Tuple Tabulation

Paulo J Azevedo
Departamento de Informatica
Universidade do Minho
Campus de Gualtar,
4710 Braga, Portugal
pja@di.uminho.pt

Abstract

This paper investigates a tuple-oriented approach to tabulation. In this approach solutions to a call are stored and computed one at a time, which results in tables contain incomplete sets of solutions. Since the answers stored in the tables are partial sets of solutions to a call, a subgoal that is an instance of a stored call can require an answer that is not among the stored solutions. Consequently recomputation can arise. This recomputation problem is overcome by introducing a program transformation. Evaluation in the transformed program is free of recomputation because it is diverted from previously visited search space, corresponding to previously computed and stored solutions in the tables. In this paper we also show that negation and tabulation can be efficiently implemented under this approach. The advantages and the problems that emerge with this proposal are discussed.

1 Introduction

One source of inefficiency in the evaluation of a program is that the same computation may be repeated many times over. This phenomenon is particularly evident if the program contains recursive definitions. The described phenomenon is often referred as *recomputation*. Tabulation is a technique that overcomes the inefficiency associated with the described recomputation. It does so by a process in which a computation is performed only once and then its results are stored in some conveniently represented table for future use. Subsequent requests for the same computation are answered by looking up for the appropriate table entry.

Within logic programming, eliminating recomputation can be crucial not just for the purpose of efficiency but also for achieving completeness. For this reason Tabulation is widthly used in areas such as Deductive Databases e.g. [Vieille89] and Abstract Interpretation e.g. [Warren92]. Procedurally, logic programming languages are nondeterministic. Consequently, a call to a procedure can have multiple answers. For this reason tabulation has been approached in a set-oriented way i.e. a subgoal to a tabulated predicate is evaluated completely: all answers are computed and stored in the tables. Thus, further subgoals that are instances of the previous evaluated one can be answered by retrieving the required solutions from the tables. Tabulated top-down proof-procedures such as OLDT [Tamaki&Sato86] and XSB [Sagonas et al. 94] implement tabulation in this form.

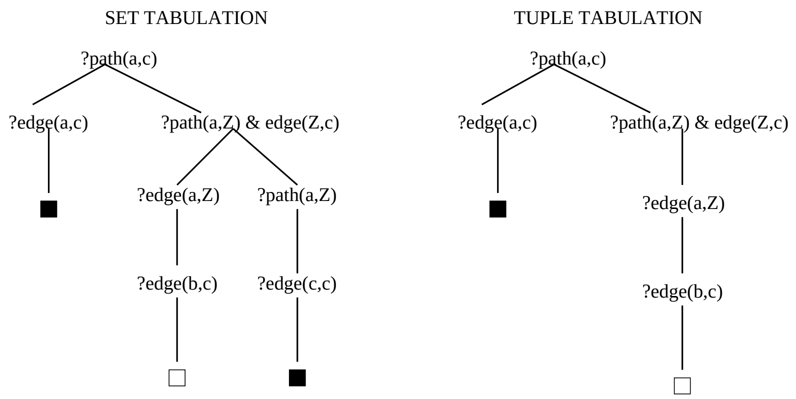

Figure 1: Set versus Tuple Tabulation

However, set-oriented tabulation is not always efficient. Consider the application of tabulation to Prolog as in XSB [Sagonas et al. 94]. It is well known the advantages of tuple-oriented evaluation of Prolog in a programming language perspective. Evaluating a subgoal to a tabulated predicate in a set-oriented way may arise irrelevant work since unnecessary solutions can be computed and stored in the tables. In a more drastic case, a call to a program can have an infinite number of solutions (e.g. a program to generate natural numbers) but one might be only interested in a specific answer. One interesting example emerges when, in general, we are only interested in obtaining a single solution to a call. Consider the example of ground calls (i.e. all parameters with constants) to the left-recursive path finder program. For the graph:

$\mathsf{edge}(\mathsf{a}, \mathsf{b}).$
$\mathsf{edge}(\mathsf{b}, \mathsf{c}).$

and the query $?\mathsf{path}(\mathsf{a}, \mathsf{c})$ the difference between set-oriented and tuple-oriented evaluation is expressed in figure 1. In set-oriented evaluation all solutions are computed for the subgoal $?\mathsf{path}(\mathsf{a}, \mathsf{Z})$ which leads to some irrelevant computation (notice the failure branch). It is irrelevant because only one solution to this subgoal is necessary to answer the top query. This is precisely what happens when tuple-oriented evaluation is used. Thus, applying tabulation to this example using a set oriented approach would also imply storing unnecessary solutions.

