On Extending a Linear Tabling Framework to Support Batched Scheduling

Miguel Areias and Ricardo Rocha

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

Abstract

Tabled evaluation is a recognized and powerful technique that overcomes some limitations of traditional Prolog systems in dealing with recursion and redundant sub-computations. During tabled execution, several decisions have to be made. These are determined by the scheduling strategy. Whereas a strategy can achieve very good performance for certain applications, for others it might add overheads and even lead to unacceptable inefficiency. The two most successful tabling scheduling strategies are *local scheduling* and *batched scheduling*. In previous work, we have developed a framework, on top of the Yap system, that supports the combination of different *linear tabling strategies* for local scheduling. In this work, we propose the extension of our framework, to support batched scheduling. In particular, we are interested in the two most successful linear tabling strategies, the DRA and DRE strategies. To the best of our knowledge, no single tabling Prolog system supports both strategies simultaneously for batched scheduling.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Linear Tabling, Scheduling, Implementation

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.9

1 Introduction

The operational semantics of Prolog is given by SLD resolution [7], an evaluation strategy particularly simple that matches current stack based machines particularly well, but that suffers from fundamental limitations, such as in dealing with recursion and redundant sub-computations. *Tabling* [3] is a proposal that overcomes those limitations. In a nutshell, tabling consists of storing intermediate solutions for subgoals so that they can be reused when a similar subgoal appears. Work on SLG resolution, as initially implemented in the XSB system [9], proved the viability of tabling technology for application areas such as Natural Language Processing, Knowledge Based Systems, Model Checking, Program Analysis, among others. Tabling based models are able to reduce the search space, avoid looping, and always terminate for programs with the *bounded term-size property* [3].

In a tabled evaluation, there are several points where we may have to choose between continuing forward execution, backtracking, consuming solutions from the table, or completing subgoals. The decision on which operation to perform is determined by the scheduling strategy. The two most successful strategies are *local scheduling* and *batched scheduling* [5]. Local scheduling tries to complete subgoals as soon as possible. When new solutions are found, they are added to the table space and the evaluation fails. Solutions are only returned when all program clauses for the subgoal at hand were resolved. Batched scheduling favors forward execution first, backtracking next, and consuming solutions or completion last. It thus tries to delay the need to move around the search tree by batching the return of solutions. When new solutions are found for a particular tabled subgoal, they are added to the table space and the evaluation continues.

The main difference between the two strategies is that in batched scheduling, variable bindings are immediately propagated to the calling environment when a solution is found. However, for some situations, this behavior may result in creating complex dependencies between subgoals. On the other hand, since local scheduling delays solutions, it does not benefit from variable propagation, and instead, when explicitly returning the delayed solutions, it incurs an extra overhead for copying them out of the table.

Currently, the tabling technique is widely available in systems like XSB [12], Yap [10], B-Prolog [13], ALS-Prolog [6], Mercury [11] and Ciao [4]. In these implementations, we can distinguish two main categories of tabling mechanisms: *suspension-based tabling* and *linear tabling*. Suspension-based tabling mechanisms need to preserve the computation state of suspended tabled subgoals in order to ensure that all solutions are correctly computed. A tabled evaluation can be seen as a sequence of sub-computations that suspend and later resume. Linear tabling mechanisms use iterative computations of tabled subgoals to compute fix-points and for that they maintain a single execution tree without requiring suspension and resumption of sub-computations. While suspension-based mechanisms are considered to obtain better results in general, they have more memory space requirements and are more complex and harder to implement than linear tabling mechanisms.

In previous work, we have developed a framework, on top of the Yap system, that supports the combination of different linear tabling strategies for local scheduling [1, 2]. As these strategies optimize different aspects of the evaluation, they were shown to be orthogonal to each other for local scheduling. In this work, we propose the extension of our framework, to combine different linear tabling strategies, but for batched scheduling. In particular, we are interested in the two most successful linear tabling strategies, the DRA and DRE strategies [2]. To the best of our knowledge, no single tabling Prolog system supports both strategies simultaneously for batched scheduling. Extending our framework from local scheduling to batched scheduling should be, in principle, smooth but, as we will see, there are some relevant details that have to be considered in order to ensure a correct and efficient integration of the DRA and DRE strategies with batched scheduling.

