Mode-Directed Tabling and Applications in the YapTab System

João Santos and Ricardo Rocha

CRACS & INESC TEC, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
`{jsantos,ricroc}@dcc.fc.up.pt`

Abstract

Tabling is an implementation technique that solves some limitations of Prolog's operational semantics in dealing with recursion and redundant sub-computations. Tabling works by memorizing generated answers and then by reusing them on similar calls that appear during the resolution process. In a traditional tabling system, all the arguments of a tabled subgoal call are considered when storing answers into the table space. Traditional tabling systems are thus very good for problems that require finding all answers. Mode-directed tabling is an extension to the tabling technique that supports the definition of selective criteria for specifying how answers are inserted into the table space. Implementations of mode-directed tabling are already available in systems like ALS-Prolog, B-Prolog and XSB. In this paper, we propose a more general approach to the declaration and use of mode-directed tabling, implemented on top of the YapTab tabling system, and we show applications of our approach to problems involving Justification, Preferences and Answer Subsumption.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Tabling, Mode Operators, Applications

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.25

1 Introduction

Logic programming languages, such as Prolog, provide a high-level, declarative approach to programming. The operational semantics of Prolog is given by SLD resolution, an evaluation strategy particularly simple that matches stack based machines particularly well, but that suffers from fundamental limitations, such as in dealing with recursion and redundant sub-computations. Tabling [1] is a recognized and powerful implementation technique that solves such limitations in a very elegant way. Tabling based models are able to reduce the search space, avoid looping, and always terminate for programs with the *bounded term-size property*[1]. Work on tabling, as initially implemented in the XSB system [11], proved its viability for application areas such as Natural Language Processing, Knowledge Based Systems, Model Checking, Program Analysis, among others. Currently, tabling is widely available in systems like XSB, Yap, B-Prolog, ALS-Prolog, Mercury and Ciao.

In a nutshell, 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

[1] A logic program has the bounded term-size property if there is a function $f : N \to N$ 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 $f(n)$.

is a general recursive strategy that consists in dividing a problem in simple sub-problems that, after solved, will constitute the final solution for the main problem. Often, many of these sub-problems are really the same. The dynamic programming approach seeks to solve each sub-problem only once, hence reducing the number of computations. 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 calculating 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 systems are very good for problems that require finding 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* [4] is an extension to the tabling technique that supports the definition of selective criteria for specifying how answers are inserted into the table space. The idea of mode-directed tabling is to use *mode operators* to define what arguments should be used in variant checking in order to select what answers should be tabled. The features added by these operators can then be elegantly applied, not only to dynamic programming problems, but also to problems related to Machine Learning [15], Justification [10, 7], Preferences [2, 5, 6], Answer Subsumption [8], among others.

In a traditional tabling system, to evaluate a predicate using tabling, we just need to declare it as '*table* p/n', where p is the predicate name and n its arity. With mode-directed tabling, tabled predicates are declared using statements of the form '*table* $p(m_1, ..., m_n)$', where p is the tabled predicate and the m_i's are the mode operators for the arguments. Implementations of mode-directed tabling are already available in systems like ALS-Prolog [4], B-Prolog [15] and XSB [14]. In this paper, we propose a more general approach to the declaration and use of mode operators, implemented on top of the YapTab tabling system [9], and we show applications of our approach to problems involving Justification, Preferences and Answer Subsumption.

The remainder of the paper is organized as follows. First, we introduce some background concepts about tabling. Next, we describe the mode operators that we propose and we show small examples of their use. Then, we address the use of such operators in three application areas. We start by discussing how these operators can be used in the generation of justifications for the answers of a program. Next, we discuss the set of transformations that allow preference problems to be implemented using mode operators and, last, we discuss the implementation of an answer subsumption mechanism using mode operators.

2 Tabled Evaluation

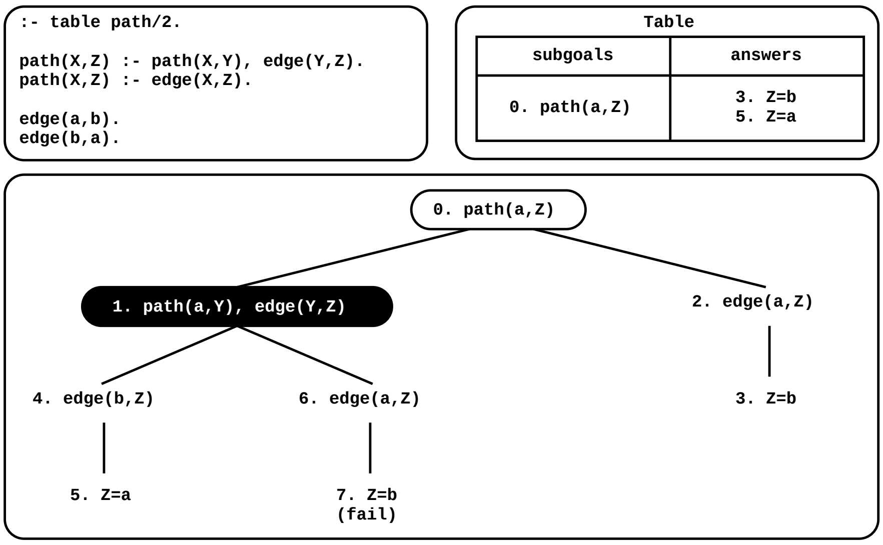
The basic idea behind tabling is straightforward: programs are evaluated by storing answers for tabled subgoals in an appropriate data space, called the *table space*. Similar calls to tabled subgoals are not re-evaluated against the program clauses, instead they are resolved by consuming the answers 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 similar calls. Within this model, the nodes in the search space are classified as either: *generator*

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

nodes, corresponding to first calls to tabled subgoals; *consumer nodes*, corresponding to similar calls to tabled subgoals; or *interior nodes*, corresponding to non-tabled subgoals.

Figure 1 illustrates the execution of a tabled program. The top left corner of the figure 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, given by a *path/2* predicate, that computes the transitive closure in a graph. The declaration *:- table path/2* specifies that *path/2* should be evaluated using tabling.