The motivation for tuple-oriented evaluation when only a single solution to a call is required was also identified by the deductive databases community in bottom-up evaluation. In [Ramakrishnan et al. 92] it is shown that for cases where only a single answer to a query is expected, the set-orientness of bottom-up evaluation performs unnecessary work. The authors propose a new technique to order the use of generated subgoals (magic facts). In order to obtain an hybrid evaluation between pure breadth-first and pure depth-first tuple-oriented evaluation, the technique controls the accessibility to the derived facts during the bottom-up evaluation.

Another important example where the inefficiency of set-oriented tabulation is clearly illustrated is the case of negation. Evaluating a negative call to a tabulated predicate can perform unnecessary work. By definition, evaluating a negative call terminates when a success branch of the search space is found or when all the branches are visited. Consider the case where the positive call generated by the negative subgoal finds one solution (one success branch), meaning that the subgoal fails. Computing all solutions for this positive call is inefficient because only one answer is necessary to answer the negative subgoal. A good example of this situation emerges with the use of 'existential negation' i.e. calls of the form $?not\ \exists X p(X)$. A natural way to evaluate calls of this form is to compute them tuple-oriented until a solution is found or all the branches are visited.

This problem of efficiently computing negative calls to tabulated predicates was also described in [Sagonas et al. 94]. These authors observed that computing all solutions for a negative call to a tabulated predicate can be inefficient. They discussed the case of existential negated calls and the most efficient way to evaluate these calls, mainly the case where a solution is found to the positive goal generated by the negated call. For this purpose XSB provides two types of negation: *SLG negation* where calls to tabulated predicates are completely evaluated and *existential negation* where tables are deleted after the first answer to the positive call generated by a negative subgoal is obtained. This deletion occurs to avoid storing partial sets of solutions to a call in the tables. Working with partial sets of solutions can be problematic because a call can require a specific answer that is not among the ones stored in the tables. Furthermore, computing this specific solution might lead to recompute the partial set of solutions stored in the tables since the proof-procedure does not distinguish between already computed and not yet computed answers.

Due to the deletion of tables, the approach given by the XSB authors does not enable the sharing of answers between existential negated calls and positive calls to tabulated predicates. One can overcome this failure (occurring in tabulation systems as XSB) in providing reuse of work by negated and positive calls, without recomputation by making use of the concepts introduced in [Azevedo&Sergot94]. Consider the following program which contains an existential negated literal (a pattern that frequently arises in practice).

q ← not ∃X p(X).
q ← p(Y) & s(Y).
p(a) ← q(a)
p(X) ← r(X)
s(c) ←
q(c) ←
q(d) ←
r(b) ←
r(c) ←
r(a) ←

Suppose that predicate p is tabulated and Prolog-like evaluation is applied. Assume also that evaluation is always tuple-oriented and that the query is ?q. One can see that in this example, the negative call to p in the first clause fails, because the first solution is computed and stored, i.e. the solution p(b). According to XSB and its existential negation the table for p would be deleted. However, one could preserve the table for ?p(X) with solution [p(b)] since another call to p could make use of this result. In our case, evaluation for the positive call must compute the solution p(c) since the only solution for a call to s is s(c). Thus, recomputation occurs because p(b) is computed twice; one for the negative call and the other for the positive call. Figure 2 illustrates this phenomenon through the visited branches in the SLD-tree.

