On the Efficient Implementation of Mode-Directed Tabling

João Santos and Ricardo Rocha

CRACS & INESC TEC and Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021, 4169-007 Porto, Portugal
{jsantos,ricroc}@dcc.fc.up.pt

Abstract. Mode-directed tabling is an extension to the tabling technique that supports the definition of modes for specifying how answers are inserted into the table space. In this paper, we focus our discussion on the efficient support for mode-directed tabling in the YapTab tabling system, which uses *tries* to implement the table space. We discuss 7 different modes and explain how we have extended and optimized YapTab's table space organization to provide engine support for them. Experimental results, in the context of benchmarks taking advantage of mode-directed tabling, show that our implementation compares favorably with the B-Prolog and XSB state-of-the-art tabling systems.

1 Introduction

Tabling [1] is a recognized and powerful implementation technique that solves some limitations of Prolog's operational semantics in dealing with recursion and redundant sub-computations. Tabling based models are able to reduce the search space, avoid looping, and always terminate for programs with the *bounded term-size property*[1]. Tabling consists of saving and reusing the results of sub-computations during the execution of a program and, for that, the calls and the answers to tabled subgoals are stored in a proper data structure called the *table space*. The tabling technique can be viewed as a natural tool to implement dynamic programming algorithms. Dynamic programming is a general recursive strategy that consists in dividing a problem in simple sub-problems that, often, are really the same. Tabling is thus suitable to use with this kind of problems since, by storing and reusing intermediate results while the program is executing, it avoids performing the same computation several times.

In a traditional tabling system, all the arguments of a tabled subgoal call are considered when storing answers into the table space. When a new answer is not a variant[2] of any answer that is already in the table space, then it is always considered for insertion. Therefore, traditional tabling is very good for problems

[1] A logic program has the bounded term-size property if there is a function $f : N \rightarrow N$ such that whenever a query goal Q has no argument whose term size exceeds n, then no term in the derivation of Q has size greater than $f(n)$.

[2] Two terms are considered to be variant if they are the same up to variable renaming.

that require storing all answers. However, with dynamic programming, usually, the goal is to dynamically calculate optimal or selective answers as new results arrive. Writing dynamic programming algorithms can thus be a difficult task without further support. *Mode-directed tabling* [2] is an extension to the tabling technique that supports the definition of *modes* for specifying how answers are inserted into the table space. The idea is to use the modes to define the arguments to be considered for variant checking and to define how variant answers should be tabled regarding the remaining arguments. Mode-directed tabling has proved its viability for applications areas such as Machine Learning [3], Justification [4], Preferences [5], Answer Subsumption [6], among others.

To evaluate a predicate p/n using traditional tabling, we just need to declare it as '*table* p/n'. With mode-directed tabling, tabled predicates are declared using statements of the form '*table* $p(m_1, ..., m_n)$', where the m_i's are modes for the arguments. Implementations of mode-directed tabling are already available in ALS-Prolog [2] and B-Prolog [3], and a restricted form of mode-directed tabling can also be reproduced in XSB Prolog by using *answer subsumption* [7].

In this paper, we focus our discussion on the efficient implementation of mode-directed tabling in the YapTab tabling system [8], which uses *tries* [9] to implement the table space. Our implementation uses a more general approach to the declaration and use of modes and, currently, it supports 7 different modes: *index*, *first*, *last*, *min*, *max*, *sum* and *all*. To the best of our knowledge, no other tabling system supports all these modes and, in particular, the *sum* mode is not supported by any other system. Experimental results, using a set of benchmarks that take advantage of mode-directed tabling, show that our implementation compares favorably with the B-Prolog and XSB state-of-the-art tabling systems. This work is already fully integrated with the latest development version of Yap[3].

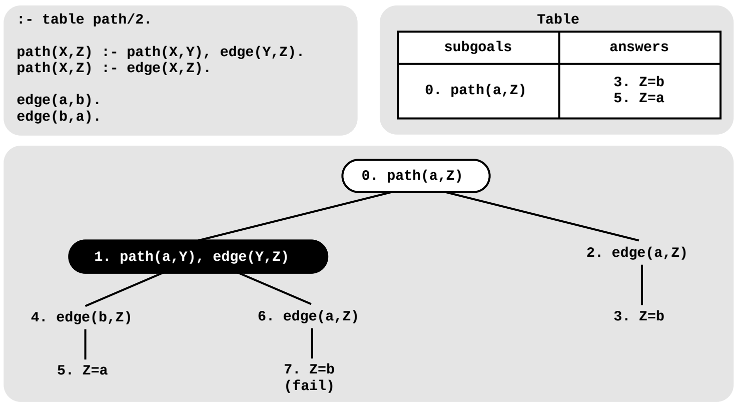
The remainder of the paper is organized as follows. First, we introduce some background concepts about tabling. Next, we describe the modes and we show examples of their use. Then, we introduce YapTab's table space organization and describe how we have extended it to efficiently support mode-directed tabling. At last, we present experimental results and outline some conclusions.

2 Tabled Evaluation

In tabling, variant calls to tabled subgoals are not re-evaluated against the program clauses, instead they are resolved by consuming the answers already stored in the corresponding table entries. During this process, as further new answers are found, they are stored in their tables and later returned to all variant calls.

Figure 1 illustrates the execution of a tabled program. The top left corner shows the program code and the top right corner shows the final state of the table space. The program defines a small directed graph, represented by two *edge/2* facts, with a relation of reachability, defined by a *path/2* tabled predicate. The bottom of the figure shows the evaluation sequence, numbered in order of evaluation, for the query goal *path(a,Z)*.

[3] `http://www.dcc.fc.up.pt/~vsc/Yap`



```
:- table path/2.

path(X,Z) :- path(X,Y), edge(Y,Z).
path(X,Z) :- edge(X,Z).

edge(a,b).
edge(b,a).
```

Table

subgoals	answers
0. path(a,Z)	3. Z=b 5. Z=a

Fig. 1. An example of a tabled evaluation

First calls to tabled subgoals correspond to generator nodes (nodes depicted by white oval boxes) and, for first calls, a new entry, representing the subgoal, is added to the table space (step 0). Next, *path(a,Z)* is resolved against the first matching clause, calling in the continuation *path(a,Y)* (step 1). Since *path(a,Y)* is a variant call to *path(a,Z)*, we do not evaluate the subgoal against the program clauses, instead we consume answers from the table space. Such nodes are called *consumer nodes* (nodes depicted by black oval boxes). However, at this point, the table does not have answers for this call, so the computation is suspended[4].