The bottom of the figure shows the evaluation sequence for the query goal *path(a,Z)*. Note that traditional Prolog would immediately enter an infinite loop because the first clause of *path/2* leads to a variant call to *path(a,Z)*. In contrast, if tabling is applied then termination is ensured. Next, we describe in more detail how this query is solved with tabled evaluation.



```
:- 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

Figure 1 An example of a tabled evaluation.

The execution starts with the subgoal call *path(a,Z)*. First calls to tabled subgoals correspond to generator nodes (generator nodes are 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 in the program 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* (consumer nodes are depicted by black oval boxes). However, at this point, the table does not have answers for this call, so the computation is suspended[3]

The only possible move after suspending is to backtrack and try the second matching clause for *path(a,Z)* (step 2). This originates 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

[3] For the sake of simplicity, we are assuming a *suspension-based tabling* mechanism [11], where a tabled evaluation can be seen as a sequence of sub-computations that suspend and later resume. Alternatively, *linear tabling* mechanisms [16, 3] use iterative computations to compute fix-points and for that they maintain a single execution tree without requiring suspension and resumption of sub-computations.

answer is then also inserted in the table space and propagated to the consumer node (step 6), which originates 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. Instead, repeated answers *fail*. This is how tabling avoids unnecessary computations, and even looping in some cases. A new answer is inserted in table space only if it is not a variant of any answer that is already there. Since there are no more answers to consume nor more clauses left to try, the evaluation ends and the table entry for *path(a,Z)* can be marked as *completed*.

3 Mode Operators

This section describes the mode operators that we are proposing and how they can be used in our implementation done on top of YapTab [9], a tabling system that extends Yap's engine [12] to support tabled evaluation for definite programs.

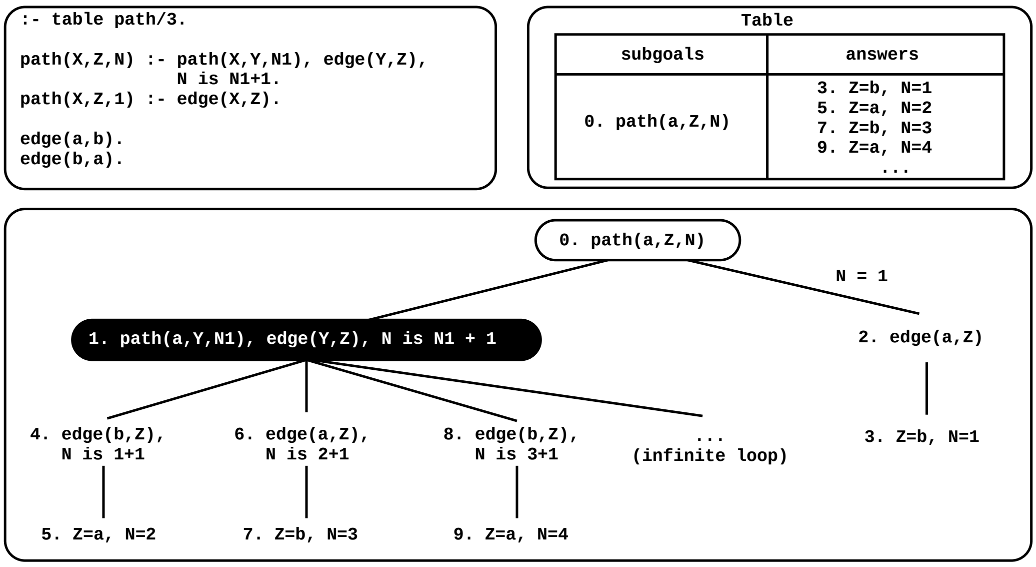
As mentioned before, in a traditional tabling system, to evaluate a predicate using tabling, we just write at the top of the program '*table p/n*'. With mode-directed tabling [4], tabled predicates are declared using statements of the form '*table* $p(m_1, ..., m_n)$', where the m_i's are mode operators for the arguments. We have defined 6 different mode operators: *index*, *min*, *max*, *first*, *last* and *all*. Arguments with modes *min*, *max*, *first*, *last* or *all* are assumed to be output arguments and only *index* arguments are considered for variant checking. After an answer be generated, the system tables the answer only if it is *better*, accordingly to the meaning of the output arguments, than some existing variant answer. Next, we describe in more detail how these mode operators work and we show some small examples of their use in the YapTab system.

3.1 *First* Mode Operator

Starting from the example in Fig. 1, consider now that we modify the program so that it also calculates the number of edges that are traversed in a path. Figure 2 illustrates the execution of this new program. As we can see, even with tabling, the program enters an infinite loop. Such behavior occurs because there is a path with an infinite number of edges starting from *a*, thus not verifying the bounded term-size property necessary to ensure termination. In particular, the answers found in steps 3 and 7 and in 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, since we may need to specify a selective criteria to restrict the number of answers to a finite set.

As we will see next, by using tabling with mode operators, termination can be ensured for programs with an infinite number of answers. The example in Fig. 2 can be solved by using the *index* and *first* mode operators. As already mentioned, the mode *index* means that only the given arguments must be considered for variant checking. The mode *first* means that only the first answer for those arguments must be stored.

Knowing that the problem with the program in Fig. 2 resides on the fact that the third argument generates an infinite number of answers, we can thus define this argument to have mode *first* and the others to have mode *index*. Figure 3 illustrates this modification and the new execution tree and table space. If we compare both programs, the only difference is the declaration of the *path/3* predicate that is now *path(index,index,first)*. By observing Fig. 3, we can see that the answer {*Z=b, N=3*} (step 7) is no longer inserted in the table. That happens because, with the *first* mode on the third argument, the answer {*Z=b, N=1*} found at step 3 is considered a variant of the answer {*Z=b, N=3*} found at step 7.



```
:- 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 ...

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

3.2 *Min*/*Max* Mode Operators

The mode operator *first* only allows to store the first answer found for the respective argument. It would be interesting if we can also define other criteria to specify which answers we would like to store. For that we introduce two new mode operators, *min* and *max*, that allow to store, respectively, the minimal and the maximal answers found for an argument. To better understand the effect of these operators, let us consider the example in Fig. 4.

The program defines again a small directed graph with a relation of reachability, but now the edge facts include a third argument with a cost. The program's goal is to compute the paths with the lowest costs. To do that, the *path/3* predicate is declared as *path(index,index,min)*, meaning that the third argument should store only the minimal answers related with the first two arguments.

By observing the example in Fig. 4, we can see that the execution tree follows the normal evaluation of a tabled program and that the answers are stored as they are found. The most interesting part happens at step 8, where the answer $\{Z{=}d,\ C{=}3\}$ is found. This answer is a variant of the answer $\{Z{=}d,\ C{=}5\}$ found at step 6. In the previous example, with the *first* operator, 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 operator works similarly to the *min* mode operator, but stores the maximal answer instead. In any case, we must be careful when using these two mode operators as they may not ensure termination for programs without the bounded term-size property. For instance, this would be the case if, in the example of Fig.4, we used the *max* mode instead of the *min* mode operator.

