A Very Compact and Efficient Representation of List Terms for Tabled Logic Programs

João Raimundo and Ricardo Rocha

DCC-FC & CRACS, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{jraimundo,ricroc}@dcc.fc.up.pt

Abstract. Tabling is an implementation technique that overcomes some limitations of traditional Prolog systems in dealing with redundant sub-computations and recursion. 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, which is regarded as a very compact and efficient data structure for term representation. Despite these good properties, we found that, for list terms, we can design even more compact and efficient representations. We thus propose a new representation of list terms for tries that avoids the recursive nature of the WAM representation of list terms in which tries are based. Our experimental results using the YapTab tabling system show a significant reduction in the memory usage for the trie data structures and impressive gains in the running time for storing and loading list terms.

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

1 Introduction

Tabling [1] 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 ground-breaking work in the XSB Prolog system and in particular in the SLG-WAM engine [2]. 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* [3]. Tries are trees in which common prefixes are represented only once. The trie data structure provides complete discrimination for terms and permits lookup and possibly insertion to be performed in a single pass through a term, hence resulting in a very compact and efficient data structure for term representation.

When representing terms in the trie, most tabling engines, like XSB Prolog, Yap Prolog and others, try to mimic the WAM [4] representation of these terms in the Prolog stacks in order to avoid unnecessary transformations when storing/loading these terms to/from the trie. Despite this idea seems straightforward for almost all type of terms, we found that this is not the case for *list terms* (also known as *pair terms*) and that, for list terms, we can design even more compact and efficient representations.

In Prolog, a non-empty list term is formed by two sub-terms, the *head of the list*, which can be any Prolog term, and the *tail of the list*, which can be either a non-empty list (formed itself by a head and a tail) or the *empty list*. WAM based implementations explore this recursive nature of list terms to design a very simple representation at the engine level that allows for very robust implementations of key features of the WAM, like the unification algorithm, when manipulating list terms. However, when representing terms in the trie, the recursive nature of the WAM representation of list terms is negligible as we are most interested in having a compact representation with fast lookup and insertion capabilities.

In this paper, we thus propose a new representation of list terms for tabled data that gets around the recursive nature of the WAM representation of list terms. In our new proposal, a list term is simply represented as the ordered sequence of the term elements in the list, i.e., we only represent the head terms in the sub-lists and avoid representing the sub-lists' tails themselves. Our experimental results show a significant reduction in the memory usage for the trie data structures and impressive gains in the running time for storing and loading list terms with and without compiled tries. 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.

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 list terms representation. Then, we discuss the implications of the new design and describe how we have extended YapTab to provide engine support for it. 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

---

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

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 [3].

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. For example, the tokenized form of the term $f(X, g(Y, X), Z)$ is the sequence of 6 tokens $< f/3, VAR_0, g/2, VAR_1, VAR_0, VAR_2 >$ where each variable is represented as a distinct VAR_i constant [6]. An essential property of the trie structure is that common prefixes are represented only once. Two terms with common prefixes will branch off from each other at the first distinguishing token. Figure 1 shows an example for a trie with three terms. Initially, the trie contains the root node only. Next, we store the term $f(X, a)$ and three trie nodes are inserted: one for the functor $f/2$, a second for variable X (VAR_0) and one last for constant a. The second step is to store $g(X, Y)$. The two terms differ on the main functor, so tries bring no benefit here. In the last step, we store $f(Y, 1)$ and we save the two common nodes with $f(X, a)$.
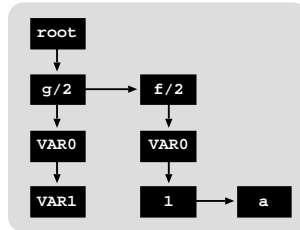

Fig. 1. Representing terms $f(X, a)$, $g(X, Y)$ and $f(Y, 1)$ in a trie

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. This optimization is called *substitution factoring* [3].