The remainder of the paper is organized as follows. First, we briefly introduce the basics of tabling and describe the execution model for standard linear tabled evaluation using batched scheduling. Next, we present the DRA and DRE strategies and discuss how they can be used to optimize different aspects of the evaluation. We then provide some implementation details regarding the integration of the two strategies on top of the Yap system. Finally, we present some experimental results and we end by outlining some conclusions.

2 Standard Linear Tabled Evaluation

Tabling works by storing intermediate solutions for tabled subgoals so that they can be reused when a similar (or repeated) call appears[1]. In a nutshell, first calls to tabled subgoals are considered *generators* and are evaluated as usual, using SLD resolution, but their solutions are stored in a global data space, called the *table space*. Similar calls to tabled subgoals are considered *consumers* and are not re-evaluated against the program clauses because they can potentially lead to infinite loops, instead they are resolved by consuming

[1] Two subgoal calls are considered to be similar if they are the same up to variable renaming.

the solutions already stored for the corresponding generator. During this process, as further new solutions are found, we need to ensure that they will be consumed by all the consumers, as otherwise we may miss parts of the computation and not fully explore the search space.

A generator call C thus keeps trying its matching clauses until a fix-point is reached. If no new solutions are found during one round of trying the matching clauses, then we have reached a fix-point and we can say that C is completely evaluated. However, if a number of subgoal calls is mutually dependent, thus forming a *Strongly Connected Component (SCC)*, then completion is more complex and we can only complete the calls in a SCC together [9]. SCCs are usually represented by the *leader call*, i.e., the generator call which does not depend on older generators. A leader call defines the next completion point, i.e., if no new solutions are found during one round of trying the matching clauses for the leader call, then we have reached a fix-point and we can say that all subgoal calls in the SCC are completely evaluated.

We next illustrate in Fig. 1 the standard execution model for linear tabling using batched scheduling. At the top, the figure shows the program code (the left box) and the final state of the table space (the right box). The program defines two tabled predicates, *a/1* and *b/1*, each defined by two clauses (clauses *c1* to *c4*). The bottom sub-figure shows the evaluation sequence for the query goal *a(X)*. Generator calls are depicted by black oval boxes and consumer calls by white oval boxes.

```
         :- table a/1, b/1.

c1)      a(X) :- b(X).
c2)      a(2).
c3)      b(X) :- a(X).
c4)      b(1).
```

Call	Solutions
1: a(X)	5: X=1 7: X=2 28: complete
2: b(X)	4: X=1 13: X=2 28: complete

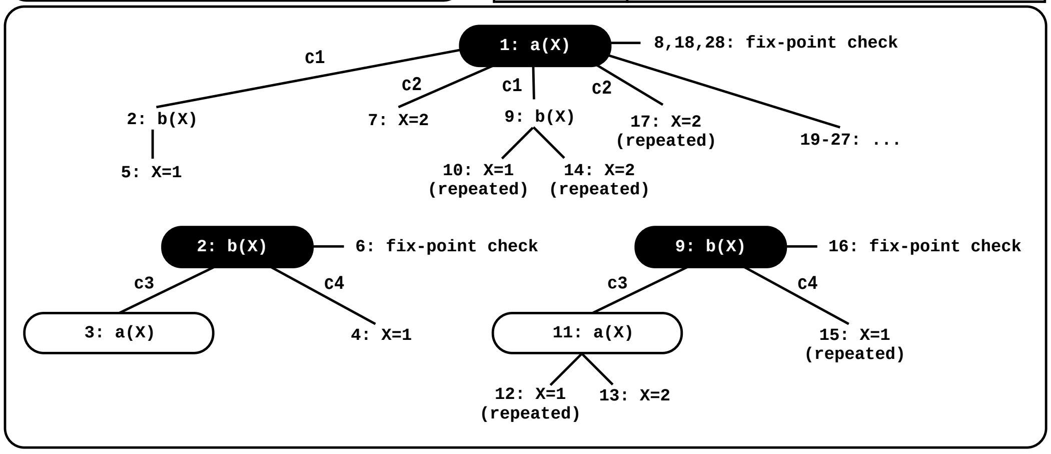


Figure 1 A standard linear tabled evaluation using batched scheduling.