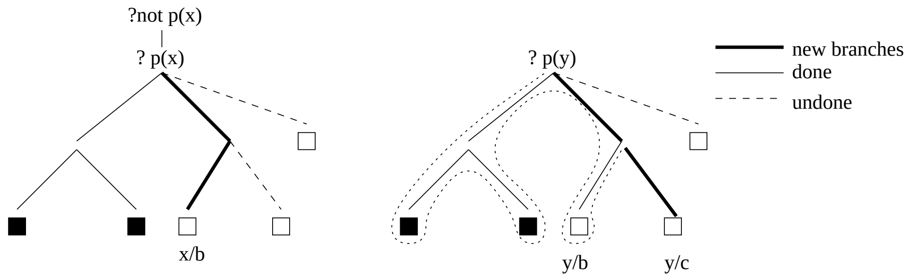

Figure 2: Example with negation

In this paper we propose a technique that enables the sharing of answers without recomputation, as it happens in the case of existential negation in XSB. We assume tuple-oriented evaluation for all subgoals that emerge during the computation of a query in a program. These include also subgoals to tabulated predicates. Thus, we consider *Tuple Tabulation* in contrast to Set Tabulation as implemented in systems like XSB [Sagonas et al. 94] or OLDT [Tamaki&Sato86]. Evaluation of a subgoal to a tabulated predicate in a program is diverted from the search space visited to computed the answers already stored in the tables. In this way the recomputation phenomenon described before is avoided. The diversion effect is obtained through a program transformation. A representation of the branch that lead to the last computed and stored solution is used to enable a recomputation-free evaluation in the transformed program.

The rest of the paper is organized as follows: Section 2 presents tuple tabulation including the program transformation and the representation of branches from a SLD-tree. Section 3 shows some measurements of performance with the negation example. Finally, section 4 concludes the work.

2 Tuple-Oriented Tabulation

In some applications, computing all solutions to a call performs unnecessary work, since only a specific solution is required for answering the call. The actual alternative approach when applying tabulation is to evaluate all the calls tuple-oriented and work with tables which contain partial sets of solutions. We can define what is meant to be *Tuple Tabulation* in the following way:

In Tuple Tabulation a subgoal that is an instance of call stored in the tables is answered by first retrieving solutions from the table. Since computation occurs "tuple at a time", the set of solutions stored in the table for a call is an incomplete set. Thus, if no solution is found in the table then the subgoal must be answered from a computation in the program.

Although tuple tabulation is desirable, there are reasons that justify the fact that it has not been adopted generally. Consider our motivative example with negation and figure 2. Suppose evaluation is tuple-oriented with a left-to-right computation rule and the call ?q is performed. With the call ?p(X) (generated from ?not p(X)) a solution is found and stored in the tables, and a subset of the search space is explored. Since evaluation is a depth-first search strategy with a left-to-right computation rule, this explored subset is located on the left-hand side of the search tree (branches in bold in figure 2). The call ?p(Y) requires solutions that are not in the table, namely p(c) since the only solution to a call to s is $\mathsf{s(c)}$. However to compute this solution requires to recompute the left-hand side of the search tree as illustrated in figure 2. Thus, one of the reasons for not implementing tabulation in tuple-oriented way is that in this approach tables contain partial sets of solutions and consequently recomputation arises. That is, evaluation recomputes because it is not aware of what was previously done in the program.

Let us assume tuple-oriented evaluation, as in Prolog. As we have seen, one particular scenario emerges in terms of search space consumption. Since evaluation is depth first left-to-right, the consumed search space is always located on the left of the last traversed branch. The situation is described in figure 3. Recomputation arises because evaluation begins from the left-most branch of the search tree (following the standard Prolog computation rule) when it should begin from the last traversed branch, which corresponds to the last computed and stored solution. In this sense, one can use the last traversed branch to divert the computation from the earlier visited search space.

A branch can be efficiently represented through *paths*. A path is an ordered list of input clauses identifiers that represents a specific branch of the search tree, going from the top node (query) to a leaf (solution). For instance, the path $[(p, 2), (r, 1)]$ associated with the solution $\mathsf{p(b)}$ represents the branch build out of a computation that uses the second clause on the definition of predicate p and the first clause on the definition of r.