An example for a tabled predicate $t/2$ is shown in Fig. 2. Initially, the subgoal trie is empty[2]. Then, the subgoal $t(X, f(1))$ is called and three trie nodes are inserted: one for variable X (VAR_0), a second for functor $f/1$ and one last for constant 1[3]. 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 on the second argument, so we need an extra node to represent variable Y (VAR_1) followed by a new subgoal frame. At the end, the answers for each subgoal are stored in the corresponding answer trie as their values are computed. Subgoal $t(X, f(1))$ has two answers, $X = f(1)$ and $X = f(Z)$, so we need three trie nodes to represent both: a common node for functor $f/1$ and two nodes for constant 1 and variable Z (VAR_0)[4]. For subgoal $t(X, Y)$ we have four answers, resulting from the combination of the answers $f(1)$ and $f(Z)$ for variables X and Y, which requires nine trie nodes.
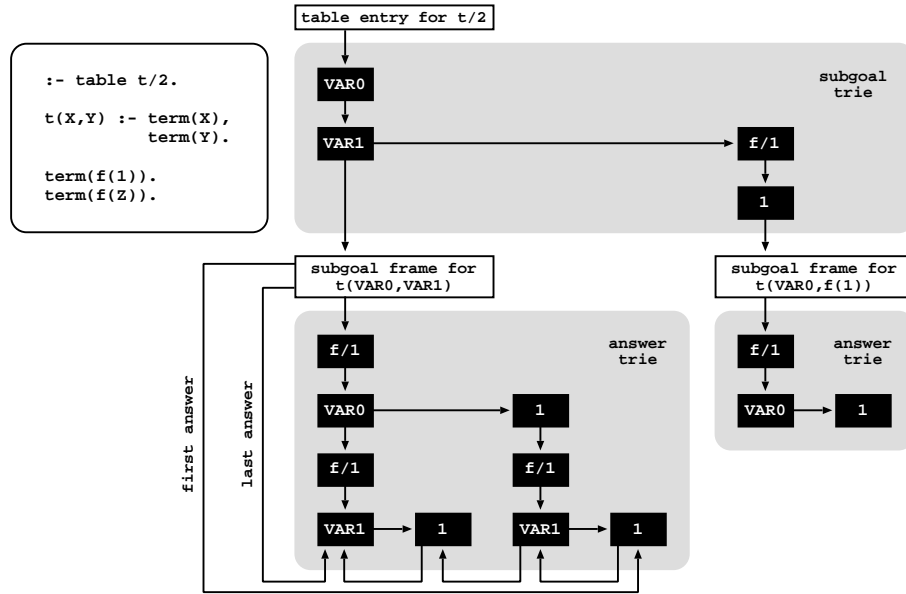

Fig. 2. YapTab table organization

Leaf answer trie nodes are chained in a linked list in insertion time order, so that we can recover answers in the same order they were inserted. The subgoal frame points to the first and last answer in this list. Thus, a repeated call

---

[2] In order to simplify the presentation of the following illustrations, we will omit the representation of the trie root nodes.

[3] Note that for subgoal tries, we can avoid inserting the predicate name, as it is already represented in the table entry.

[4] The way variables are numbered in a trie is specific to each trie and thus there is no correspondence between variables sharing the same number in different tries.

only needs to point at the leaf node for its last loaded answer, and consumes more answers by just following the chain. To load an answer, the trie nodes are traversed in bottom-up order and the answer is reconstructed.

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*. This optimization implements answer recovery by top-down traversing the completed answer tries and by executing dynamically compiled WAM-like instructions from the answer trie nodes. These dynamically compiled instructions are called *trie instructions* and the answer tries that consist of these instructions are called *compiled tries* [3]. Compiled tries are based on the observation that all common prefixes of the terms in a trie are shared during execution of the trie instructions. Thus, when backtracking through the terms of a trie that is represented using the trie instructions, each edge of the trie is traversed only once. Figure 3 shows the compiled trie for subgoal call $t(VAR_0, VAR_1)$ in Fig. 2.
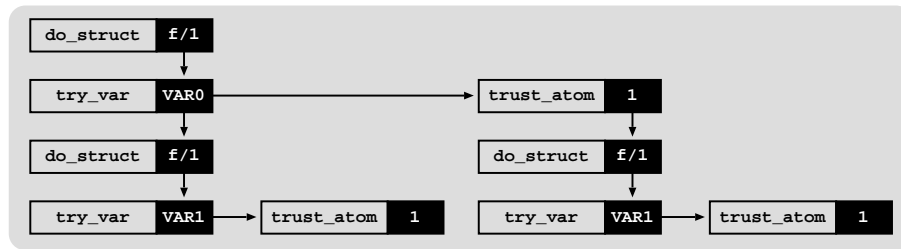