The evaluation starts by inserting a new entry in the table space representing the generator call *a(X)* (step 1). Then, *a(X)* is resolved against its first matching clause, clause *c1*, calling *b(X)* in the continuation. As this is a first call to *b(X)*, we insert a new entry in the table space representing *b(X)* and proceed as shown in the bottom left tree (step 2). Subgoal *b(X)* is also resolved against its first matching clause, clause *c3*, calling again *a(X)* in the continuation (step 3). Since *a(X)* is a repeated call, we try to consume solutions from the table space, but at this stage no solutions are available, so execution fails.

We then try the second matching clause for *b(X)*, clause *c4*, and a first solution for *b(X)*, {*X=1*}, is found and added to the table space (step 4). We then follow a batched scheduling strategy and the evaluation continues with *forward execution* [5]. With batched scheduling, new solutions are immediately returned to the calling environment, thus the solution for *b(X)* should now be propagated to the context of the previous call, which originates a first solution for *a(X)*, {*X=1*} (step 5). The execution then fails back to node 2 and we check for a fix-point (step 6), but *b(X)* is not a leader call because it has a dependency (consumer node 3) to an older call, *a(X)*. Remember that we reach a fix-point when no new solutions are found during the last round of trying the matching clauses for the leader call. Then, we try the second matching clause for *a(X)* and a second solution, {*X=2*}, is found and added to the table space (step 7). We then backtrack again to the generator call for *a(X)* and because we have already explored all matching clauses, we check for a fix-point (step 8). We have found new solutions for both *a(X)* and *b(X)* in this round, thus the current SCC is scheduled for re-evaluation.

The evaluation then repeats the same sequence as in steps 2 to 3 (now steps 9 to 11), but since we are following a batched scheduling strategy, we first consume the solutions already available for *b(X)* (this will be further explained later in section 4), which leads to a repeated solution for *a(X)* (step 10). Tabling does not store duplicate solutions in the table space. Instead, repeated solutions fail. This is how tabling avoids unnecessary computations, and even looping in some cases. Next, the evaluation jumps to the consumer call of *a(X)* (step 11). Solution {*X=1*} is first forwarded to it, which originates a repeated solution for *b(X)* (step 12) and thus execution fails. Then, solution {*X=2*} is also forward to it and a new solution for *b(X)* is found (step 13) and propagated to *a(X)*, which leads to a repeated solution for *a(X)* (step 14).

In the continuation, we find another repeated solution for *b(X)* (step 15) and we fail a second time in the fix-point check for *b(X)* (step 16). Again, as we are following a batched scheduling strategy, the solutions for *b(X)* were already all propagated to the context of *a(X)*, thus we can safely backtrack to the generator call for *a(X)*. Because we have found a new solution for *b(X)* during this last round, the current SCC is scheduled again for re-evaluation (step 18). The re-evaluation of the SCC does not find new solutions for both *a(X)* and *b(X)* (steps 19 to 27). Thus, when backtracking again to *a(X)* we have reached a fix-point and because *a(X)* is a leader call, we can declare the two subgoal calls to be completed (step 28).