To solve the described recomputation problem one has to store the last traversed branch (last computed path) along with the computed answers. To obtain, together with an answer, the associated path (corresponding to the traversed branch) the rules of the program are transformed in the following way:
Each original clause

$$p(\vec{x}) \leftarrow a_1(\vec{y}_1) \ \&...\& \ a_m(\vec{y}_m)$$

which is clause n of predicate p, is rewritten into:

transf I) $p(\vec{x}, [(p, n) \mid Pa_1] - Pa_{m+1}) \leftarrow$
$$a_1(\vec{y}_1, Pa_1 - Pa_2) \ \& \ a_2(\vec{y}_2, Pa_2 - Pa_3) \ \&...\& \ a_m(\vec{y}_m, Pa_m - Pa_{m+1})$$

where Pa_i are distinct new variables that will be bound by the path followed. The constant p is the predicate name and n is the clause number. The last parameter added to the head of the predicate p is a difference-list that represents the path traversed for each predicate in the body. This transformation was first introduced in [Azevedo&Sergot94]. Evaluating the query $?\mathsf{p}(\mathsf{X}, \mathsf{P} - [])$ in our example, after applying transformation **transf I)**, receives as answer $\mathsf{X} = \mathsf{b}$ and $\mathsf{P} = [(\mathsf{p}, 2), (\mathsf{r}, 1)]$.

In this way, whenever a new solution is computed and stored in the table, the traversed path is also stored to represent the last visited branch. A "representation" of the consumed search space is then obtained through the last computed path. To avoid the recomputation associated with tuple-oriented tabulation it is required that evaluation considers not the left-most branch but the last visited branch represented by the last stored path. We obtain this behaviour through a program transformation. The computation in the transformed program is diverted from the earlier visited branches using the last stored path. This path is included in the query to the tabulated predicate. Under this new transformation evaluation is driven to follow the last path top-down until the last described clause. Subsequently and via backtracking, evaluation starts searching for the next unvisited branches to the right of the last path.

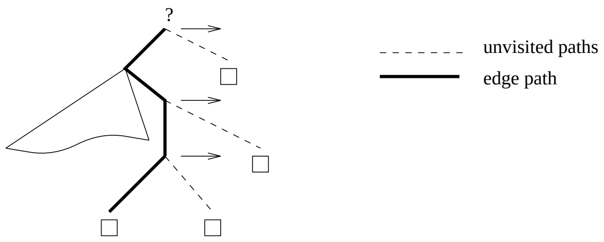

Figure 3: Prolog's evaluation strategy and search space consumption

The referred transformation, together with the last path embedded in the query, implements the mechanism of diversion. The transformation generates several clauses that deal with each of the situations where, under the diversion mechanism, a clause of the original program would be used. Three clauses are necessary to cover all possible cases that emerge during computation. The first clause expresses the case where a new unvisited (branch) path is already followed. Hence the last path is assigned as empty, meaning that the diversion mechanism is switched 'off'. The second clause represents the case where the last path is followed. However the clause can only be used if it is not the last one described in the last path. Finally, the third clause expresses the situation where diversion from the given last path occurs. That is, the next branch to the right of the given last path is followed. Hence, for rule number n of a predicate p:

$$
\begin{array}{l}
p(\vec{x}) \leftarrow \\
\qquad a_1(\vec{y}_1) \ \& \ \ldots \ \& \ a_m(\vec{y}_m).
\end{array}
$$

it is transformed into three new mutually exclusive clauses, namely:

P1) $p(\vec{x}, [(p,n) \mid Path] - R, [\,], [\,]) \leftarrow$
$\quad a_1(\vec{y}_1, Path - R_1, [\,], [\,]) \ \& \ \ldots \ \&$
$\quad a_m(\vec{y}_m, R_{m-1} - R, [\,], [\,]).$

P2) $p(\vec{x}, [(p,n) \mid Path] - R, [(p,N) \mid Rest], Out) \leftarrow$
$\quad n = N \quad \&$
$\quad Rest \neq [\,] \quad \&$
$\quad a_1(\vec{y}_1, Path - R_1, Rest, Out_1) \ \& \ \ldots \ \&$
$\quad a_m(\vec{y}_m, R_{m-1} - R, Out_{m-1}, Out).$