3.3 *All* Mode Operator

Another mode operator that can be useful is the *all* operator, that allows us to store all the answers for a given argument. Consider, for example, the program in Fig. 5, that is a mix of the programs in Fig. 3 and Fig. 4. In this new program, the path predicate is declared as *path(index,index,min,all)* meaning that, for each path, we want to store the lowest cost of

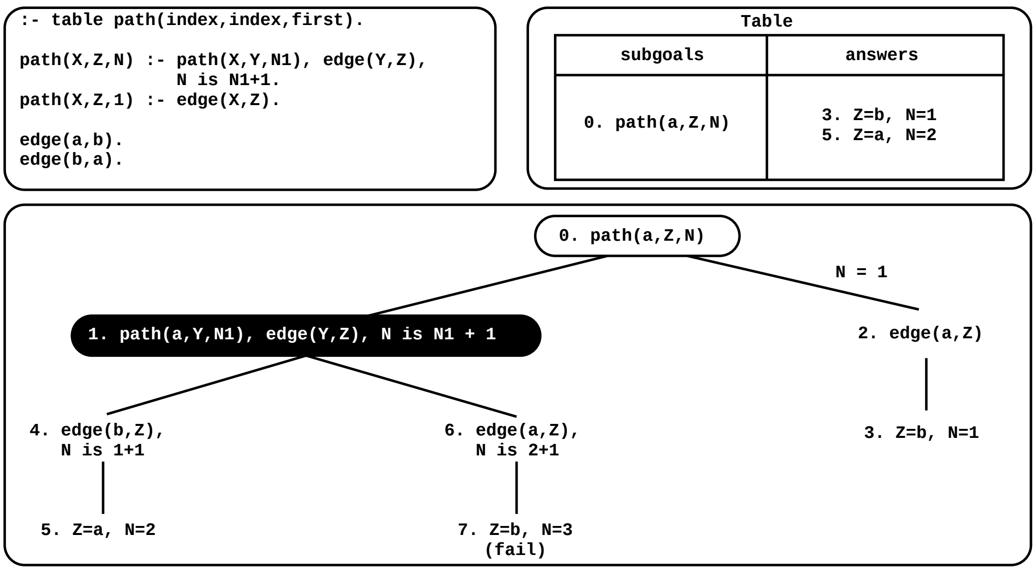


Figure 3 Using tabling with the mode operator *first*.

the path (the third argument) and, at the same time, we want to store the number of edges traversed, for all paths with minimal cost (the fourth argument).

The execution tree for the program in Fig. 5 is similar to the previous ones. 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 $\{Z{=}b,\ C{=}2,\ N{=}1\}$ 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) are different.

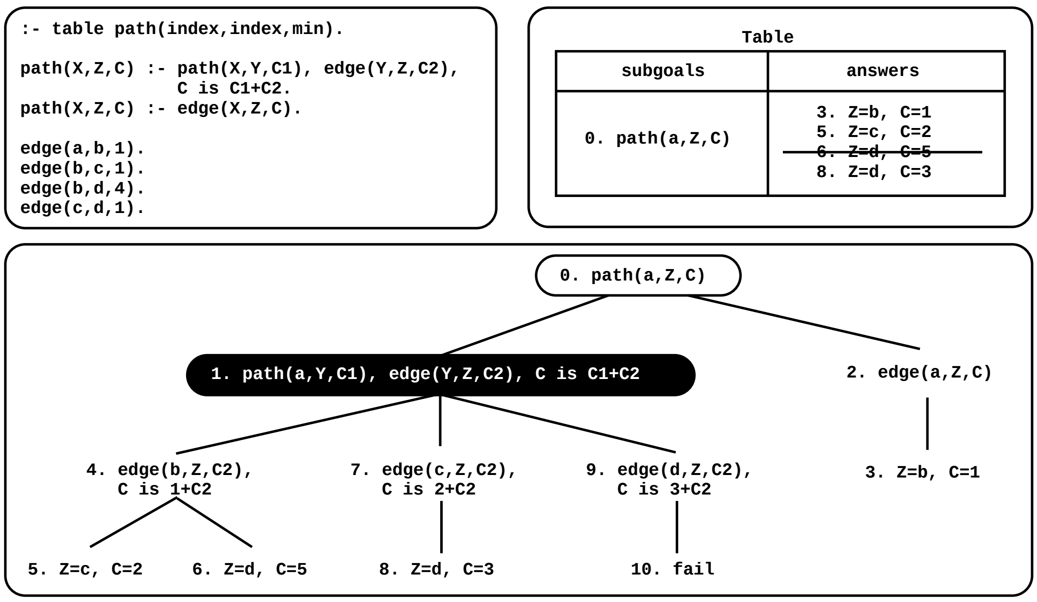
Notice that when the *all* operator is used in conjunction with another mode operator, like the *min* operator in the example, it is important to keep in mind that the aggregation of answers made for the *all* argument depends on the corresponding answer for the *min* argument. Consider, for example, 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 path, as defined by the value $C{=}1$ in the *min* argument.

3.4 *Last* Mode Operator

Finally, we introduce the *last* mode operator. The *last* operator implements the opposite behaviour of the *first* operator. In other words, it always stores the last answer being found and deletes the previous one, if any. As we will see in the next sections, this operator is very useful for implementing problems involving Preferences and Answer Subsumption.