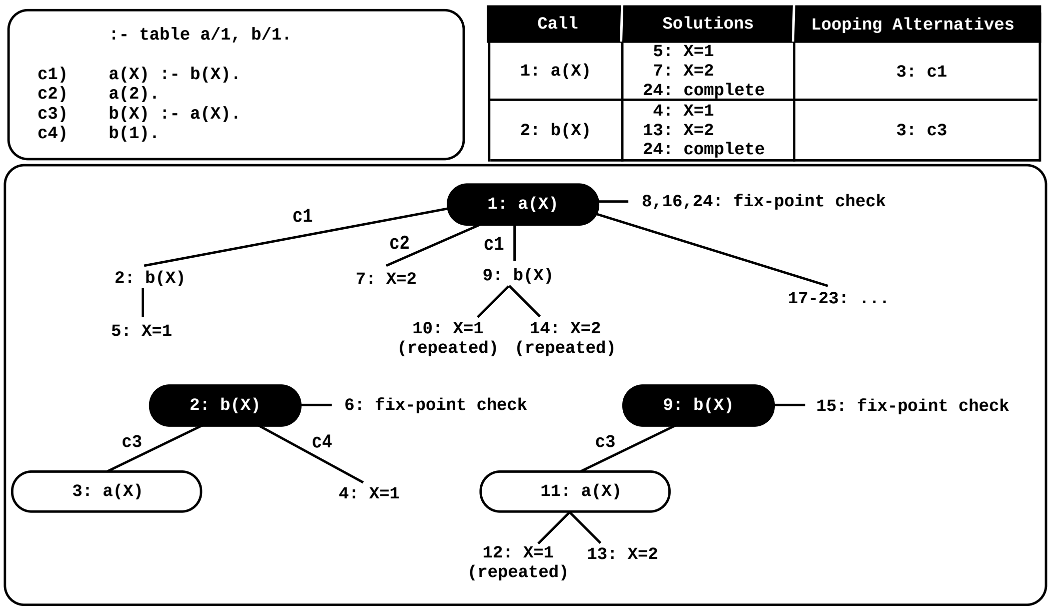
3 Linear Tabling Strategies

The standard linear tabling mechanism uses a naive approach to evaluate tabled logic programs. Every time a new solution is found during the last round of evaluation, the complete search space for the current SCC is scheduled for re-evaluation. However, some branches of the SCC can be avoided, since it is possible to know beforehand that they will only lead to repeated computations, hence not finding any new solutions. Next, we present two different strategies for optimizing the standard linear tabled evaluation. The common goal of both strategies is to minimize the number of branches to be explored, thus reducing the search space, and each strategy tries to focus on different aspects of the evaluation to achieve it.

3.1 Dynamic Reordering of Alternatives

The key idea of the *Dynamic Reordering of Alternatives (DRA)* strategy, as originally proposed by Guo and Gupta [6], is to memoize the clauses (or alternatives) leading to consumer calls, the *looping alternatives*, in such a way that when scheduling an SCC for re-evaluation, instead of trying the full set of matching clauses, we only try the looping alternatives.

Initially, a generator call C explores the matching clauses as in standard linear tabled evaluation and, if a consumer call is found, the current clause for C is memoized as a looping alternative. After exploring all the matching clauses, C enters the *looping state* and from this point on, it only tries the looping alternatives until a fix-point is reached. Figure 2 uses the same program from Fig. 1 to illustrate how DRA evaluation works.



```
        :- table a/1, b/1.

c1)     a(X) :- b(X).
c2)     a(2).
c3)     b(X) :- a(X).
c4)     b(1).
```

Call	Solutions	Looping Alternatives
1: a(X)	5: X=1 7: X=2 24: complete	3: c1
2: b(X)	4: X=1 13: X=2 24: complete	3: c3

Figure 2 A linear tabled evaluation using batched scheduling with DRA evaluation.

The evaluation sequence for the first SCC round (steps 2 to 7) is identical to the standard evaluation of Fig. 1. The difference is that this round is also used to detect the alternatives leading to consumers calls. We only have one consumer call at node 3 for *a(X)*. The clauses in evaluation up to the corresponding generator, call *a(X)* at node 1, are thus marked as looping alternatives and added to the respective table entries. This includes alternative *c3* for *b(X)* and alternative *c1* for *a(X)*. As for the standard strategy, the SCC is then scheduled for two extra re-evaluation rounds (steps 9 to 15 and steps 17 to 23), but now only the looping alternatives are evaluated, which means that the clauses *c2* and *c4* are ignored.

3.2 Dynamic Reordering of Execution

The second strategy, that we call *Dynamic Reordering of Execution (DRE)*, is based on the original SLDT strategy, as proposed by Zhou et al. [14]. The key idea of the DRE strategy is to give priority to the program clauses and, for that, it lets repeated calls to tabled subgoals execute from the *backtracking clause of the former call*. A first call to a tabled subgoal is called a *pioneer* and repeated calls are called *followers* of the pioneer. When backtracking to a pioneer or a follower, we use the same strategy and we give priority to the exploitation of the remaining clauses. The fix-point check operation is still performed by pioneer calls. Figure 3 uses again the same program from Fig. 1 to illustrate how DRE evaluation works.