Fig. 3. Compiled trie for subgoal call $t(VAR_0, VAR_1)$ in Fig. 2

Each trie node is compiled accordingly to its position in the list of sibling nodes and to the term type it represents. For each term type there are four specialized trie instructions. First nodes in a list of sibling nodes are compiled using *try_?* instructions, intermediate nodes are compiled using *retry_?* instructions, and last nodes are compiled using *trust_?* instructions. Trie nodes without sibling nodes are compiled using *do_?* instructions. For example, for atom terms, the trie instructions are: *try_atom*, *retry_atom*, *trust_atom* and *do_atom*. As the *try_?/retry_?/trust_?* instructions denote the choice possibilities when traversing top-down an answer trie, at the engine level, they allocate and manipulate a choice point in a manner similar to the generic *try/retry/trust* WAM instructions, but here the failure continuation points to the next sibling node. The *do_?* instructions denote no choice and thus they don't allocate choice points.

The implementation of tries requires the following fields per trie node: a first field (`token`) stores the token for the node, a 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. For the answer tries, an additional fifth field (`code`) is used to support compiled tries.

3 Representation of List Terms

In this section, we introduce YapTab's new design for the representation of list terms. In what follows, we will refer to the original design as *standard lists* and to our new design as *compact lists*. Next, we start by briefly introducing how standard lists are represented in YapTab and then we discuss in more detail the new design for representing compact lists.

3.1 Standard Lists

YapTab follows the seminal WAM representation of list terms [4]. In YapTab, list terms are recursive data structures implemented using *pairs*, where the first pair element, the *head of the list*, represents a list element and the second pair element, the *tail of the list*, represents the list continuation term or the end of the list. In YapTab, the end of the list is represented by the empty list atom [ ]. At the engine level, a pair is implemented as a pointer to two contiguous cells, the first cell representing the head of the list and the second the tail of the list. In YapTab, as we will see next, the tail of a list can be any term. Figure 4(a) shows YapTab's WAM representation for lists in more detail.

Alternatively to the standard notation for list terms, we can use the pair notation $[H|T]$, where H denotes the head of the list and T denotes its tail. For example, the list term $[1, 2, 3]$ in Fig. 4 can be alternatively denoted as $[1|[2, 3]]$, $[1|[2|[3]]]$ or $[1|[2|[3|[\ ]]]]$. The pair notation is also useful when the tail of a list is neither a continuation list nor the empty list. See, for example, the list term $[1, 2|3]$ in Fig. 4(a) and its corresponding WAM representation. In what follows, we will refer to these lists as *term-ending lists* and to the lists ending with the empty list atom as *empty-ending lists*.
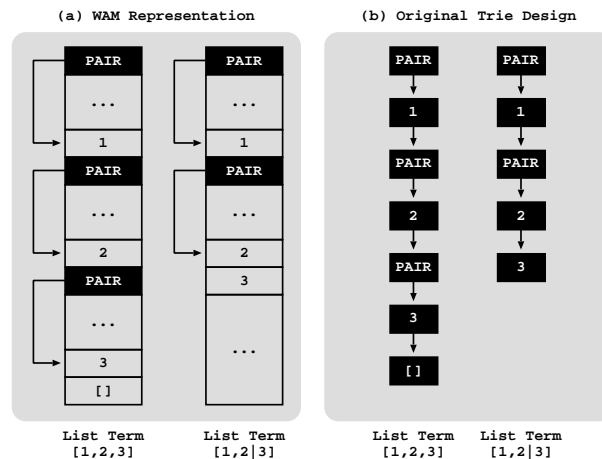

Fig. 4. YapTab's WAM representation and original trie design for standard lists

Regarding the trie representation of lists, the original YapTab design, as most tabling engines, including XSB Prolog, tries to mimic the corresponding WAM representation. This is done by making a direct correspondence between each pair pointer at the engine level and a trie node labelled with the special token PAIR. For example, the tokenized form of the list term $[1, 2, 3]$ is the sequence of 7 tokens $<$ PAIR, 1, PAIR, 2, PAIR, 3, [ ] $>$. Figure 4(b) shows in more detail YapTab's original trie design for the list terms represented in Fig. 4(a).

