**	**1.07**	**1.16**	**1.01**

Table 2. Table memory usage (in MBytes) and store/load times (in seconds) for YapTab with and without support for the common global trie data structure

Conj denotes the tabling of literal conjunctions (as described in [3]). By tabling conjunctions, we only need to compute them once. The strategy is then recursively applied as the ILP system generates more specific clauses, but this can increase the table memory usage arbitrarily.

Table 2 shows the table memory usage (columns ***Mem***), in MBytes, and the running times, in seconds, to store (columns ***Str***) the tables (first execution) and to load from the tables (second execution) the complete set of subgoals/answers without (columns ***Load***) and with (columns ***Cmp***) compiled tries for YapTab using the original table organization (column ***YapTab***), using the previous GT-CA approach (column ***GT-CA/YapTab***) and using the new GT-T design (column ***GT-T/YapTab***). Again, for the GT-CA and GT-T approaches we only show the memory and running time ratios over YapTab's original table organization.

In general, the results in Table 2 confirm the results obtained in Table 1 for memory usage. GT-T support clearly outperforms the original and GT-CA designs for memory usage. In particular, for the ***Conj*** strategy, memory usage showed to be significantly less with GT-T support. This happens because after a certain time, the ***Conj*** strategy will not table new terms, but only answers that are combinations of previous terms, therefore making the GT-T approach more feasible as it can share the representation of common terms appearing at different argument or substitution positions.

Regarding running time, the results in Table 2 also confirm and reinforce the results obtained in Table 1. GT-T support clearly outperforms the GT-CA design for storing and loading times and, for some configurations, it also outperforms the original YapTab design. This is the case for configurations either without or with compiled tries. These results suggest that, at least for some class of applications, GT-T support has potential to achieve significant reductions in memory usage without compromising running time.

6 Conclusions and Further Work

We have presented a new design for the table space organization where all argument and substitution terms appearing in tabled subgoal calls and/or answers are represented only once in a common global trie instead of being spread over several different trie data structures. The goal is to reduce redundancy in term representation by maximizing the sharing of tabled data that is structurally equal. Our experiments using the YapTab tabling system showed that our approach has potential to achieve significant reductions on memory usage without compromising running time.

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

References

1. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems **20**(3) (1998) 586–634
2. 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
3. Rocha, R., Fonseca, N.A., Santos Costa, V.: On Applying Tabling to Inductive Logic Programming. In: European Conference on Machine Learning. Number 3720 in LNAI, Springer-Verlag (2005) 707–714
4. 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
5. Costa, P., Rocha, R., Ferreira, M.: Relational Models for Tabling Logic Programs in a Database. In: Workshop on (Constraint) Logic Programming. Number 5437 in LNAI, Springer-Verlag (2009) 99–116
6. Goto, E.: Monocopy and Associative Algorithms in Extended Lisp. Technical Report TR 74-03, University of Tokyo (1974)
7. 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
8. 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
9. Costa, J., Rocha, R.: One Table Fits All. In: International Symposium on Practical Aspects of Declarative Languages. Number 5418 in LNCS, Springer-Verlag (2009) 195–208
10. Fonseca, N.A., Silva, F., Camacho, R.: April - An Inductive Logic Programming System. In: European Conference on Logics in Artificial Intelligence. Number 4160 in LNAI, Springer-Verlag (2006) 481–484