The only possible move after suspending is to backtrack and try the second matching clause for *path(a,Z)* (step 2). This produces the answer $\{Z{=}b\}$, which is then stored in the table space (step 3). At this point, the computation at node 1 can be resumed with the newly found answer (step 4), giving rise to one more answer, $\{Z{=}a\}$ (step 5). This second answer is then also stored in the table space and propagated to the consumer node (step 6), which produces the answer $\{Z{=}b\}$ (step 7). This answer had already been found at step 3. Tabling does not store duplicate answers in the table space and, instead, variant answers *fail*. This is how tabling avoids unnecessary computations, and even looping in some cases. Since there are no more answers to consume nor more clauses left to try, the table entry for *path(a,Z)* is then marked as *completed*.

3 Mode-Directed Tabling

Traditional tabling can be viewed as a composition of two procedural operations: $Generate()$ and $Aggregate()$. The $Generate()$ operation corresponds to performing tabled evaluation from where a *bag of answers* is generated, i.e., we

[4] We are assuming a *suspension-based tabling* mechanism, where a tabled evaluation can be seen as a sequence of computations that suspend and later resume. Alternatively, *linear tabling* mechanisms use iterative computations to compute fix-points and for that they maintain a single execution tree (no suspension is needed).

may have duplicate (and infinite) answers as in Prolog. The $Aggregate()$ operation then defines the criterion for specifying how answers are tabled which, for traditional tabling, is to eliminate variant answers.

Mode-directed tabling can be thought of as an extension to the $Aggregate()$ operation that allows to define alternative criteria for specifying how variant answers (the index arguments) should be tabled (the output arguments). Index arguments are represented with mode *index*, while arguments with modes *first*, *last*, *min*, *max*, *sum* and *all* represent output arguments. Given the generic declaration $p(m_1, ..., m_j, m_{j+1}, ..., m_n)$, where for $1 <= i <= j$, m_i is an index argument and for $j+1 <= i <= n$, m_i is an output argument, the $Aggregate(p/n)$ operation can be defined as the set of answers:

$$\begin{aligned}\{p(x_1, ..., x_n) \mid \; & \exists(z_{j+1}, ..., z_n) : p(x_1, ..., x_j, z_{j+1} ..., z_n) \in Generate(p/n) \\ & \wedge\, x_{j+1} \in m_{j+1}(Out_{j+1}(x_1, ..., x_j)) \\ & \wedge \ldots \\ & \wedge\, x_n \in m_n(Out_n(x_1, ..., x_{n-1}))\}\end{aligned}$$
$$\text{where } Out_j(x_1, ..., x_{j-1}) = \{y \mid \exists(z_{j+1}, ..., z_n) : \\ p(x_1, ..., x_{j-1}, y, z_{j+1} ..., z_n) \in Generate(p/n)\}$$

For example, consider a $p/3$ predicate declared as $p(index, min, all)$ and the set of answers $\{p(a, 2, 2), p(b, 2, 1), p(b, 1, 2), p(b, 1, 1)\}$. The $Aggreg(p/3)$ operation is then:

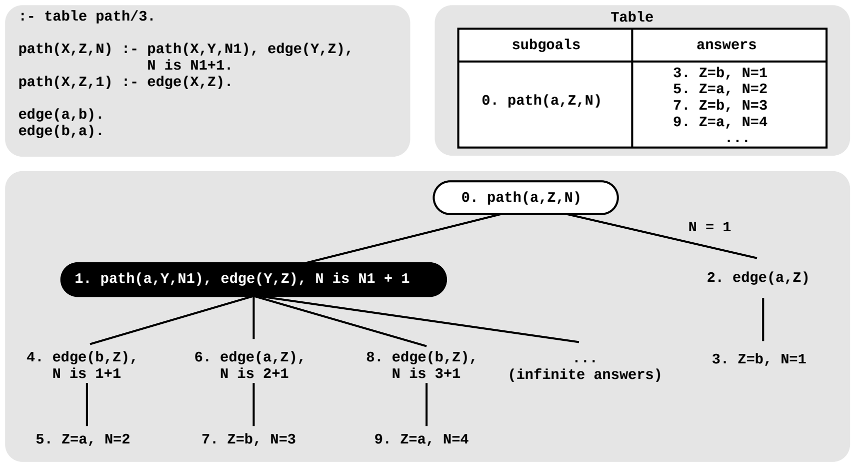
$$\begin{aligned}\{p(x_1, x_2, x_3) \mid \; & \exists(z_2, z_3) : p(x_1, z_2, z_3) \in \{p(a, 2, 2), p(b, 2, 1), p(b, 1, 2), p(b, 1, 1)\} \\ & \wedge\, x_2 \in min(Out_2(x_1)) \wedge\; x_3 \in all(Out_3(x_1, x_2))\}\end{aligned}$$

Since $min(Out_2(a)) = min(\{2\}) = \{2\}$, $all(Out_3(a, 2)) = max(\{2\}) = \{2\}$ and $min(Out_2(b)) = min(\{2, 1\}) = \{1\}$, $all(Out_3(b, 1)) = all(\{2, 1\}) = \{2, 1\}$ then $Aggreg(p/3) = \{p(a, 2, 2), p(b, 1, 2), p(b, 1, 1)\}$.

3.1 Index/First/Last Modes

Starting from the example in Fig. 1, consider now that we modify the program so that it also calculates the number of edges traversed in a path. As we can see in Fig. 2, the program does not terminate. Such behavior occurs because there is a path with an infinite number of edges starting from a, thus not satisfying the bounded term-size property necessary to ensure termination. In particular, the answers found at steps 3 and 7 and at steps 5 and 9 have the same answer for variable *Z* ($\{$*Z=b*$\}$ and $\{$*Z=a*$\}$, respectively), but they are both inserted in the table space because they are not variants for variable *N*. For programs with an infinite number of answers, traditional tabling is thus not enough.