3.2 Compact Lists

In this section, we introduce the new design for the representation of list terms. The discussion we present next tries to follow the different approaches that we have considered until reaching our current final design. The key idea common to all these approaches is to avoid the recursive nature of the WAM representation of list terms and have a more compact representation where the unnecessary intermediate PAIR tokens are removed.

Figure 5 shows our initial approach. In this first approach, all intermediate PAIR tokens are removed and a compact list is simply represented by its term elements surrounded by a begin and a end list mark, respectively, the BLIST and ELIST tokens. Figure 5(a) shows the tokenized form of the empty-ending list $[1, 2, 3]$ that now is the sequence of 6 tokens $<$ BLIST, 1, 2, 3, [ ], ELIST $>$ and the tokenized form of the term-ending list $[1, 2|3]$ that now is the sequence of 5 tokens $<$ BLIST, 1, 2, 3, ELIST $>$.

Our approach clearly outperforms the standard lists representation when representing individual lists (except for the base cases of list terms of sizes 1 to 3). It requires about half the nodes when representing individual lists. For an empty-ending list of S elements, standard lists require $2S + 1$ trie nodes and
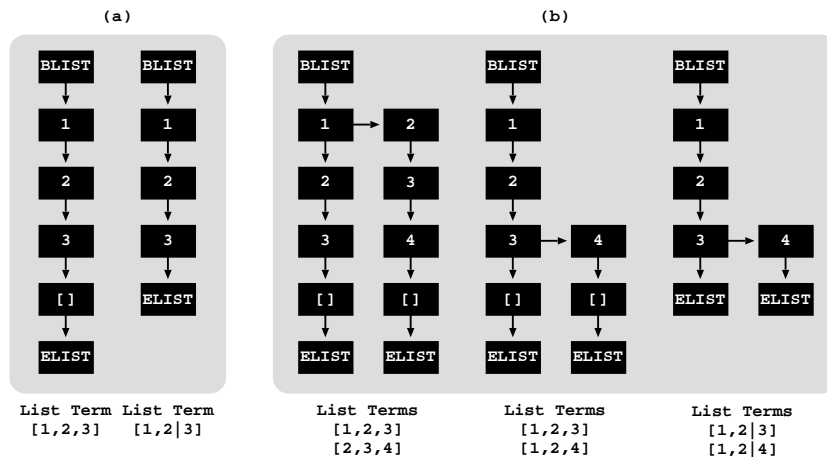

Fig. 5. Trie design for compact lists: initial approach

compact lists require $S+3$ nodes. For a term-ending list of S elements, standard lists require $2S - 1$ trie nodes and compact lists require $S + 2$ nodes.

Next, in Fig. 5(b) we try to illustrate how this approach behaves when we represent more than a list in the same trie. It presents three different situations: the first situation shows two lists with the first element different (a kind of worst case scenario); the second and third situations show, respectively, two empty-ending and two term-ending lists with the last element different (a kind of best case scenario).

Now consider that we generalize these situations and represent in the same trie N lists of S elements each. Our approach is always better for the first situation, but this may not be the case for the second and third situations. For the second situation (empty-ending lists with last element different), standard lists require $2N + 2S - 1$ trie nodes and compact lists require $3N + S$ nodes and thus, if $N > S - 1$ then standard lists is better. For the third situation (term-ending lists with last element different), standard lists require $N + 2S - 2$ trie nodes and compact lists require $2N + S$ nodes and again, if $N > S - 2$ then standard lists is better.

The main problem with this approach is that it introduces an extra token in the end of each list, the ELIST token, that do not exists in the representation of standard lists. To avoid this problem, we have redesigned our compact lists representation in such a way that the ELIST token appears only once for lists with the last element different. Figure 6 shows our second approach for the representation of compact lists.