As for the standard strategy, the evaluation starts with (pioneer) calls to *a(X)* (step 1) and *b(X)* (step 2), and then, in the continuation, *a(X)* is called repeatedly (step 3). With DRE evaluation, *a(X)* is now considered a follower and thus we *steal* the backtracking clause

```
        :- table a/1, b/1.

c1)    a(X) :- b(X).
c2)    a(2).
c3)    b(X) :- a(X).
c4)    b(1).
```

Call	Solutions
1: a(X)	4: X=2 8: X=1 19: complete
2: b(X)	5: X=2 7: X=1 19: complete

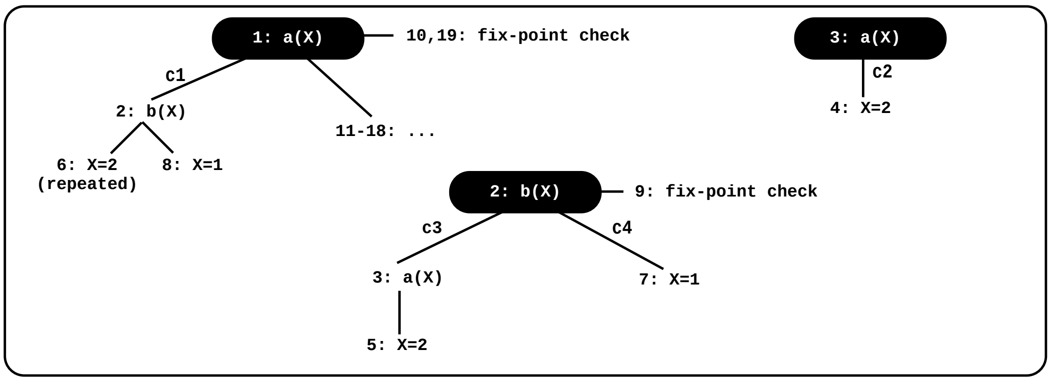


Figure 3 A linear tabled evaluation using batched scheduling with DRE evaluation.

of the former call at node 1, i.e., clause *c2*. The evaluation then proceeds as for a generator call (right upper tree in Fig. 3), which means that new solutions can be generated for *a(X)*. We thus try clause *c2* and a first solution for *a(X)*, $\{X{=}2\}$, is found and added to the table space (step 4). Then, we follow a batched scheduling strategy and the solution $\{X{=}2\}$ is propagated to the context of *b(X)*, which originates the solution $\{X{=}2\}$ (step 5), and to the context of *a(X)*, which leads to a repeated solution (step 6).

As both matching clauses for *a(X)* were already taken, the execution backtracks to the pioneer node 2. Next, we find a second solution for *b(X)* (step 7), which is then propagated, leading also to a second solution for *a(X)* (step 8). In step 9, we check for a fix-point, but *b(X)* is not a leader call because it has a dependency (follower node 3) to an older call, *a(X)*. We then backtrack to the pioneer call for *a(X)* and because we have already explored the matching clause *c2* in the follower node 3, we check for a fix-point. Since we have found new solutions during the last round, the current SCC is scheduled for re-evaluation (step 10). The re-evaluation of the SCC does not find any further solutions (steps 11 to 18), and thus the evaluation can be completed at step 19.

4 Propagation of Solutions in Re-evaluation Rounds

In the previous sections, one could observe that tabling does not store duplicate solutions in the table space and, instead, repeated solutions fail. This is how tabling avoids unnecessary computations, and even looping in some cases. However, since repeated solutions also fail in re-evaluation rounds, this means that, in fact, a solution is only propagated once, i.e., in the round it is first found, which might be not sufficient to ensure the completeness of the evaluation. To solve this problem, this is why, in a re-evaluation round, we start by propagating (consuming) the solutions already available for the subgoal call at hand. Alternatively, we could propagate the solutions at the end, after the fix-point check procedure, but by doing that some solutions will be propagated more than once in a single round, which is worthless.