P3) $p(\vec{x}, [(p,n) \mid Path] - R, [(p,N) \mid \_], [\,]) \leftarrow$
$\quad N < n \quad \&$
$\quad a_1(\vec{y}_1, Path - R_1, [\,], [\,]) \ \& \ \ldots \ \&$
$\quad a_m(\vec{y}_m, R_{m-1} - R, [\,], [\,]).$

In clause **P2)**, explicit equalities and inequalities were used for the sake of clarity. Three new variables are added to the original program. The first ($Path$) is the path performed during a computation of an answer, as it was previously introduced in **transf I)**. The second represents the path (last path) used to divert the computation. The variable Out

carries the residue of the last path to be used on the next clause. The transformed program follows the given path until the last clause identifier. At the last clause identifier, clause type **P2)** cannot be used. Hence evaluation proceeds by trying the remained clauses of type **P3)**. This corresponds to starting the diversion by trying the next clauses 'on the right' of the given last path. If one is found computation continues with the last path as [ ] i.e. using only clauses of type **P1)**.

One should notice that to avoid the generation of redundant clauses (and associated computation), transformed clauses of type **P3)** are not applied to the first clause on the definition of each predicate. Redundancy is based on the fact that constraint $N < 1$ can never be satisfied i.e. no path has a clause whose number is less than 1. Taking in consideration this transformation, for each query to a tabulated predicate:

$$?p(\vec{x})$$

is transformed into

$$?p(\vec{x}, P - [\,], [\,], [\,])$$

if it is admissible (in the sense of [Vieille89]) and into

$$?p(\vec{x}, P - [\,], Edge, \_)$$

if the diversion mechanism is to be used, where the variable $Edge$ is the last path that was previously stored. Computation in the transformed program is always performed by the stored call. This ensures a contiguous consumption of search space and completeness of the tabulation system. Consider the case where the call to be answered is ?p(a) which is subsumed by the stored call ?p(X). If one computes in the transformed program answers for the call ?p(a) instead of the call ?p(X) then some solutions to the stored call can be skipped and consequently incompleteness arise. The described format of calls is equivalent to adding the following new rules to the transformed program, connecting in this way the queries to the tabulated predicate with the transformed program.

I) $p(\vec{x}) \leftarrow$
$$tables(p(\vec{x})).$$

II) $p(\vec{x}) \leftarrow$
$$\begin{array}{l} last\_path(Path, p(\vec{w})) \rightarrow \\ (\ copy\_term(\vec{w}, \vec{w_c})\ \ \& \\ p(\vec{w}, P - [\,], Path, \_)\ \ \& \\ retract(last\_path(Path, p(\vec{w_c})))\ \ \& \\ assert(last\_path(P, p(\vec{w_c})))\ \& \\ assert(tables(p(\vec{w}))\ \& \\ \vec{x} = \vec{w}.) \\ \vee \\ (\ copy\_term(\vec{x}, \vec{x_c})\ \ \& \\ p(\vec{x}, P - [\,], [\,], [\,])\ \ \& \\ assert(tables(p(\vec{x}))\ \& \\ assert(last\_path(P, p(\vec{x_c})))\ \ ). \end{array}$$

The first clause represents the case where there are solutions in the tables for the query. The second expresses two cases, having in both of them the storing of new answers. One where it is necessary to compute solutions from the transformed program. Thus, the diversion mechanism is applied. Notice that in this case computation occurs for the stored call that gave rise to the last path i.e. $?p(\vec{w})$. Evaluating the program always with the same call preserves the contiguous consumption of search space and consequently one does not skip solutions in relation to the stored call. Consider the case where ?p(X) is the call in the last path and ?p(a) the call to be answered. Evaluating the ?p(a) in the program can skip solutions to ?p(X). The last step on this case is to retrieve the computed solutions to the call to be answered i.e to $?p(\vec{x})$. The second case covers the situation where answers are computed for the first time. Let us assume the following program P where p is a tabulated predicate:

```
q ← not ∃X p(X).
q ← p(X) & s(X).
s(c) ←
s(h) ←

p(a) ← s(a).
p(X) ← t(X).
t(b) ←
t(c) ←
t(a) ←
```