In Fig. 2, the problem relies on the fact that the third argument generates an infinite number of answers. We can thus define the *path/3* predicate as *path(index,index,first)* meaning that only the first and second arguments must be considered for variant checking and that, for the third argument, only the first answer must be tabled. With this declaration, the answer $\{$*Z=b, N=3*$\}$ found at step 7 is no longer inserted in the table space and execution fails.



```
:- table path/3.

path(X,Z,N) :- path(X,Y,N1), edge(Y,Z),
               N is N1+1.
path(X,Z,1) :- edge(X,Z).

edge(a,b).
edge(b,a).
```

Table

subgoals	answers
0. path(a,Z,N)	3. Z=b, N=1 5. Z=a, N=2 7. Z=b, N=3 9. Z=a, N=4 ...

Fig. 2. A tabled evaluation with an infinite number of answers

The *last* mode implements the opposite behavior of the *first* mode, i.e., it always stores the last answer being found and discards the previous one, if any. Remember that with tabling, the order of answers is not important. However, in a particular implementation, the order of answers may depend on the tabling mechanism and on the evaluation strategy being use. Hence, we may question the necessity and/or correctness of the *first* and *last* modes.

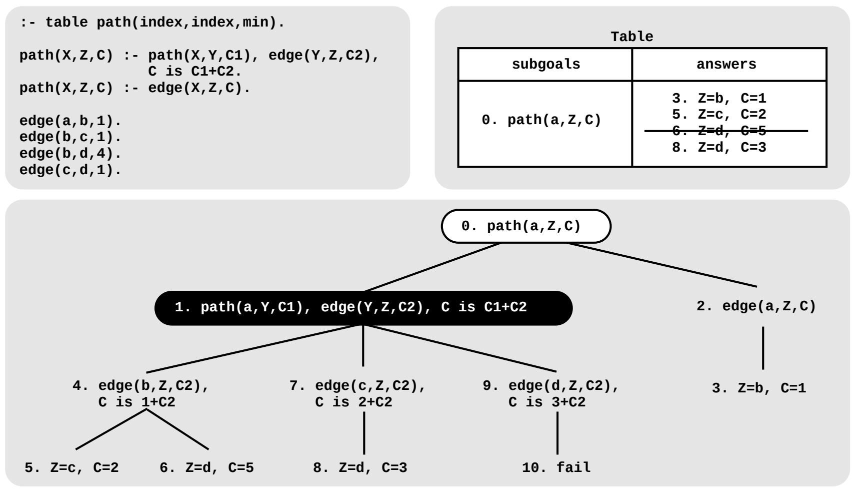
The *first* mode can be seen as a way to prune the search space, once an answer is found. This mode can also be read as *any*, *don't care* or *none*. We adopted the name *first* mainly to reflect the fact that, at the implementation level, we are storing the first answer as a way to represent a justification for that.

On the other hand, the *last* mode can be seen as a way to dynamically compute *preferable answers*. It is usually used in conjunction with a *preferable predicate* that is responsible for computing the preferable answers as new results arrive or fail if no preferable answer exists. In particular, all the other modes can be reproduced by using the *last* mode with appropriate preferable predicates. Please refer to [5, 6] for examples where the *last* mode has shown to be very useful for implementing problems involving Preferences and Answer Subsumption.

3.2 Min/Max Modes

The *min* and *max* modes allow to specify a selective criterion that stores, respectively, the minimal and maximal answers for an argument. At the implementation level, we assume that when using the *min* and/or *max* modes, a tabled predicate is monotonic. Figure 3 shows an example using the *min* mode. The program's goal is to compute the paths with the shortest distances. The *path/3* predicate is declared as *path(index,index,min)*, meaning that the third argument should store only the minimal answers for the first two arguments.

By observing the example in Fig. 3, the most interesting part happens at step 8, where the answer $\{Z{=}d,\ C{=}3\}$ is found. This answer is a variant of



```
:- table path(index,index,min).

path(X,Z,C) :- path(X,Y,C1), edge(Y,Z,C2),
               C is C1+C2.
path(X,Z,C) :- edge(X,Z,C).

edge(a,b,1).
edge(b,c,1).
edge(b,d,4).
edge(c,d,1).
```

Table

subgoals	answers
0. path(a,Z,C)	3. Z=b, C=1 5. Z=c, C=2 ~~6. Z=d, C=5~~ 8. Z=d, C=3

0. path(a,Z,C)
1. path(a,Y,C1), edge(Y,Z,C2), C is C1+C2
2. edge(a,Z,C)
4. edge(b,Z,C2), C is 1+C2
7. edge(c,Z,C2), C is 2+C2
9. edge(d,Z,C2), C is 3+C2
3. Z=b, C=1
5. Z=c, C=2
6. Z=d, C=5
8. Z=d, C=3
10. fail

Fig. 3. Using the *min* mode to compute the paths with the shortest distances

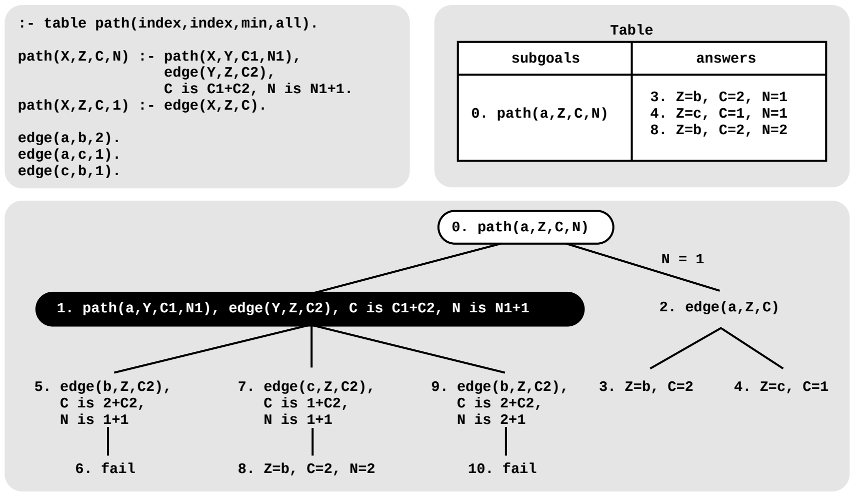
the answer $\{Z{=}d,\ C{=}5\}$ found at step 6. In the previous example, with the *first* mode, the old answer would have been kept in the table. Here, as the new answer is minimal on the third argument, the old answer is replaced by the new answer.