In this second approach, a compact list still contains the begin and end list tokens, BLIST and ELIST, but now the ELIST token plays the same role of the last PAIR token in standard lists, i.e., it marks the last pair term in the list. Figure 6(a) shows the new tokenized form of the empty-ending list $[1, 2, 3]$
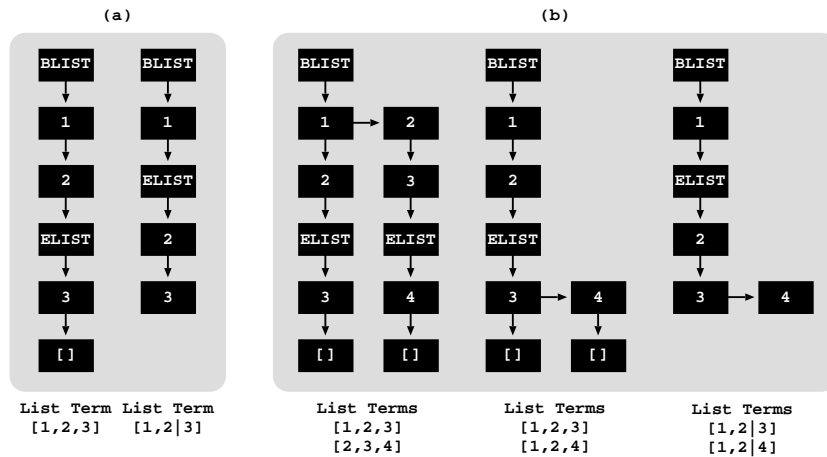

Fig. 6. Trie design for compact lists: second approach

that now is < BLIST, 1, 2, ELIST, 3, [ ] >, and the new tokenized form of the term-ending list [1, 2|3] that now is < BLIST, 1, ELIST, 2, 3 >.

Figure 6(b) illustrates again the same three situations showing how this second approach behaves when we represent more than a list in the same trie. For the first situation, the second approach is identical to the initial approach. For the second and third situations, the second approach is not only better than the initial approach, but also better than the standard lists representation (except for the base cases of list terms of sizes 1 and 2).

Consider again the generalization to represent in the same trie N lists of S elements each. For the second situation (empty-ending lists with last element different), compact lists now require $2N + S + 1$ trie nodes (the initial approach for compact lists require $3N + S$ nodes and standard lists require $2N + 2S - 1$ nodes). For the third situation (term-ending lists with last element different), compact lists now require $N + S + 1$ trie nodes (the initial approach for compact lists require $2N + S$ nodes and standard lists require $N + 2S - 2$ nodes). Despite these better results, this second approach still contains some drawbacks that can be improved. Figure 7 shows our final approach for the representation of compact lists.

In this final approach, we have redesigned our previous approach in such a way that the empty list token [ ] was avoided in the representation of empty-ending lists. Note that, in our previous approaches, the empty list token is what allows us to distinguish between empty-ending lists and term-ending lists. As we need to maintain this distinction, we cannot simply remove the empty list token from the representation of compact lists. To solve that, we use a different end list token, EPAIR, for term-ending lists. Hence, the ELIST token marks the last element in an empty-ending list and the EPAIR token marks the last element in an term-ending list. Figure 7(a) shows the new tokenized form of the empty-ending list [1, 2, 3] that now is < BLIST, 1, 2, ELIST, 3 >, and the new tokenized form of the term-ending list [1, 2|3] that now is < BLIST, 1, 2, EPAIR, 3 >.
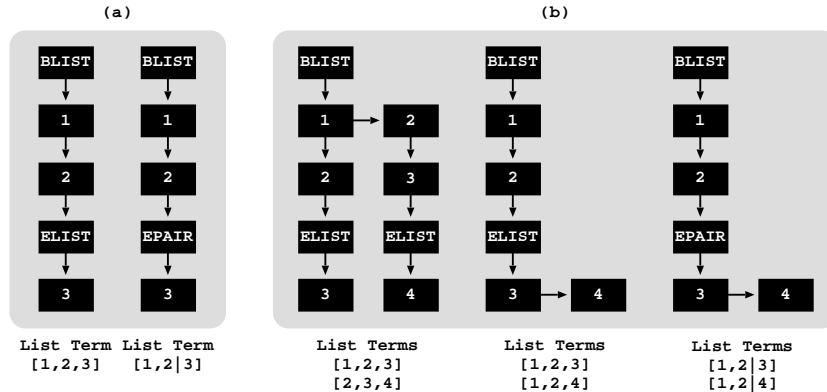

Fig. 7. Trie design for compact lists: final approach

Figure 7(b) illustrates again the same three situations showing how this final approach behaves when we represent more than a list in the same trie. For the three situations, this final approach clearly outperforms all the other representations for standard and compact lists. For lists with the first element different (first situation), it requires $NS + N + 1$ trie nodes for both empty-ending and term-ending lists. For lists with the last element different (second and third situations), it requires $N + S + 1$ trie nodes for both empty-ending and term-ending lists. Table 1 summarizes the comparison between all the approaches regarding the number of trie nodes required to represent in the same trie N list terms of S elements each.