Applying the transformation from **P1)** to **P3)** to this example P with the redundant information removed yields:

```
q ← not ∃X p(X).
q ← p(X) & s(X).

s(c) ←
s(h) ←

p(X) ←
    tables(p(X)).
p(X) ←
    last_path(Path, p(W)) →
    ( copy_term(W, CW) &
    p(W, P − [ ], Path, _) &
    assert(tables(p(W)) &
    retract(last_path(Path, p(CW))) &
    assert(last_path(P, p(W))) &
    X = W)
    ∨
    ( copy_term(X, CX) &
    (p(X, P − [ ], [ ], [ ]) &
    assert(tables(p(X)) &
    assert(last_path(P, p(CX))) ).
```

$\mathsf{p(a, [(p, 1) \mid P1] - Pn, [\,], [\,]) \leftarrow}$
$\quad \mathsf{s(a, P1 - Pn, [\,], [\,]).}$
$\mathsf{p(a, [(p, 1) \mid P1] - Pn, [(p, 1), V \mid Rest], Out) \leftarrow}$
$\quad \mathsf{s(a, P1 - Pn, [V \mid Rest], Out).}$
$\mathsf{p(X, [(p, 2) \mid P1] - Pn, [\,], [\,]) \leftarrow}$
$\quad \mathsf{t(X, P1 - Pn, [\,], [\,]).}$
$\mathsf{p(X, [(p, 2) \mid P1] - Pn, [(p, 2), V \mid Rest], Out) \leftarrow}$
$\quad \mathsf{t(X, P1 - Pn, [V \mid Rest], Out).}$
$\mathsf{p(X, [(p, 2) \mid P1] - Pn, [(p, N) \mid Rest], [\,]) \leftarrow}$
$\quad \mathsf{N < 2 \,\&\, t(X, P1 - Pn, [\,], [\,]).}$

$\mathsf{t(b, [(t, 1) \mid Pm] - Pm, [\,], [\,]) \leftarrow}$
$\mathsf{t(b, [(t, 1) \mid Pm] - Pm, [(t, 1), V \mid Rest], [V \mid Rest]) \leftarrow}$
$\mathsf{t(c, [(t, 2) \mid Pm] - Pm, [\,], [\,]) \leftarrow}$
$\mathsf{t(c, [(t, 2) \mid Pm] - Pm, [(t, 2), V \mid Rest], [V \mid Rest]) \leftarrow}$
$\mathsf{t(c, [(t, 2) \mid Pm] - Pm, [(t, N) \mid \_], [\,]) \leftarrow}$
$\quad \mathsf{N < 2.}$
$\mathsf{t(a, [(t, 3) \mid Pm] - Pm, [\,], [\,]) \leftarrow}$
$\mathsf{t(a, [(t, 3) \mid Pm] - Pm, [(t, 3), V \mid Rest], [V \mid Rest]) \leftarrow}$
$\mathsf{t(a, [(t, 3) \mid Pm] - Pm, [(t, N) \mid \_], [\,]) \leftarrow}$
$\quad \mathsf{N < 3.}$

In the next section we present measurements of performance on variants of example P.

3 Measurements of Performance

We perform measurements in XSB Prolog [Sagonas et al. 94] using three version of the same program. The first using XSB existential negation, the second XSB standard form of evaluating negation and tabulation, and finally the third uses tuple tabulation where the original programs are transformed according to **P1)** to **P3)**. Figure 4 illustrates the behaviour of the three programs through the search space consumption that occurs during the evaluation in variants of program P.

In table 1 the results express the measurement of five different versions of program P. The different versions of P vary on the number of irrelevant answers to the query ?$\mathsf{p(X)}$. That is, the number of branches in figure 4 that appear on the right of the branch which leads to the solution $\mathsf{p(c)}$. Example **1** is a variant of program P with two extra solutions, example **2** is P with five more irrelevant solutions, **3** with 10 more solutions, **4** with 13 more and finally **5** with 15 additional solutions. Notice that although tuple tabulation preserves the same performance in all versions of example P (row 3 in table 1), in the other two approaches the irrelevant computation is illustrated in the performance. Thus, the performance in both XSB approaches to negation degenerates with the number of irrelevant solutions. This is explained by the fact that all the extra solutions add along the five different examples to query ?$\mathsf{p(X)}$ are irrelevant to answer the top query ?**not** $\mathsf{p(X)}$.