The *max* mode works similarly, but stores the maximal answer instead. For programs without the bounded term-size property, we must be careful when using these two modes as they may not ensure termination. For instance, this would be the case if, in Fig.3, we used the *max* mode instead of the *min* mode.

3.3 Sum/All Modes

Two other modes are the *sum* and the *all*. The *sum* mode allows to sum all the answers for an argument and the *all* mode allows to store all the answers. Consider now the example in Fig. 4 where a *path/4* predicate is declared as *path(index,index,min,all)* meaning that, for each path, we want to store the shortest distance (third argument) and, for the paths with the same shortest distances, the number of edges traversed (fourth argument). By following the example, the most interesting part happens when the answer $\{Z{=}b,\ C{=}2,\ N{=}2\}$ is found at step 8. This answer is a variant of the answer found at step 3 and although both have the same minimal value ($C{=}2$), the new answer is still inserted in the table space since the number of edges (fourth argument) is different.

Notice that when the *sum* or *all* modes are used in conjunction with another mode, like the *min* mode in the example, it is important to keep in mind that the aggregation of answers made for the *sum* or *all* argument depends on the corresponding answer for the *min* argument. Consider, for example, that in the previous example we had found one more answer $\{Z{=}b,\ C{=}1,\ N{=}4\}$. In this case, the new answer would be inserted and the answers $\{Z{=}b,\ C{=}2,\ N{=}1\}$ and $\{Z{=}b,\ C{=}2,\ N{=}2\}$ would be deleted because the new answer corresponds to a shorter distance, as defined by the value $C{=}1$ in the *min* argument.



```
:- table path(index,index,min,all).

path(X,Z,C,N) :- path(X,Y,C1,N1),
                 edge(Y,Z,C2),
                 C is C1+C2, N is N1+1.
path(X,Z,C,1) :- edge(X,Z,C).

edge(a,b,2).
edge(a,c,1).
edge(c,b,1).
```

Table

subgoals	answers
0. path(a,Z,C,N)	3. Z=b, C=2, N=1 4. Z=c, C=1, N=1 8. Z=b, C=2, N=2

0. path(a,Z,C,N)
N = 1
1. path(a,Y,C1,N1), edge(Y,Z,C2), C is C1+C2, N is N1+1
2. edge(a,Z,C)
5. edge(b,Z,C2), C is 2+C2, N is 1+1
7. edge(c,Z,C2), C is 1+C2, N is 1+1
9. edge(b,Z,C2), C is 2+C2, N is 2+1
3. Z=b, C=2
4. Z=c, C=1
6. fail
8. Z=b, C=2, N=2
10. fail

Fig. 4. Using the *all* mode to compute the paths with the shortest distances together with the number of edges traversed

3.4 Related Work

The ALS-Prolog [2] and B-Prolog [3] systems also implement mode-directed tabling but using a different syntax. For example, the *index* and *first* modes are known as + and - and in ALS-Prolog the *all* mode is known as @. The *sum* mode is not supported by any other system and B-Prolog also does not implement the *last* and *all* modes. On the other hand, B-Prolog includes an extra mode, named *nt*, to indicate that a given argument should not be tabled and, thus, not considered to be inserted in the table space. B-Prolog also extends the mode-directed tabling declaration to include a *cardinality limit* that allows to define the maximum number of answers to be stored in the table space [3].

Mode-directed tabling can also be reproduced in the XSB system by using two *answer subsumption* mechanisms [7]. One is called *partial order answer subsumption* and can be used to mimic, in terms of functionality, the *min* and *max* modes. Consider that we want to use it with the program in Fig. 3 that computes the paths with the shortest distances. Then, we should declare the *path/3* predicate as $path(\_, \_, po(< /2))$ meaning that the third argument will be evaluated using partial order answer subsumption, where $< /2$ implements the partial order relation. The other two arguments are considered to be index arguments.

The other XSB's mechanism, called *lattice answer subsumption*, is more powerful and can be used to mimic, in terms of functionality, the other modes. Considering the same example, we only need to change the *path/3* declaration to $path(\_, \_, lattice(min/3))$. The *min/3* predicate has three arguments since, with this mechanism, we must generate a third answer starting from the new answer and from the answer stored in the table:

$$min(Old, New, Res) :- Old < New \rightarrow Res = Old\ ;\ Res = New.$$

4 Implementation

In this section, we start by presenting some background about the table space organization in YapTab and then we discuss in more detail how we have extended it to efficiently support mode-directed tabling.

4.1 YapTab's Table Space Organization

Like we have seen, during the execution of a program, 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 or preferable answer is already in the table and, if not, insert it; and (iii) to load answers from the tables.

With these requirements, a careful 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 [9]. A trie is a tree structure where each different path through the *trie nodes* corresponds to a term described by the tokens labeling the traversed nodes. For example, the tokenized form of the term $path(X, 1, f(Y))$ is the sequence of 5 tokens $path/3$, VAR_0, 1, $f/1$ and VAR_1, where each variable is represented as a distinct VAR_i constant. Two terms with common prefixes will branch off from each other at the first distinguishing token. Consider, for example, a second term $path(Z, 1, b)$. Since the main functor, token $path/3$, and the first two arguments, tokens VAR_0 and 1, are common to both terms, only one additional node will be required to fully represent this second term in the trie, thus allowing to save three nodes in this case.

YapTab implements tables using two levels of tries. The first level, named *subgoal trie*, stores the tabled subgoal calls and the second level, named *answer trie*, stores the answers for a given call. 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 then represented as a unique path in the subgoal trie, starting at the 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. Contrary to subgoal tries, answer trie paths hold just the substitution terms for the free variables that exist in the argument terms of the corresponding call [9].