3.5 Related Work

The ALS-Prolog [4] and B-Prolog [15] systems also implement mode-directed tabling using a very similar syntax. Some operators, however, have different names in those systems. For example, the operators *index*, *first* and *all* are known as *+*, - and *@*, respectively. B-Prolog has an extra operator, named *nt*, to indicate that a given argument should not be tabled and, thus, not considered to be inserted in the table space.



```
:- 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

Figure 4 Using tabling with the mode operator *min*.

B-Prolog also extends the mode-directed tabling declaration to include a cardinality limit C that allows to define the maximum number of answers to be stored in the table space:

$$:-\ table\ p(m_1, ..., m_n) : C$$

Until the C limit be reached, all the answers generated are inserted in the table. After that, if a preferable answer is generated, it replaces the less preferable answer in the table. As we will see, this behavior can be recreated by using answer subsumption.

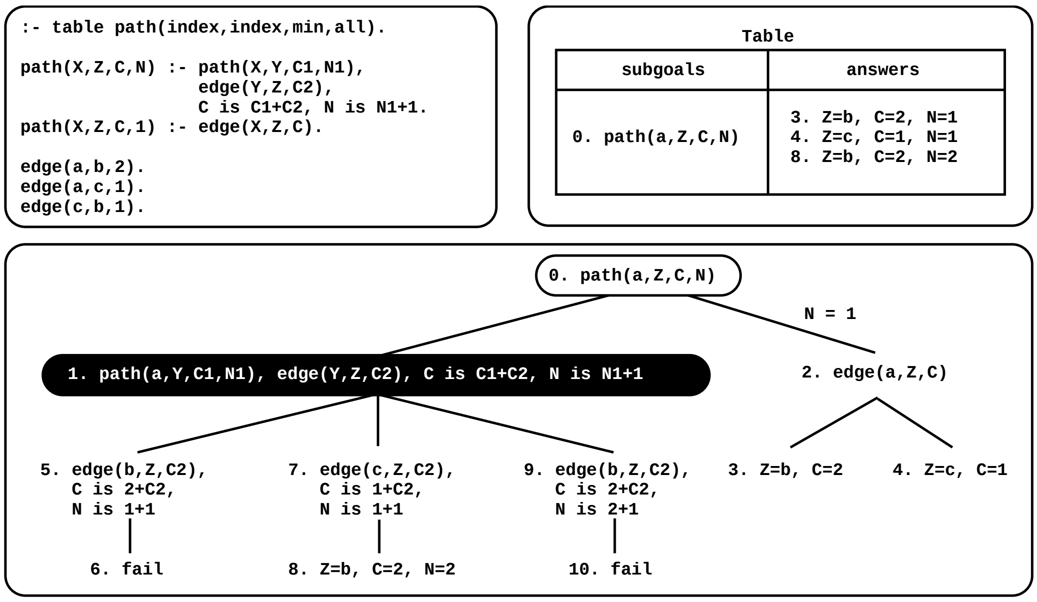
4 Mode-Directed Tabling and Justification

When the execution of a program generates a certain amount of answers, it could be interesting to have access to *proofs* showing that those answers are indeed correct. The process of generating proofs to answers is called *Justification* [10, 7].

Consider again the path problem, the justification for an answer could be, for example, the list of nodes traversed. Despite this apparent simplicity, generating justifications could be a very difficult task.

There are two main approaches for generating justifications. The first is called *post-processing justification* [10]. It has the advantage of being totally independent of the execution of the program but has the disadvantage of needing the program to be run twice: one to generate the answers and another to generate the justifications. The second approach is called *on-line justification* [7] and is done during program's execution, so it is much faster than the post-processing approach. Next, we discuss on-line justification in more detail.

Pemmasani et al. [7] proposed a technique for on-line justifications that involves the transformation of the original program. Figure 6 shows the result of applying such transformation to the program in Fig. 1. As we can see, the main difference between both programs is the inclusion of a new predicate, *store_evid/2*, at the end of the *path/2* clauses. This new predicate stores the justifications and guarantees that repeated answers are stored only once, thus avoiding infinite loops.



```
:- 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

Figure 5 Using tabling with the mode operator *all*.

```
path(X,Z) :- path(X,Y), edge(Y,Z), store_evid(path(X,Z),
                          [((path(X,Y),true),ref(path(X,Y))),((edge(Y,Z,true),[])]).
path(X,Z) :- edge(X,Z), store_evid(path(X,Z),[((edge(X,Z),true),[])]).

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

Figure 6 On-line justification with the *store_evid/2* predicate.

Mode-directed tabling can also be used to generate on-line justifications [4]. Figure 7 shows how the program in Fig. 1 can be modified to implement on-line justifications using mode-directed tabling in the YapTab system.

The modifications are minimal. The path predicate includes an extra argument to represent the justifications and its declaration is modified to use mode operators. The built-in predicate *append/3* is also used to generate the list of nodes defining a path. With this, the task of avoiding infinite loops and controlling the generation of justifications is now delegated to the operators *index* and *first*, since they only store the first answer found for a certain path. For example, executing the query *path(a,Z,J)*, we would get the answers {*Z=b, J=[(a,b)]*} and {*Z=a, J=[(a,b),(b,a)]*}, where variable *J* represents the justification for the corresponding path.

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

path(X,Z,J) :- path(X,Y,J1), edge(Y,Z), append(J1,[(Y,Z)],J).
path(X,Z,[(X,Z)]) :- edge(X,Z).

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

Figure 7 On-line justification with mode-directed tabling.

5 Mode-Directed Tabling and Preferences

Logic programming is commonly used to solve optimization problems. When we write that kind of programs, it could be difficult to define the solution as a simple maximization or minimization problem. Preference Logic Programming (or Preferences) tries to solve this issue by dividing the program into the *specification of the problem* and the *definition of the optimal solution*. A very simple and declarative syntax for Preferences is the one proposed by Govindarajan et al. [2]. To better understand its syntax, let us consider the example in Fig. 8.

```
% problem specification
path(X,Z,C,D) :- path(X,Y,C1,D1), edge(Y,Z,C2,D2), C is C1+C2, D is D1+D2.
path(X,Z,C,D) :- edge(X,Z,C,D).

edge(a,b,1,4).
edge(b,a,1,3).

% preference clauses
path(X,Z,C1,D1) < path(X,Z,C2,D2) :- C2<C1, !.
path(X,Z,C1,D1) < path(X,Z,C2,D2) :- C2=C1, D2<D1.
```

Figure 8 Example of a program using Preferences.

The program finds the shortest path between two nodes in a graph and, if there are two or more paths with the same cost, the tie is undone by selecting the answer with the less distance. The program is divided in two parts: the problem specification (the top part in Fig. 8) and the preference clauses that are responsible for selecting the optimal answers for the problem (the bottom part in Fig. 8). The symbol $<$, used in the preference clauses, can be read as *"is less preferred than"*.

To efficiently implement Preferences, Guo et al. [5, 6] proposed the use of mode-directed tabling. For that, they introduced slightly modifications to Govindarajan's original syntax together with a *program transformation* that takes advantage of mode-directed tabling. For example, if we consider the program in Fig. 8, we only need to declare the predicate arguments subjected to preferences which, in this case, corresponds to include the declaration *path(index,index,<,<)* at the top of the program, meaning that the first two arguments are indexed and the last two are subjected to preferences. Starting from this declaration, we next describe Guo's program transformation:

- The first $<$ argument in the declaration of the tabled predicate is replaced by the mode operator *last*. The following $<$ arguments, if any, are replaced by the mode operator *first*.
- The head of the tabled predicate is modified by adding the word *New* to the name of the predicate.
- A new clause, with the same name as the original tabled predicate, is added to the program in order to control the execution of the program and the correct generation of the preferred answers.

For our example, this corresponds to the code in Fig. 9.

However, Guo's approach has one major limitation: all indexed arguments should be instantiated when calling a transformed tabled predicate. The problem resides in the new clause introduced by the third transformation item described above. For example, if we call the open query *path(X,Y,C,D)*, the new *path/4* clause will then call *pathNew(X,Y,C,D)*, in order to generate answers, after what variables X and Y will be instantiated with an answer