List Terms	*Standard Lists*	*Compact Lists*		
		Initial	*Second*	*Final*
First element different				
$N\ [E_1, ..., E_{S-1}, E_S]$	$2NS + 1$	$NS + 2N + 1$	$NS + 2N + 1$	$NS + N + 1$
$N\ [E_1, ..., E_{S-1}\|E_S]$	$2NS - 2N + 1$	$NS + N + 1$	$NS + N + 1$	$NS + N + 1$
Last element different				
$N\ [E_1, ..., E_{S-1}, E_S]$	$2N + 2S - 1$	$3N + S$	$2N + S + 1$	$N + S + 1$
$N\ [E_1, ..., E_{S-1}\|E_S]$	$N + 2S - 2$	$2N + S$	$N + S + 1$	$N + S + 1$

Table 1. Number of trie nodes to represent in the same trie N list terms of S elements each, using the standard lists representation and the three compact lists approaches

4   Compiled Tries for Compact Lists

We then discuss the implications of the new design in the completed table optimization and describe how we have extended YapTab to support compiled tries for compact lists.

We start by presenting in Fig. 8(a) the compiled trie code for the standard list $[1, 2, 3]$. For standard lists, each PAIR token is compiled using one of the *?_list* trie instructions. At the engine level, these instructions create a new pair term in the heap stack and then they bind the previous tail element of the list being constructed to the new pair.

Figure 8(b) shows the new compiled trie code for compact lists. In the new representation for compact lists, the PAIR tokens were removed. Hence, we need to include the pair terms creation step in the trie instructions associated with the elements in the list, except for the last list element. To do that, we have extended the set of trie instructions for each term type with four new specialized trie instructions: *try_?_in_list*, *retry_?_in_list*, *trust_?_in_list* and *do_?_in_list*. For example, for atom terms, the new set of trie instructions is: *try_atom_in_list*, *retry_atom_in_list*, *trust_atom_in_list* and *do_atom_in_list*. The ELIST tokens are compiled using *?_ending_list* instructions and the BLIST and EPAIR tokens are compiled using *?_void* instructions. At the engine level, the *?_ending_list*
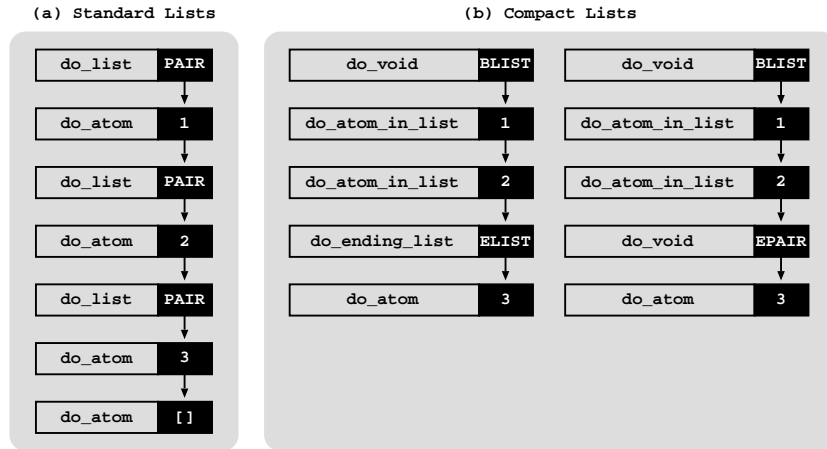
Fig. 8. Comparison between the compiled trie code for standard and compact lists

instructions also create a new pair term in the heap stack to be bound with the previous tail element of the list being constructed. Besides, in order to denote the end of the list, they bind the tail of the new pair with the empty list atom [ ]. The *?_void* instructions do nothing. Note however that the trie nodes for the tokens BLIST and EPAIR cannot be avoided because they are necessary to mark the beginning and the end of list terms when traversing the answer tries bottom-up, and to distinguish between a term t and the list term whose first element is t.