An example for a tabled predicate $p/3$ is shown in Fig. 5. Initially, the table entry for $p/3$ points to an empty subgoal trie. Then, the subgoal $p(X, 1, Y)$ is called and three trie nodes are inserted to represent the arguments in the call: one for variable X (VAR_0), a second for integer 1, and a last one for variable Y (VAR_1). Since the predicate's functor term is already represented by its table entry, we can avoid inserting an explicit node for $p/3$ in the subgoal trie. Then, the leaf node is set to point to a subgoal frame, from where the answers for the call will be stored. The example shows two answers for $p(X, 1, Y)$: $\{X{=}VAR_0,\ Y{=}f(VAR_1)\}$ and $\{X{=}VAR_0,\ Y{=}b\}$. Since both answers have the same substitution term for argument X, they share the top node in the answer trie (VAR_0). For argument Y, each answer has a different substitution term and, thus, a different path is used to represent each.

When adding answers, the leaf nodes are chained in a linked list in insertion time order, so that the recovery may happen the same way. In Fig. 5, we can observe that the leaf node for the first answer (node VAR_1) points (dashed arrow) to the leaf node of the second answer (node b). To maintain this list, two fields in the subgoal frame data structure point, respectively, to the first and last answer of this list (for simplicity of illustration, these pointers are not shown in Fig. 5). When consuming answers, a consumer node only needs to keep a pointer to the leaf node of its last loaded answer, and consumes more answers just by following the chain. Answers are loaded by traversing the trie nodes bottom-up (again, for simplicity of illustration, such pointers are not shown in Fig. 5).

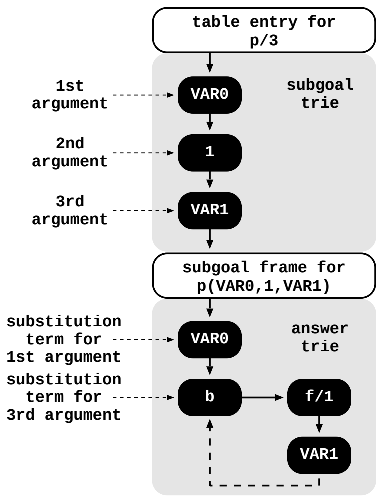


table entry for p/3
subgoal trie
1st argument
VAR0
2nd argument
1
3rd argument
VAR1
subgoal frame for p(VAR0,1,VAR1)
answer trie
substitution term for 1st argument
VAR0
substitution term for 3rd argument
b
f/1
VAR1

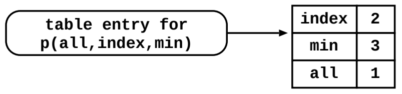
Fig. 5. Table space organization

4.2 Mode-Directed Tabled Subgoal Calls

In YapTab, mode-directed tabled predicates are compiled by extending the table entry data structure to include a *mode array*, where the information about the modes is stored. In this mode array, the modes appear in the order in which the arguments are accessed, which can be different from their position in the original declaration. For example, *index* arguments must be considered first, irrespective of their position. Or, if using the *all* and *min* modes in a declaration, all *min* arguments must be considered before any *all* argument, since the *all* means that all answers must be stored, making meaningless the notion of being minimal in this case. As we will see in Section 4.3, changing the order is also strictly necessary to achieve an efficient implementation. In YapTab, the mode information is thus stored in the order mentioned below, together with the argument's position:

1. arguments with *index* mode;
2. arguments with *min* and *max* mode;
3. arguments with *all* mode;
4. arguments with *last* or *first* or *sum* (only one *sum* argument is allowed) mode (the combination of different modes is not allowed).

Figure 6 shows an example for a *p(all,index,min)* mode-directed tabled predicate. The *index* mode is placed first in the mode array, then the *min* mode and last the *all* mode. With traditional tabling, tabled calls are inserted in their own subgoal tries by following the order of the arguments in the call. With mode-directed tabling,



index	2
min	3
all	1

Fig. 6. Mode array

we follow the order defined in the corresponding mode array. Figure 7 shows the difference between the resulting subgoal tries with and without mode-directed tabling for the subgoal call *p(X,1,Y)*. The values in the mode array indicate that we should start by inserting first the second argument of the subgoal call (1), then the third argument (Y or VAR_0) and last the first argument (X or VAR_1).

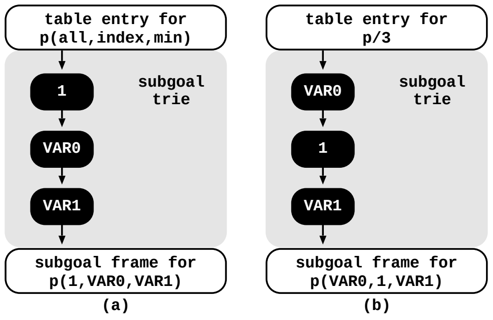


table entry for p(all,index,min)
subgoal trie
1
VAR0
VAR1
subgoal frame for p(1,VAR0,VAR1)
(a)
table entry for p/3
subgoal trie
VAR0
1
VAR1
subgoal frame for p(VAR0,1,VAR1)
(b)

Fig. 7. Subgoal tries for *p(X,1,Y)* considering *p/3* declared (a) with and (b) without mode-directed tabling

The mode information is used when creating the subgoal frame associated with the subgoal call at hand. With mode-directed tabling, subgoal frames were extended to include a new array, named *substitution array*, where the mode information is stored, together with the number of free variables associated with each argument in the subgoal call. The argument's order is the same as in the mode array. Figure 8 shows the substitution array for the subgoal call *p(X,1,Y)*. The first position, corresponding to the argument with constant 1, has no free variables and thus we store a 0 in the substitution array. The other two arguments are free variables and, thus, they have a 1 in the substitution array. It is possible to optimize the array by removing entries that have 0 variables and by joining contiguous entries with the same mode. As we will see next, the substitution array plays an important role in the process of inserting answers in the answer trie.