```
:- table path(index,index,last,first).

path(X,Z,C,D):-
   pathNew(X,Z,C,D),                   % generate answers
   (path(X,Z,C1,D1) ->                 % if old answers ...
      path(X,Z,C1,D1) < path(X,Z,C,D)  % ... test if the new answer is preferable
   ;
      true                             % otherwise accept the new answer
   ).

pathNew(X,Z,C,D) :- path(X,Y,C1,D1), edge(Y,Z,C2,D2), C is C1+C2, D is D1+D2.
pathNew(X,Z,C,D) :- edge(X,Z,C,D).
```

Figure 9 Preferences with mode-directed tabling.

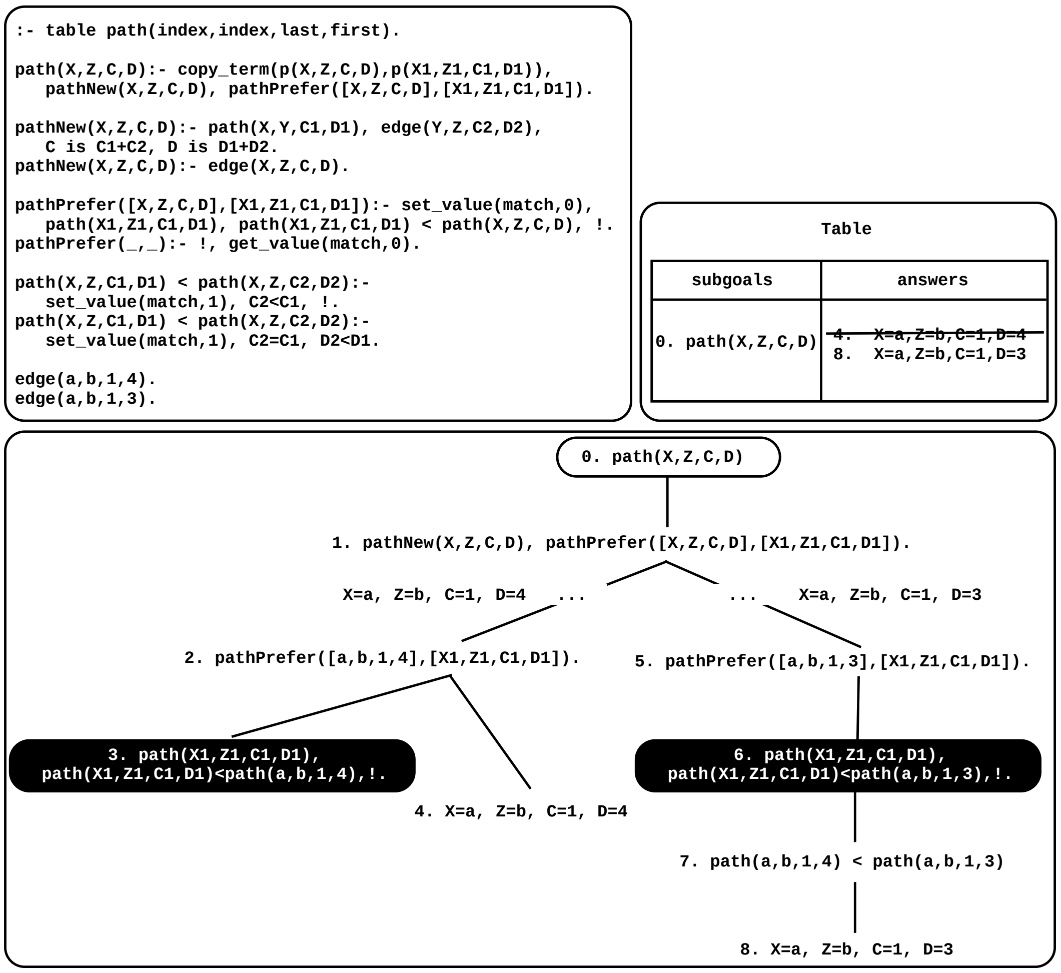
(for instance $\{X{=}x,\ Y{=}y\}$). Next, when calling *path(X,Y,C1,D1)* (now *path(x,y,C1,D1)*), to get old answers from the table and check if the new answer is preferable, the problem arises since the first call was *path(X,Y,C,D)* and the new call is *path(x,y,C,D)*. Remember that tabling only consumes answers from the table space when it detects similar calls, which is not the case here. To solve this problem, we next propose some modifications to Guo's program transformation in order to allow calling open queries. Figure 10 shows our proposal along with the execution tree and table space for the same example and query goal *path(X,Y,C,D)*.

Initially, the original variables are first copied, using the built-in predicate *copy_term/2*, and only then we call the *pathNew/4* predicate (step 1). This call then generates a first intermediate answer (for illustration purposes, we are simplifying some steps here) for the path between *a* and *b* with cost 1 and distance 4 ($\{X{=}a,\ Z{=}b,\ C{=}1,\ D{=}4\}$). After that, we call the new *pathPrefer/2* predicate, introduced by our transformation, which is used to control the correct generation of the preferred answers. The *pathPrefer/2* predicate receives two arguments, one with the new path found and another with the copy of the variables in the original call (step 2).

In the first clause of *pathPrefer/2*, the global variable *match* is set to 0 (we will explain the usefulness of this step next) and then we call *path/4* with the copied variables of the original call (step 3). By doing that, we guarantee that a similar call will be done, hence avoiding Guo's problem. Tabling will thus access the table space and get one by one the available answers. In this case, there are no answers in table, so execution suspends and the second clause for *pathPrefer/2* is tried.

The second clause starts by executing a cut operation (!). This action is done in order to prevent the computation to be later resumed to the first *pathPrefer/2* clause (step 3). Next, it verifies if the global variable *match* is set to 0. We should note that the variable *match* is only set to 1 in the preference clauses, which means that there is at least one matching (but not necessarily preferable) answer in the table space. More, the cut operation performed at the end of the first *pathPrefer/2* clause ensures that, if we are considering the second *pathPrefer/2* clause, it is because the first clause did not succeed in finding a preferable answer. Thus, if the global variable *match* is set to 0 in the second clause, it is because there are no matching answers for the path we are considering, so the new answer can be safely inserted in the table (step 4). If the variable *match* was set to 1, that would mean that, at least, one matching answer existed in the table, and thus the current new answer must be discarded.

In the continuation, we backtrack to node 1 and a second answer is found for the path between *a* and *b* with cost 1 and distance 3 ($\{X{=}a,\ Z{=}b,\ C{=}1,\ D{=}3\}$). Again, the first