Next we present in Fig. 9, two more examples showing how list terms including compound terms, the empty list term and sub-lists are compiled using the compact lists representation. The tokenized form of the list term $[f(1, 2), [\,], g(a)]$ is the sequence of 8 tokens $<$ BLIST, f/2, 1, 2, [ ], ELIST, g/1, a $>$ and the tokenized form of the list term $[1, [2, 3], [\,]]$ is the sequence of 8 tokens $<$ BLIST, 1, BLIST, 2, ELIST, 3, ELIST, [ ] $>$. To see how the new trie instructions for compact lists are associated with the tokens representing list elements, please consider a tokenized form where the tokens representing common list elements are explicitly aggregated:

$[f(1, 2), [\,], g(a)]$: $<$ BLIST, $<$ f/2, 1, 2 $>$, [ ], ELIST, $<$ g/1, a $>>$
$[1, [2, 3], [\,]]$: $<$ BLIST, 1, $<$ BLIST, 2, ELIST, 3 $>$, ELIST, [ ] $>$.

The tokens that correspond to first tokens in each list element are the ones that need to be compiled with the new *?_in_list* trie instructions (please see Fig. 9 for full details).

5   Experimental Results

We next present some experimental results comparing YapTab with and without support for compact lists. The environment for our experiments was an Intel(R)
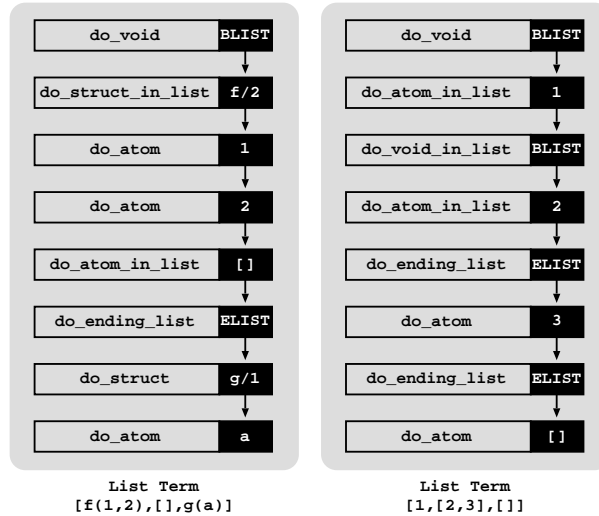
Fig. 9. Compiled trie code for the compact lists $[f(1,2), [\,], g(a)]$ and $[1, [2,3], [\,]]$

Core(TM)2 Quad 2.66GHz with 2 GBytes of main memory and running the Linux kernel 2.6.24-24-generic with YapTab 6.0.0.

To put the performance results in perspective, we have defined a top query goal that calls recursively a tabled predicate `list_terms/1` that simply stores in the table space list terms facts. We experimented the `list_terms/1` predicate using 100,000 list terms of sizes 60, 80 and 100 for empty-ending and term-ending lists with the first and with the last element different.

Tables 2 and 3 show the table memory usage (columns **Mem**), in KBytes, and the running times, in milliseconds, to store (columns **Store**) the tables (first execution) and to load from the tables (second execution) the complete set of answers without (columns **Load**) and with (columns **Cmp**) compiled tries for YapTab using standard lists (column **YapTab**) and using the final design for compact lists (column **YapTab+CL / YapTab**). For compact lists, we only show the memory and running time ratios over YapTab using standard lists. The running times are the average of five runs.

As expected, the memory results obtained in these experiments are consistent with the formulas presented in Table 1. The results in Tables 2 and 3 clearly confirm that the new trie design based on compact lists can decrease significantly memory usage when compared with standard lists. In particular, for empty-ending lists, with the first and with the last element different, and for term-ending lists with the first element different, the results show an average reduction of 50%. For term-ending lists with the last element different, memory usage is almost the same. This happens because the memory reduction obtained in the representation of the common list elements (respectively 59, 79 and 99 elements