In the previous examples, for simplicity of explanation, we have omitted some steps regarding the propagation of solutions since, for all the examples, one propagation per solution was enough to correctly compute the corresponding evaluations. To better illustrate the importance of the propagation of solutions in re-evaluation rounds, Fig. 4 shows a new example, using again the same program from Fig. 1, but for the query goal *a(X), b(Y)*. For simplicity of explanation, we consider a standard linear tabled evaluation, i.e., without DRA and DRE support.

```
        :- table a/1, b/1.

c1)     a(X) :- b(X).
c2)     a(2).
c3)     b(X) :- a(X).
c4)     b(1).
```

Call	Solutions
1: a(X)	5: X=1 9: X=2 41: complete
2: b(X)	4: X=1 21: X=2 41: complete

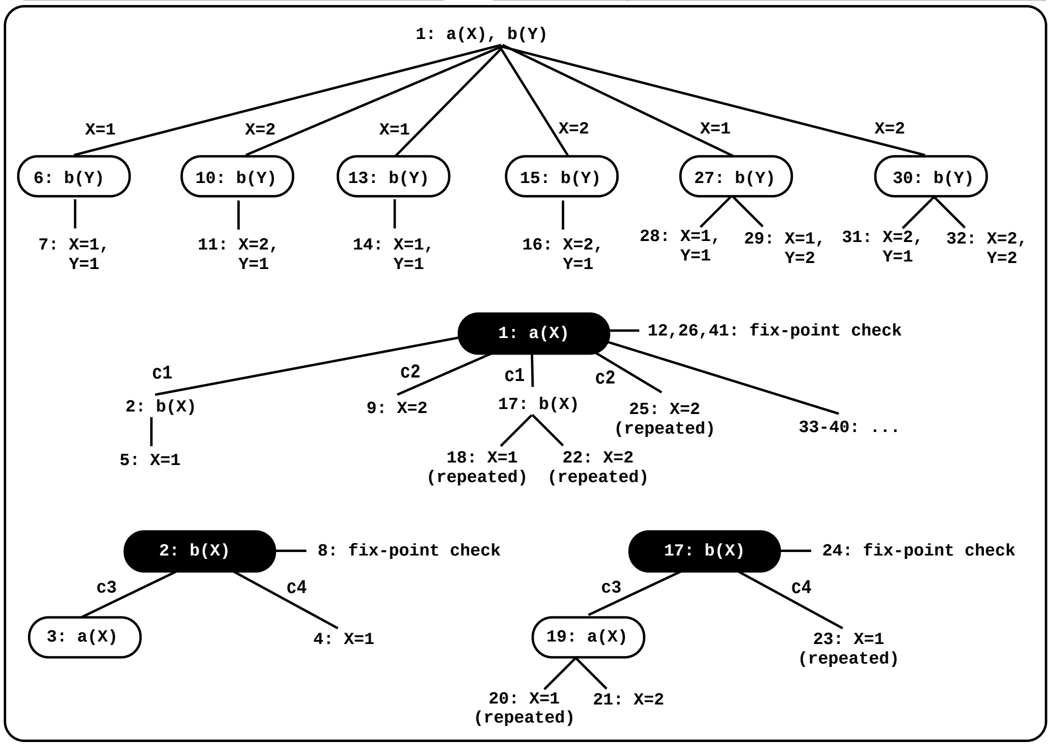


Figure 4 Propagation of solutions in re-evaluation rounds using batched scheduling.

In the first round of the evaluation (steps 1 to 12), the solutions found for *a(X)*, at steps 5 and 9, are propagated to the context of the top query goal and, in the continuation, *b(Y)* consumes (note that *b(Y)* is a variant call of *b(X)*) the available answer found for *b(X)* at step 4, which originates the solutions $\{X{=}1,\ Y{=}1\}$ (step 7) and $\{X{=}2,\ Y{=}1\}$ (step 11) for the top query goal.

Next, in the second round of the evaluation (steps 13 to 26), the generator node for *a(X)* starts by propagating its solutions (steps 13 to 16), but only repeated solutions are found for the top query goal (steps 14 and 16). Later, at step 21, a new solution, $\{X{=}2\}$, is found for *b(X)*. The combination of this new solution with the previous solutions for *a(X)* should originate new solutions for the top query goal. The solution for *b(X)* is then propagated to the context of *a(X)* (step 22) but, since this originates a repeated solution for *a(X)*, the computation fails.