```
:- table path(index,index,last,first).

path(X,Z,C,D):- copy_term(p(X,Z,C,D),p(X1,Z1,C1,D1)),
   pathNew(X,Z,C,D), pathPrefer([X,Z,C,D],[X1,Z1,C1,D1]).

pathNew(X,Z,C,D):- path(X,Y,C1,D1), edge(Y,Z,C2,D2),
   C is C1+C2, D is D1+D2.
pathNew(X,Z,C,D):- edge(X,Z,C,D).

pathPrefer([X,Z,C,D],[X1,Z1,C1,D1]):- set_value(match,0),
   path(X1,Z1,C1,D1), path(X1,Z1,C1,D1) < path(X,Z,C,D), !.
pathPrefer(_,_):- !, get_value(match,0).

path(X,Z,C1,D1) < path(X,Z,C2,D2):-
   set_value(match,1), C2<C1, !.
path(X,Z,C1,D1) < path(X,Z,C2,D2):-
   set_value(match,1), C2=C1, D2<D1.

edge(a,b,1,4).
edge(a,b,1,3).
```

Table

subgoals	answers
0. path(X,Z,C,D)	~~4. X=a,Z=b,C=1,D=4~~ 8. X=a,Z=b,C=1,D=3

0. path(X,Z,C,D)
1. pathNew(X,Z,C,D), pathPrefer([X,Z,C,D],[X1,Z1,C1,D1]).
X=a, Z=b, C=1, D=4 ...
... X=a, Z=b, C=1, D=3
2. pathPrefer([a,b,1,4],[X1,Z1,C1,D1]).
5. pathPrefer([a,b,1,3],[X1,Z1,C1,D1]).
3. path(X1,Z1,C1,D1), path(X1,Z1,C1,D1)<path(a,b,1,4),!.
4. X=a, Z=b, C=1, D=4
6. path(X1,Z1,C1,D1), path(X1,Z1,C1,D1)<path(a,b,1,3),!.
7. path(a,b,1,4) < path(a,b,1,3)
8. X=a, Z=b, C=1, D=3

Figure 10 Using our transformation for Preferences with mode-directed tabling.

clause for *pathPrefer/2* is called (step 5) but now, there is an answer in the table space, the one found at step 4. Using the preference clauses (step 7), it is concluded that the new answer is preferable than the previous one. The predicate thus succeeds and the new answer is inserted in the table space replacing the older one (step 8).

6 Mode-Directed Tabling and Answer Subsumption

Traditional tabling only inserts an answer in table space if it is not a variant of one already there. In this paper, we presented mode operators that allow changing this behavior, giving us more control over the process of answer insertion. In particular, in the previous section, we discussed how to use mode-directed tabling for defining our own criteria of optimization through preferences. Preferences could be seen as a sub-case of Answer Subsumption, a technique that we will now introduce.

Answer subsumption allows a newly found answer to modify the set of answers present in the table. Plus, it can generate a third answer based on such set, instead of choosing among the old and new answers, like we ought to do with Preferences. This possibility allows us to construct, for example, counters and aggregates.

XSB Prolog has two answer subsumption mechanisms [13]. The first one is called *partial order answer subsumption* and is comparable, in terms of functionality, with Preferences. Consider that we want to use it with the program that finds the shortest path in a graph:

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

then, we should declare the *path/3* predicate as:

```
:- table path(_,_,po(</2))
```

meaning that the third argument, related to the cost of the path, will be evaluated using partial order answer subsumption, where the preference predicate *</2* implements the partial order relation. The other arguments are considered to be index arguments. The preference predicate is then used to decide when a new answer is preferable to an answer already stored in the table space.

The second XSB's mechanism is more powerful and is called *lattice answer subsumption*. With this mechanism, we can write a preference predicate that can generate a third answer starting from the new answer and from the answer stored in the table. To use it with the last example, we only need to change the declaration of the *path/3* predicate to:

```
:- table path(_,_,lattice(min/3))
```

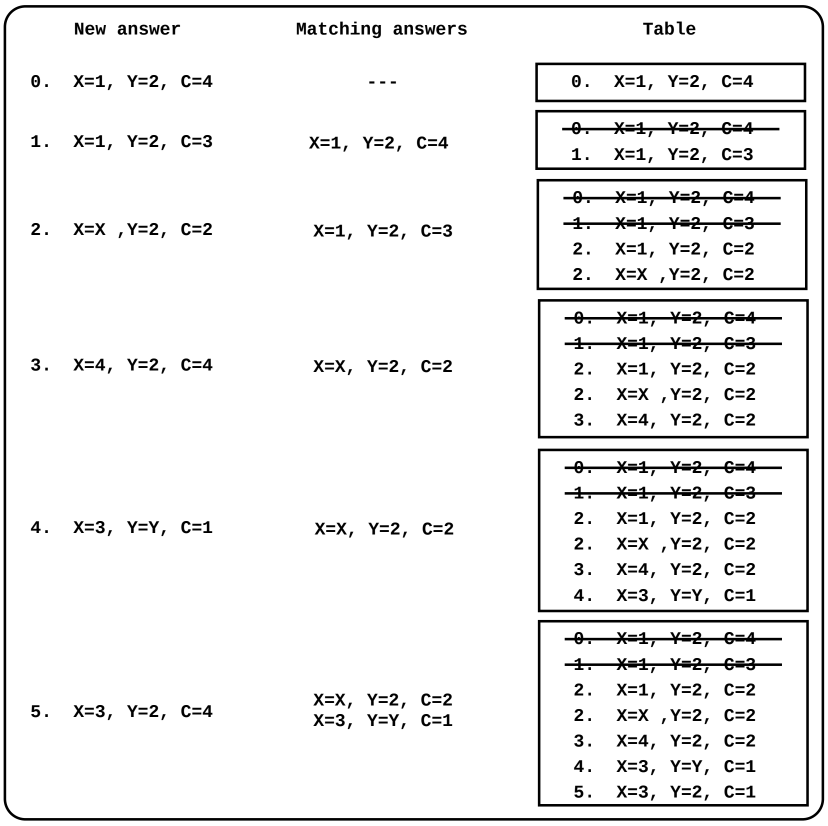
Note that the *min/3* predicate must have three arguments. This is necessary since this mechanism can generate a third answer from two others. For example, for the shortest path problem, the preference predicate *min/3* could be something like:

```
min(Old,New,Result) :- Old<New -> Result=Old ; Result=New.
```

Consider now that we run the open query goal *path(A,B,C)* and that we obtain the answers {*A=a, B=b, C=6*} and {*A=a, B=c, C=7*}. Next, if we find a third answer {*A=a, B=Y, C=1*}, it would make sense to also generate the answers {*A=a, B=b, C=1*} and {*A=a, B=c, C=1*} to replace the previous ones. Instead, with XSB, we get the answers {*A=a, B=b, C=6*} and {*A=a, B=c, C=1*}, meaning that XSB is only able to use the new answer found, {*A=a, B=Y, C=1*}, with just one matching answer present in the table.

Before describing our proposal for answer subsumption, that solves XSB's limitation, we present its behavior. For that, we consider the same *min/3* example and we present, in Fig. 11, an example showing how a sequence of new answers being found alters the table space. The left column shows the new answer being found and the right column shows the table state after considering that answer. In the middle column, we show the list of matching answers for the new answer at hand.

At step 0, we have the simplest case: the table space is empty, thus the new answer {*X=1, Y=2, C=4*} is inserted. Next, at step 1, we have the answer {*X=1, Y=2, C=3*}. This answer is a variant of the previous one and, since it is a preferable answer (3<4), it is inserted in the table and the previous one is deleted. At step 2, we have an answer with a free variable in the index arguments, {*X=X, Y=2, C=2*}. This answer matches with the previous answer and, since the new answer is preferable (2<3), the answer in the table space is updated to {*X=1, Y=2, C=2*}. In fact, by updated we mean that the previous answer is deleted and that this new one is inserted. Then, the original answer {*X=X, Y=2, C=2*} is also inserted in the table space, since it is not a variant of any other answer. At step 3, we have the opposite case: the new answer, {*X=4, Y=2, C=4*}, matches with the answer with



New answer	Matching answers	Table
0. X=1, Y=2, C=4	---	0. X=1, Y=2, C=4
1. X=1, Y=2, C=3	X=1, Y=2, C=4	~~0. X=1, Y=2, C=4~~
		1. X=1, Y=2, C=3
2. X=X ,Y=2, C=2	X=1, Y=2, C=3	~~0. X=1, Y=2, C=4~~
		~~1. X=1, Y=2, C=3~~
		2. X=1, Y=2, C=2
		2. X=X ,Y=2, C=2
3. X=4, Y=2, C=4	X=X, Y=2, C=2	~~0. X=1, Y=2, C=4~~
		~~1. X=1, Y=2, C=3~~
		2. X=1, Y=2, C=2
		2. X=X ,Y=2, C=2
		3. X=4, Y=2, C=2
4. X=3, Y=Y, C=1	X=X, Y=2, C=2	~~0. X=1, Y=2, C=4~~
		~~1. X=1, Y=2, C=3~~
		2. X=1, Y=2, C=2
		2. X=X ,Y=2, C=2
		3. X=4, Y=2, C=2
		4. X=3, Y=Y, C=1
5. X=3, Y=2, C=4	X=X, Y=2, C=2	~~0. X=1, Y=2, C=4~~
	X=3, Y=Y, C=1	~~1. X=1, Y=2, C=3~~
		2. X=1, Y=2, C=2
		2. X=X ,Y=2, C=2
		3. X=4, Y=2, C=2
		4. X=3, Y=Y, C=1
		5. X=3, Y=2, C=1

Figure 11 How our proposal for Answer Subsumption changes the table space.

free variables. Since the matching answer is a preferable answer (2<4), then a third answer, $\{X{=}4,\ Y{=}2,\ C{=}2\}$, is created and inserted in the table space.

At step 4, we illustrate a situation where the new answer $\{X{=}3,\ Y{=}Y,\ C{=}1\}$ unifies with a matching answer, $\{X{=}X,\ Y{=}2,\ C{=}2\}$, but they cannot be considered compatible. If we consider them compatible, then the matching answer will be updated to $\{X{=}X,\ Y{=}2,\ C{=}1\}$, which can lead to wrong answers. For example, if the answer $\{X{=}1,\ Y{=}2,\ C{=}4\}$ is then found, it will be turned into $\{X{=}1,\ Y{=}2,\ C{=}1\}$, which will be a wrong answer since the answer $\{X{=}3,\ Y{=}Y,\ C{=}1\}$ does not unifies with it. Our approach prevents those situations and, for such cases, the answer being found is simply inserted in the table space without modifying any previous answer.

The last step presents a situation where we have more than a matching answer. Since both matching answers are preferable to the new answer, $\{X{=}3,\ Y{=}2,\ C{=}4\}$, we must be careful as otherwise we can incorrectly create the answers $\{X{=}3,\ Y{=}2,\ C{=}2\}$ and $\{X{=}3,\ Y{=}2,\ C{=}1\}$. Since the answer $\{X{=}3,\ Y{=}2,\ C{=}1\}$ is preferable (1<2), we must consider only it to be inserted in the table space.

Next, we discuss, in more detail, the implementation of our answer subsumption proposal. We use a mode-directed tabled predicate named *answer_subsumption/3*. The first argument is the template representing the predicate name and the index arguments of the predicate to be answer subsumed. The second argument is the name of the preference predicate (as explained for XSB, this predicate must have three arguments). The third argument is the answer subsumption argument for the template in the first argument. Figure 12 uses again

```
path(X,Z,C) :- answer_subsumption(pathNew(X,Z),min,C).