4.3 Mode-Directed Tabled Answers

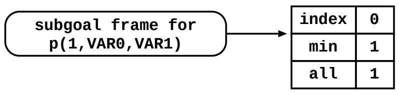


Fig. 8. Substitution array

Like in traditional tabling, tabled answers are only represented by the substitution terms for the free variables in the arguments of the corresponding subgoal call. However, for mode-directed tabling, when we are considering the substitution terms individually, it is important to know beforehand which mode applies to each, and for that, we use the information stored in the corresponding substitution array.

Consider again the substitution array for the subgoal call *p(X,1,Y)*. Now, if we find the answer {*X=f(a), Y=5*}, the first binding to be considered is {*Y=5*} with *min* mode and then {*X=f(a)*} with *all* mode. Please note that the substitutions are considered in the same order that the variables they substitute have been inserted in the subgoal trie. Since the answer trie is initially empty, both terms can be inserted as usual. Later, if another answer is found, for example, {*X=b, Y=3*}, we begin the insertion process by considering the binding {*Y=3*} with *min* mode. As there is already an answer in the table, we must compare both accordingly to the *min* mode. Since the new answer is preferable ($3 < 5$), the old answer must be *invalidated* and the new one inserted in the table. The invalidation process consists in: (a) deleting all intermediate nodes corresponding to the answers being invalidated; and (b) tagging the leaf nodes of such answers as *invalid nodes*. Invalid nodes are only deleted when the table is later completed or abolished. Figure 9 illustrates the aspect of the answer trie before and after the invalidation process.

Invalid nodes are opaque to subsequent subgoal calls, but can be still visible from the consumer calls already in evaluation. Hence, when invalidating a node, we may have consumers still pointing to it. By deleting leaf nodes, this would make consumers unable to follow the chain of answers. An alternative would be to traverse the stacks and update the consumers pointing to invalidated answers, but this could be a very costly operation.

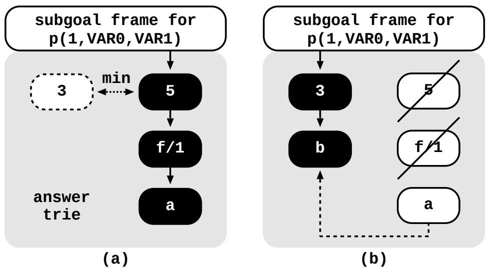

subgoal frame for p(1,VAR0,VAR1)
3
min
5
f/1
answer trie
a
(a)
subgoal frame for p(1,VAR0,VAR1)
3
5
b
f/1
a
(b)

Fig. 9. Aspect of the answer trie (a) before and (b) after the invalidation process

Notice also that the mode's order in the substitution array is crucial for the simplicity and efficiency of the invalidation process. When, at a given node N, we decide that an answer should be invalidated, the substitution array's order ensures that all nodes below node N (including N) are the ones we want to invalidate and that the upper nodes are the ones we want to keep.

This might not be the case if we used a *bad* order. For example, Fig. 10 illustrates the aspect of the answer trie before the invalidation process if we considered the original arguments order for *p(X,1,Y)*. In Fig. 10, to detect that the second answer is preferable ($3 < 5$), we need to navigate in the trie until reaching the leaf node 5 for the first answer. Thus, the invalidation process may require deleting upper nodes (as the example in Fig. 10 shows) and/or traverse several paths to fully detect all preferable answers (this would be the case if we had two intermediate answers with the same minimal values, for instance $\{$*X=f(a), Y=5*$\}$ and $\{$*X=h(c), Y=5*$\}$), making therefore the invalidation process much more complex and costly.

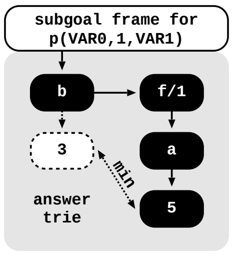


Fig. 10. Before the invalidation process if using a *bad* order

4.4 Scheduling and Mode-Directed Tabling

In a tabled evaluation, there are several points where we may have to choose between continuing forward execution, backtracking, consuming answers or completing subgoals. Such decision is determined by the scheduling strategy. The two most successful strategies are *batched scheduling* and *local scheduling* [10].

Batched scheduling evaluates programs in a depth-first manner as does the WAM. When new answers are found for a particular tabled subgoal, they are added to the table space and the evaluation continues with forward execution. Only when all clauses have been resolved, the newly tabled answers will be forwarded to the consumers. Batched scheduling thus tries to delay the need to move around the search tree by batching the consumption of answers.

Local scheduling is an alternative strategy that tries to complete subgoals sooner. The key idea is that whenever new answers are added to the table space, the execution then fails. Local scheduling thus explores the whole search space for a tabled predicate before returning answers for forward execution.

To the best of our knowledge, YapTab is the only tabling system that supports the dynamic mixed-strategy evaluation of batched and local scheduling within the same evaluation [10]. This is very important, because for mode-directed tabled predicates, the ability of being able to use local evaluation can be crucial to correctly and/or efficiently support some modes.

This is the case for the *sum* mode, that we discuss next in more detail. As it sums all the answers for a given argument, we might end with wrong results if we return partial results instead of aggregating them and only returning the aggregated result. Consider, for example, the two mode-directed tabled predicates *num_links*/2 and *num_nodes*/1 in Fig. 11 and the query goal *num_nodes*(N). If *num_links*/2 is evaluated using local scheduling, we get the right result (*N=3*) but, with batched scheduling, we end with a wrong result (*N=6*). This occurs because, with batched evaluation, the *num_links*(_, _) call in the second clause of *num_nodes*/2 succeeds 2 times for each *edge*/2 fact. Moreover, with batched scheduling, there is no means to return the partial sums while the table is being computed. With local scheduling, since the result is only returned at the end, this problem does not apply.

```
:- table num_links(index, sum).
num_links(A, 0) :- edge(_, A).
num_links(A, 1) :- edge(A, _).

:- table num_nodes(sum).
num_nodes(0).
num_nodes(1) :- num_links(_, _).

edge(a, b).     edge(a, c).     edge(b, c).
```

Fig. 11. A cascade of two mode-directed tabled predicates using the *sum* mode

Batched evaluation can also yield useless computations for mode-directed tabled predicates. Consider a *p*(*max*) tabled predicate and the query goal:

$$:-\ p(Max),\ do\_work(Max, Res).$$

With batched evaluation, the call to *do_work*(Max, Res) will be executed for each Max partial result computed by $p(Max)$, hence producing many useless computations as the number of non-maximal results.