Empty-Ending Lists	YapTab				YapTab+CL / YapTab			
	Mem	Store	Load	Cmp	Mem	Store	Load	Cmp
First element different								
100, 000 $[E_1, ..., E_{60}]$	234,375	1036	111	105	**0.51**	**0.52**	**0.71**	**0.69**
100, 000 $[E_1, ..., E_{80}]$	312,500	1383	135	128	**0.51**	**0.52**	**0.73**	**0.64**
100, 000 $[E_1, ..., E_{100}]$	390,625	1733	166	170	**0.51**	**0.53**	**0.67**	**0.55**
Last element different								
100, 000 $[E_1, ..., E_{60}]$	3,909	138	50	7	**0.50**	**0.75**	**0.64**	**0.56**
100, 000 $[E_1, ..., E_{80}]$	3,909	171	71	8	**0.50**	**0.81**	**0.61**	**0.40**
100, 000 $[E_1, ..., E_{100}]$	3,910	211	82	9	**0.50**	**0.76**	**0.62**	**0.44**

Table 2. Table memory usage (in KBytes) and store/load times (in milliseconds) for empty-ending lists using YapTab with and without support for compact lists

Term-Ending Lists	YapTab				YapTab+CL / YapTab			
	Mem	Store	Load	Cmp	Mem	Store	Load	Cmp
First element different								
100, 000 $[E_1, ..., E_{59}|E_{60}]$	230,469	1028	113	97	**0.52**	**0.54**	**0.67**	**0.64**
100, 000 $[E_1, ..., E_{79}|E_{80}]$	308,594	1402	138	134	**0.51**	**0.53**	**0.69**	**0.63**
100, 000 $[E_1, ..., E_{99}|E_{100}]$	386,719	1695	162	163	**0.51**	**0.55**	**0.66**	**0.60**
Last element different								
100, 000 $[E_1, ..., E_{59}|E_{60}]$	1,956	121	45	4	1.00	**0.86**	**0.82**	1.00
100, 000 $[E_1, ..., E_{79}|E_{80}]$	1,956	150	59	4	1.00	**0.88**	**0.72**	1.00
100, 000 $[E_1, ..., E_{99}|E_{100}]$	1,957	194	96	4	1.00	**0.88**	**0.53**	1.00

Table 3. Table memory usage (in KBytes) and store/load times (in milliseconds) for term-ending lists using YapTab with and without support for compact lists

in these experiments) is residual when compared with the number of different last elements (100,000 in these experiments).

Regarding running time, the results in Tables 2 and 3 indicate that compact lists can achieve impressive gains for storing and loading list terms. In these experiments, the storing time using compact lists is around 2 times faster for list terms with the first element different, and around 1.15 to 1.30 times faster for list terms with the last element different. Note that this is the case even for term-ending lists, where there is no significant memory reduction. This happens because the number of nodes to be traversed when navigating the trie data structures for compact lists is considerably smaller than the number of nodes for standard lists.

These results also indicate that compact lists can outperform standard lists for loading terms, both with and without compiled tries, and that the reduction on the running time seems to decrease proportionally to the size of the list terms being considered. The exception is compiled tries for term-ending lists with the last element different, but the 4 milliseconds of the execution time in these experiments is too small to be taken into consideration.

6 Conclusions

We have presented a new and more compact representation of list terms for tabled data that avoids the recursive nature of the WAM representation by removing unnecessary intermediate pair tokens. Our presentation followed the different approaches that we have considered until reaching our current final design. We focused our discussion on a concrete implementation, the YapTab system, but our proposals can be easy generalized and applied to other tabling systems. Our experimental results are quite interesting, they clearly show that with compact lists, it is possible not only to reduce the memory usage overhead, but also the running time of the execution for storing and loading list terms, both with and without compiled tries.

Further work will include exploring the impact of our proposal in real-world applications, such as, the recent works on Inductive Logic Programming [7] and probabilistic logic learning with the ProbLog language [8], that heavily use list terms to represent, respectively, hypotheses and proofs in trie data structures.

Acknowledgements

References

1. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43**(1) (1996) 20–74
2. 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
3. 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
4. Aït-Kaci, H.: Warren's Abstract Machine – A Tutorial Reconstruction. The MIT Press (1991)
5. 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
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
7. Fonseca, N.A., Camacho, R., Rocha, R., Costa, V.S.: Compile the hypothesis space: do it once, use it often. Fundamenta Informaticae **89**(1) (2008) 45–67
8. Kimmig, A., Costa, V.S., Rocha, R., Demoen, B., Raedt, L.D.: On the Efficient Execution of ProbLog Programs. In: International Conference on Logic Programming. Number 5366 in LNCS, Springer-Verlag (2008) 175–189