The results in table 1 represent averages from a population of 1000 queries ?q perform in both programs. XSB Prolog version 1.4 as used in a Sun 3/60 workstation. Time is expressed in milliseconds.

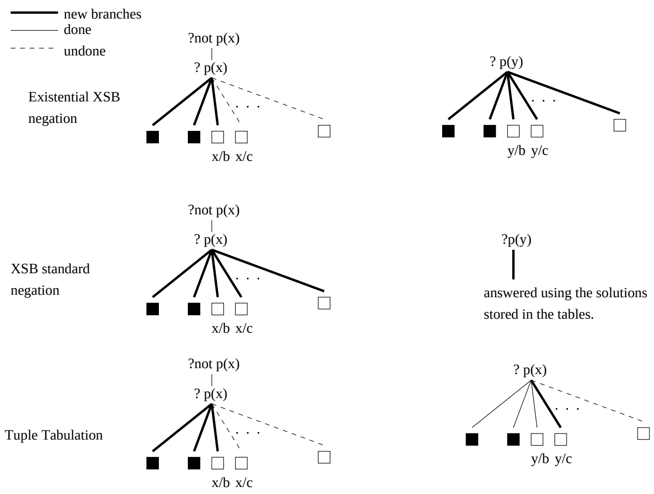

Figure 4: Comparing the three approach to evaluating negation and tabulation.

Version	*Variants*				
	1	**2**	**3**	**4**	**5**
XSB Existential	3.507	4.201	4.860	5.037	5.470
XSB Negation	3.300	3.990	4.670	4.990	5.272
Tuple Tabulation	2.552	2.552	2.552	2.552	2.552

Table 1: Measurements for the three approaches to negation and tabulation (time in ms).

4 Conclusions

In this paper a tuple-oriented approach to tabulation was introduced. One consequence of evaluating calls to tabulated predicates in a tuple-oriented way is that tables contain incomplete sets of solutions. This approach to tabulation is desirable in applications where calls require specific solutions and where computing all solutions to a call performs unnecessary work. Obviously this form of implementing tabulation leads to recomputation because, as tables store incomplete sets of solutions, a call can require a solution that is not in the tables. A diversion mechanism was adapted to overcome this recomputation and a new transformation was proposed. Negation seems to be a good example in practice where it is desirable to have tuple tabulation since a negative call might be answered with only a short number of computations. Thus, irrelevant work is avoided.

Acknowledgements: Paulo Azevedo is supported by Junta Nacional de Investigação Científica e Tecnológica - Programa PRAXIS XXI grant number BD/2832/94.

References

[Azevedo&Sergot94] Azevedo P.J., Sergot M.J.
A Proof Procedure based on Recomputation-Free Lemmatization
in Journal of Computers and A.I., Vol. 13, 1994, No 2-3, pag. 257-273.

[Azevedo95] Azevedo P.J.
Techniques for Preventing Recomputation in Logic Programs
PhD Thesis, Department of Computing,
Imperial College of Science Technology and Medicine, London, 1995.

[Ramakrishnan et al. 92] Ramakrishnan R., Srivastava D., Sudarshan S.,
Controlling the Search in Bottom-up Evaluation
in Proceedings of the International Conference of Logic Programming,
Washington 1992.

[Sagonas et al. 94] Sagonas K., Swift T., Warren D.S.,
XSB as an Efficient Deductive Database Engine
in Proceedings of SIGMOD 1994 Conference ACM.

[Tamaki&Sato86] Tamaki H., Sato T.,
OLD Resolution with Tabulation
in Proceedings of the 3^{rd} International Conference on Logic Programming
London U.K. 1986, pp 84-98.

[Warren92] Warren D. S.,
Memoing for Logic Programs
in Communications of the ACM Vol 35, No 3, March 1992, pp 93-111.

[Vieille89] Vieille L.,
Recursive Query Processing: The Power of Logic
in Theoretical Computer Science, vol 69,
Elsevier Science Publishing 1989, pp 1-53.