5 Experimental Results

In this section, we present some experimental results for a set of benchmarks that take advantage of mode-directed tabling. The environment for our experiment was a machine with a AMD FX(tm)-8150 8-core processor with 32 GBytes of main memory and running the Linux kernel 64 bits version 3.2.0. To put our results in perspective, we compare our implementation, on top of Yap Prolog (development version 6.3), with the B-Prolog (version 7.8 beta-6) and the XSB (version 3.3.6) systems, both using local scheduling. For XSB, we adapted the benchmarks to use lattice answer subsumption (as discussed in Section 3.4)[5]. For benchmarking, we used the following set of programs:

short(N) uses the *min* mode to determine all-pairs shortest paths in a graph representing the flight connections between the N busiest commercial airports in US[6].

short_first(N) uses the *first* mode to extend the all-pairs shortest paths program to also include the first justification for each shortest path.

short_all(N) uses the *all* mode to extend the all-pairs shortest paths program to also include all the justifications for each shortest path.

short_pref(N) uses the *last* mode to solve the all-pairs shortest paths program using Preferences [6].

knapsack(N) uses the *max* mode to determine the maximum number of items to include in a collection, from N weighted items, so that the total weight is equal to a given value.

lcs(N) uses the *max* mode to find the longest subsequence common to two different sequences of size N.

matrix(N) uses the *min* mode to implement the matrix chain multiplication problem that determines the most efficient way to multiply a sequence of N matrices.

pagerank(N) uses the *sum* mode to measure the rank values of web pages in a realistic dataset of web links called *search engines*[7], using N iterations.

Table 1 shows the execution times, in milliseconds, for running the benchmarks with YapTab, B-Prolog and XSB. In parentheses, it also shows the execution time ratios against YapTab with local evaluation. The execution times are the average of 3 runs. The entries marked with *n.a.* correspond to programs using modes not available in B-Prolog. The ratios marked with (—) mean that we are *not considering* them in the average results (they correspond either to *n.a.* entries or to execution times much higher than YapTab).

In addition to these results, we also collected some statistics for YapTab when running with local and batched evaluation. Table 2 shows the number of answer trie nodes (column **#nodes**) and the number of tabled answers (column **#ans**) present in the table space for YapTab at the end of the execution (columns **Final**) and the respective differences for the full execution with local and batched evaluation (columns **Extra/Deleted**). These differences represent the extra trie nodes and answers that were allocated/found during the evaluation and later deleted and, thus, are not present in the final tables.

[5] For programs using *min*/*max* modes, we also tried with partial order answer subsumption but, unexpectedly, we got worse results.

[6] `http://toreopsahl.com/datasets`

[7] `http://www.cs.toronto.edu/~tsap/experiments/download/download.html`

Table 1. Execution times, in milliseconds, for YapTab, B-Prolog and XSB and the respective ratios when compared with YapTab's local evaluation

Programs	**YapTab**		**B-Prolog**	**XSB**
	Local	**Batched**		
short(300)	1,088	1,261 (1.16)	2,990 (2.75)	2,922 (2.69)
short(400)	1,544	1,785 (1.16)	4,216 (2.73)	4,321 (2.80)
short(500)	2,170	2,472 (1.14)	5,792 (2.67)	6,218 (2.87)
short_first(300)	1,394	2,641 (1.89)	3,225 (2.31)	5,013 (3.60)
short_first(400)	2,052	3,432 (1.67)	4,614 (2.25)	7,257 (3.54)
short_first(500)	2,866	4,488 (1.57)	6,379 (2.23)	10,328 (3.60)
short_all(300)	4,324	8,383 (1.94)	*n.a.* (—)	61,803 (—)
short_all(400)	5,861	10,590 (1.81)	*n.a.* (—)	122,985 (—)
short_all(500)	8,337	13,598 (1.63)	*n.a.* (—)	239,451 (—)
short_pref(300)	2,882	4,241 (1.47)	*n.a.* (—)	6,666 (2.31)
short_pref(400)	4,152	5,621 (1.35)	*n.a.* (—)	9,932 (2.39)
short_pref(500)	5,773	7,473 (1.29)	*n.a.* (—)	14,129 (2.45)
knapsack(1000)	1,013	998 (0.99)	837 (0.83)	2,684 (2.65)
knapsack(1500)	1,581	1,561 (0.99)	1,229 (0.78)	3,977 (2.52)
knapsack(2000)	2,037	2,040 (1.00)	1,582 (0.78)	5,473 (2.69)
lcs(1000)	1,196	1,170 (0.98)	2,900 (2.42)	3,060 (2.56)
lcs(1500)	2,768	2,722 (0.98)	5,784 (2.09)	7,128 (2.58)
lcs(2000)	4,864	4,804 (0.99)	10,116 (2.08)	13,338 (2.74)
matrix(100)	192	224 (1.17)	582 (3.03)	396 (2.06)
matrix(150)	925	1,076 (1.16)	2,549 (2.76)	1,610 (1.74)
matrix(200)	3,005	3,534 (1.18)	7,816 (2.60)	4,688 (1.56)
pagerank(1)	365	*n.a.* (—)	*n.a.* (—)	128,377 (—)
pagerank(16)	813	*n.a.* (—)	*n.a.* (—)	> 10 *min* (—)
pagerank(36)	1,260	*n.a.* (—)	*n.a.* (—)	> 10 *min* (—)
Average ratio		(1.31)	(2.15)	(2.63)

In general, the results show that, for all combinations of experiments and systems, there is no clear tendency showing that the execution time ratios increase or decrease as we increase the size of the corresponding set of programs.