By failing for *a(X)*, we cannot combine the new answer for *b(X)* with the previous answers for *a(X)* in this round. Hence, this fact, i.e., the fact that tabling fails for repeated solutions, can lead to a collateral effect where it can be blocking forward execution. To solve this problem, this is why, in a re-evaluation round, we start by propagating all the available solutions. Therefore, in the third and final round of the evaluation (steps 27 to 40), the generator node for *a(X)* starts again by propagating its solutions (steps 28 to 32) and the solutions $\{X{=}1,\ Y{=}2\}$ (step 29) and $\{X{=}2,\ Y{=}2\}$ (step 32) are generated. Since this round does not find any new solution, the evaluation is finally completed at step 41.

5 Implementation Details

This section describes some implementation details regarding the extension of our framework to support batched scheduling, with particular focus on the table space data structures and on the tabling operations.

5.1 Table Space

To implement the table space, Yap uses *tries* which is considered a very efficient way to implement the table space [8]. Tries are trees in which common prefixes are represented only once. Tries provide complete discrimination for terms and permit look up and insertion to be done in a single pass. Figure 5 details the table space organization for the example used on the previous sections.

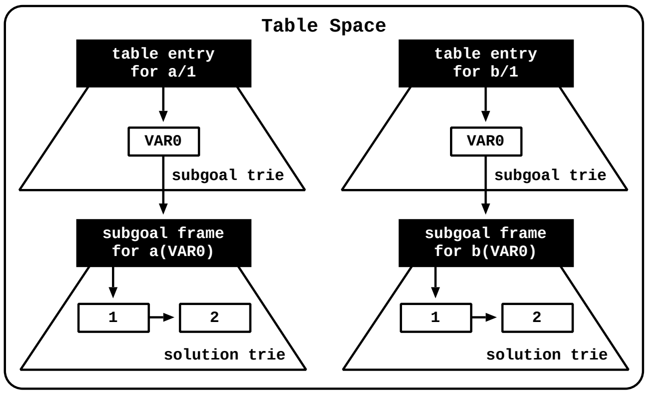


Figure 5 Table space organization.

As other tabling engines, Yap uses two levels of tries: one for the subgoal calls and other for the computed solutions. A tabled predicate accesses the table space through a specific *table entry* data structure. Each different subgoal call is represented as a unique path in the *subgoal trie* and each different solution is represented as a unique path in the *solution trie*. A key data structure in this organization is the *subgoal frame*. Subgoal frames are used to store information about each tabled subgoal call, namely: the entry point to the solution trie; the state of the subgoal (*ready*, *evaluating* or *complete*); support to detect if the subgoal is a leader call; and support to detect if new solutions were found during the last round of evaluation. The DRA and DRE strategies extend the subgoal frame data structure with the following extra information [2]:

DRA: support to detect, store and load looping alternatives; and two new states, *loop_ready* and *loop_evaluating*, used to detect, respectively, generator and consumer calls in re-evaluating rounds.

DRE: the pioneer call; and the backtracking clause of the former call.

To support the propagation of solutions, as discussed in section 4, an extra field, named *SgFr_last_consumed*, marks the last solution consumed in a pioneer or follower call.

5.2 Tabling Operations

We next introduce the pseudo-code for the main tabling operations required to support batched scheduling with DRA and DRE evaluation.

We start with Fig. 6 showing the pseudo-code for the *new solution* operation. Initially, the operation simply inserts the given solution *SOL* in the solution trie structure for the given subgoal frame *SF* and, if the solution is new, it updates the *SgFr_new_solutions* subgoal frame field to *TRUE*. Then, for batched scheduling, it adjusts the execution's environment and proceeds with forward execution.

```
new_solution(solution SOL, subgoal frame SF) {
  if (solution_check_insert(SOL,SF) == TRUE)                      // new solution
    SgFr_new_solutions(SF) = TRUE
  else                                                       // repeated solution
    fail()
  if (batched_scheduling_mode(SF))                          // batched scheduling
    proceed()
  else { ... }                                                // local scheduling
}
```

Figure 6 Pseudo-code for the *new solution* operation.