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

min(Old,New,Result) :- Old<New -> Result=Old ; Result=New.
```

Figure 12 The shortest path example with Answer Subsumption.

the shortest path program to illustrate how the new *answer_subsumption/3* predicate can be used to implement answer subsumption in the YapTab system.

As you may have already noticed, for the *answer_subsumption/3* call in the body of the *path/3* predicate, the *pathNew/3* predicate appears with the answer subsumption argument, variable *C*, moved to the last argument of the *answer_subsumption/3* call. This is necessary since, for variant checking when calling a *answer_subsumption/3* subgoal, we should have the index arguments for *pathNew/3* separate from the non-index arguments. This is also the reason for the *answer_subsumption/3* predicate be declared as *answer_subsumption(index,index,last)* in our implementation. Figure 13 shows, in more detail, the implementation for the *answer_subsumption/3* predicate.

Initially, the third argument variable *Result* is first copied, using the built-in predicate *copy_term/2*, to two auxiliary variables, *CallResult* and *CopyResult*. The goal of this action is to avoid the original variable *Result* to become early bound. Next, using the built-in predicate *call/2*, we call the predicate to be answer subsumed in order to find a new answer. For example, to the shortest path example this corresponds to call *pathNew(X,Y,CallResult)*.

Then, we execute the *answer_subsumption_matching_answers/4* predicate. This predicate is responsible for accessing the table space and returning a list with all answers matching with the index arguments for the predicate at hand.

At the end, we test if the list of matching answers is empty or not. If the list is empty, we simply insert the *CallResult* answer returned by the call to the predicate at hand. Otherwise, if there are matching answers, we execute the *answer_subsumption_check_insert_update/5* predicate. This predicate is responsible for implementing the answer subsumption mechanism, putting into practice the rules described earlier for the example in Fig. 11.

A final and more interesting example of the power of using answer subsumption is the program shown in Figure 14, that takes advantage of the functionality of combining answers to produce a third answer.

```
:- table answer_subsumption(index,index,last).

answer_subsumption(Call,Op,Result) :-
   copy_term(Result,CallResult),
   copy_term(Result,CopyResult),
   call(Call,CallResult),
   answer_subsumption_matching_answers(Call,Op,CopyResult,AnswersList),
   (
      AnswersList \= []
   ->
      answer_subsumption_check_insert_update(Call,Op,CallResult,AnswersList,Result)
   ;
      Result=CallResult
   ).
```

Figure 13 Our implementation of Answer Subsumption with mode-directed tabling.

```
path(X,Z,N) :- answer_subsumption(pathNew(X,Z),sum,N).

pathNew(X,Z,N) :- path(X,Y,N), edge(Y,Z).
pathNew(X,Z,1) :- edge(X,Z).

sum(Old,New,Result) :- Result is Old+New.

edge(1,3).
edge(1,2).
edge(2,3).
```

Figure 14 Using answer subsumption to compute the number of paths between nodes in a graph.

The predicate *sum/3* adds the number of paths for the answer being found with the answer in the table, if any. For example, if we run the query goal *path(1,3,N)*, we will get the answer $\{N{=}2\}$, meaning that there are two different paths between nodes *1* and *3*, the first using the direct path between *1* and *3* and the other using the path through node *2*.

7 Conclusions

Mode-directed tabling is an extension to the tabling technique that supports the definition of selective criteria for specifying how answers are inserted into the table space. In this paper, we have presented a more general approach to the declaration and use of mode-directed tabling, implemented on top of the YapTab tabling system, and we have discussed the use of mode operators for implementing Justification, Preferences and Answer Subsumption in tabled logic programs. In particular, for Preferences and Answer Subsumption, we have presented alternative approaches that solve the limitations present in Guo's and XSB's original proposals, respectively. As further work, we plan to investigate how mode-directed tabling can be applied to other application areas.

Acknowledgements 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 HORUS (PTDC/EIA-EIA/100897/2008) and LEAP (PTDC/EIA-CCO/112158/2009).

References

1 W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs. *Journal of the ACM*, 43(1):20–74, 1996.

2 K. Govindarajan, B. Jayaraman, and S. Mantha. Preference Logic Programming. In *International Conference on Logic Programming*, pages 731–745. MIT Press, 1995.

3 Hai-Feng Guo and G. Gupta. A Simple Scheme for Implementing Tabled Logic Programming Systems Based on Dynamic Reordering of Alternatives. In *International Conference on Logic Programming*, number 2237 in LNCS, pages 181–196. Springer-Verlag, 2001.

4 Hai-Feng Guo and G. Gupta. Simplifying Dynamic Programming via Mode-directed Tabling. *Software Practice and Experience*, 38(1):75–94, 2008.

5 Hai-Feng Guo and B. Jayaraman. Mode-directed Preferences for Logic Programs. In *ACM Symposium on Applied Computing*, pages 1414–1418. ACM, 2005.

6 Hai-Feng Guo, B. Jayaraman, G. Gupta, and M. Liu. Optimization with Mode-Directed Preferences. In *7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 242–251. ACM, 2005.

7 G. Pemmasani, Hai-Feng Guo, Y. Dong, C. R. Ramakrishnan, and I. V. Ramakrishnan. Online Justification for Tabled Logic Programs. In *International Symposium on Functional and Logic Programming*, number 2998 in LNCS, pages 24–38. Springer-Verlag, 2004.

8 F. Riguzzi and T. Swift. Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions. In *International Conference on Logic Programming, Technical Communications*, volume 7 of *LIPIcs*, pages 162–171. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

9 R. Rocha, F. Silva, and V. Santos Costa. On applying or-parallelism and tabling to logic programs. *Theory and Practice of Logic Programming*, 5(1 & 2):161–205, 2005.

10 A. Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying Proofs Using Memo Tables. In *ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 178–189. ACM, 2000.

11 K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(3):586–634, 1998.

12 V. Santos Costa, R. Rocha, and L. Damas. The YAP Prolog System. *Journal of Theory and Practice of Logic Programming*, 12(1 & 2):5–34, 2012.

13 T. Swift and D. S. Warren. Tabling with Answer Subsumption: Implementation, Applications and Performance. In *European Conference on Logics in Artificial Intelligence*, number 6341 in LNAI, pages 300–312. Springer-Verlag, 2010.

14 T. Swift and D. S. Warren. XSB: Extending Prolog with Tabled Logic Programming. *Theory and Practice of Logic Programming*, 12(1 & 2):157–187, 2012.

15 Neng-Fa Zhou, Y. Kameya, and T. Sato. Mode-Directed Tabling for Dynamic Programming, Machine Learning, and Constraint Solving. In *IEEE International Conference on Tools with Artificial Intelligence*, volume 2, pages 213–218. IEEE Computer Society, 2010.

16 Neng-Fa Zhou, Yi-Dong Shen, Li-Yan Yuan, and Jia-Huai You. Implementation of a Linear Tabling Mechanism. In *Practical Aspects of Declarative Languages*, number 1753 in LNCS, pages 109–123. Springer-Verlag, 2000.