Comparing the results for local and batched evaluation, they show that, on average, batched evaluation is around 31% worse than local evaluation. These results are confirmed in Table 2, where we can observe that batched evaluation always allocates/deletes more trie nodes and inserts/deletes more tabled answers than local evaluation. In particular, batched evaluation gets worse the more answers are inserted into the table space. This affects in particular the **short_first()**, **short_all()** and **short_pref()** set of programs, which confirms our discussion regarding the fact that, in general, local evaluation is more suitable to reduce the search space for mode-directed tabled predicates.

Regarding the comparison with the other systems, YapTab's results clearly outperform those of B-Prolog and XSB. On average, B-Prolog and XSB are, respectively, around 2.15 and 2.63 times worse than YapTab using local evaluation. Please note that for B-Prolog and XSB we do not include the performance of some programs into the average results. For B-Prolog, this is because these pro-

Table 2. Number of answer trie nodes and tabled answers for YapTab at the end of the execution and the respective differences (extra nodes and answers allocated/found that were later deleted) for the full execution with local and batched evaluation

Programs	Final		Extra/Deleted			
			Local		Batched	
	#nodes	#ans	#nodes	#ans	#nodes	#ans
short(300)	179,401	89,401	77,911	77,911	586,488	586,488
short(400)	317,618	157,618	122,435	122,435	706,060	706,060
short(500)	500,000	250,000	196,831	196,831	877,913	877,913
short_first(300)	661,458	89,401	586,348	77,911	16,476,991	586,488
short_first(400)	1,213,352	157,618	947,584	122,435	18,733,939	706,060
short_first(500)	1,997,262	250,000	1,609,053	196,831	21,760,014	877,913
short_all(300)	2,615,740	690,614	5,609,890	1,584,000	30,418,627	4,740,397
short_all(400)	4,351,566	1,084,942	8,129,237	2,172,438	35,762,267	5,632,706
short_all(500)	6,806,102	1,611,082	12,039,458	3,017,929	43,281,969	6,835,251
short_pref(300)	179,401	89,401	77,911	77,911	586,488	586,488
short_pref(400)	317,618	157,618	122,435	122,435	706,060	706,060
short_pref(500)	500,000	250,000	196,831	196,831	877,913	877,913
knapsack(1000)	1,960,131	973,453	87,816	87,816	307,055	307,055
knapsack(1500)	2,963,665	1,475,220	109,613	109,613	450,276	450,276
knapsack(2000)	3,960,969	1,973,872	127,957	127,957	584,980	584,980
lcs(1000)	1,980,191	989,118	101,997	101,997	206,485	206,485
lcs(1500)	4,445,865	2,221,466	234,713	234,713	484,700	484,700
lcs(2000)	7,917,402	3,956,741	420,051	420,051	866,027	866,027
matrix(100)	10,100	5,050	11,089	11,089	14,862	14,862
matrix(150)	22,648	11,324	17,791	17,791	39,775	39,775
matrix(200)	40,194	20,097	36,325	36,325	68,848	68,848
pagerank(1)	85,111	30,896	1,825,175	1,240,703	*n.a.*	*n.a.*
pagerank(16)	378,783	104,314	3,237,305	1,711,413	*n.a.*	*n.a.*
pagerank(36)	741,343	194,954	4,828,085	2,241,673	*n.a.*	*n.a.*

grams use the *all*, *last* and *sum* modes, which are not supported in B-Prolog. For XSB, the execution times for the **short_all()** and **pagerank()** are much higher than YapTab and including them would have distorted the comparison between the three systems. To the best of our knowledge, YapTab is thus the only system that supports the *all*, *last* and *sum* modes and handles them efficiently.

6 Conclusions

We discussed how we have extended YapTab's table space organization to provide engine support for mode-directed tabling. In particular, we presented how we deal with mode-directed tabled subgoal calls and answers and we discussed the role of scheduling in mode-directed tabled evaluations. Our implementation uses a more general approach to the declaration and use of modes and, currently, it supports 7 different modes. To the best of our knowledge, no other tabling system supports all these modes and, in particular, the *sum* mode is not supported by any other system.

Experimental results on benchmarks that take advantage of mode-directed tabling, showed that our implementation clearly outperforms the B-Prolog and XSB state-of-the-art tabling systems. In particular, YapTab is the only system that efficiently handles programs that use the *all* mode.

Further work will include extending our implementation to support multi-threaded mode-directed tabling and explore the impact of applying mode-directed tabling to other problems.

Acknowledgments

This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) within projects LEAP (FCOMP-01-0124-FEDER-015008) and HORUS (FCOMP-01-0124-FEDER-010074). João Santos is funded by the FCT grant SFRH/BD/76307/2011.

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. Guo, H.F., Gupta, G.: Simplifying Dynamic Programming via Mode-directed Tabling. Software Practice and Experience **38**(1) (2008) 75–94
3. Zhou, N.F., Kameya, Y., Sato, T.: Mode-Directed Tabling for Dynamic Programming, Machine Learning, and Constraint Solving. In: IEEE International Conference on Tools with Artificial Intelligence. Volume 2., IEEE Computer Society (2010) 213–218
4. Pemmasani, G., Guo, H.F., Dong, Y., Ramakrishnan, C.R., Ramakrishnan, I.V.: Online Justification for Tabled Logic Programs. In: International Symposium on Functional and Logic Programming. Number 2998 in LNCS, Springer-Verlag (2004) 24–38
5. Guo, H.F., Jayaraman, B., Gupta, G., Liu, M.: Optimization with Mode-Directed Preferences. In: 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, ACM (2005) 242–251
6. Santos, J., Rocha, R.: Mode-Directed Tabling and Applications in the YapTab System. In: Symposium on Languages, Applications and Technologies. (2012) 25–40
7. Swift, T., Warren, D.S.: Tabling with Answer Subsumption: Implementation, Applications and Performance. In: European Conference on Logics in Artificial Intelligence. Number 6341 in LNAI, Springer-Verlag (2010) 300–312
8. 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
9. 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
10. Rocha, R., Silva, F., Santos Costa, V.: Dynamic Mixed-Strategy Evaluation of Tabled Logic Programs. In: International Conference on Logic Programming. Number 3668 in LNCS, Springer-Verlag (2005) 250–264