Figure 7 shows the pseudo-code for the *tabled call* operation. New calls to tabled subgoals are inserted into the table space by allocating the necessary data structures. This includes allocating and initializing a new subgoal frame *SF* to represent the given subgoal call *SC* (this is the case where the state of *SF* is *ready*). In such case, the tabled call operation then stores a new generator node[2]; updates the state of *SF* to *evaluating*; and proceeds by executing the current alternative.

On the other hand, if the subgoal call is a repeated call, then the subgoal frame *SF* is already in the table space, and three different situations may occur. First, if the call is already evaluated (this is the case where the state of *SF* is *complete*), the operation consumes the available solutions by implementing the *completed table optimization* which executes compiled code directly from the solution trie structure associated with the completed call [8].

Second, if the call is a first call in a re-evaluating round (this is the case where the state of *SF* is *loop_ready*), the operation stores a new generator node; updates the state of *SF* to *loop_evaluating*; and resets the *SgFr_last_consumed* field to the first solution. Then, it executes the *consume_solutions_and_execute()* procedure in order to consume the available solutions before re-evaluate the matching alternatives. This procedure, consumes all the available solutions for the subgoal, starting from the first solution, and, when no more solutions are to be consumed, it starts with the evaluation of the first matching alternative, which for DRA is the first looping alternative.

Third, if the call is a repeated call (this is the case where the state of *SF* is *evaluating* or *loop_evaluating*), the operation first marks the current branch as a non-leader branch and, if in DRA, it also marks the current branch as a looping branch. Next, if DRE mode is enabled and there are unexploited alternatives (i.e., there is a backtracking clause for the

[2] Generator, consumer and follower nodes are implemented as regular WAM choice points extended with some extra fields related to the table space data structures.

```
tabled_call(subgoal call SC) {
  SF = subgoal_check_insert(SC)                    // SF is the subgoal frame for SC
  if (SgFr_state(SF) == ready) {                                     // first round
    store_generator_node()
    SgFr_state(SF) = evaluating
    goto execute(current_alternative())
  } else if (SgFr_state(SF) == complete) {                     // already evaluated
    goto completed_table_optimization(SF)
  } else if (SgFr_state(SF) == loop_ready) {                 // re-evaluation round
    store_generator_node()
    SgFr_state(SF) = loop_evaluating
    if (batched_scheduling_mode(SF)){
      SgFr_last_consumed(SF) = SgFr_first_solution(SF)
      if (DRA_mode(SF))
        goto consume_solutions_and_execute(SF,first_looping_alternative())
      else
        goto consume_solutions_and_execute(SF,first_alternative())
    } else { ... }                                              // local scheduling
  } else if (SgFr_state(SF) == evaluating or                         // first round
             SgFr_state(SF) == loop_evaluating) {            // re-evaluation round
    mark_current_branch_as_a_non_leader_branch(SF)
    if (DRA_mode(SF))
      mark_current_branch_as_a_looping_branch(SF)
    if (DRE_mode(SF) && has_unexploited_alternatives(SF)) {
      store_follower_node()
      if (DRA_mode(SF) and SgFr_state(SF) == loop_evaluating)
        goto consume_solutions_and_execute(SF,next_looping_alternative())
      else
        goto consume_solutions_and_execute(SF,next_alternative())
    } else {
      store_consumer_node()
      goto consume_solutions(SF)
    }
  }
}
```

Figure 7 Pseudo-code for the *tabled call* operation.

former call), it stores a follower node and proceeds by consuming the available solutions before executing the next looping alternative or the next matching alternative, according to whether the DRA mode is enabled or disabled for the subgoal. Otherwise, it stores a new consumer node and starts consuming the available solutions.

To mark the current branch as a non-leader branch, we follow all intermediate generator calls in evaluation up to the generator call for frame *SF* and we mark them as non-leader calls (note that the call at hand defines a new dependency for the current SCC). To mark the current branch as a looping branch, we follow all intermediate generator calls in evaluation up to the generator call for frame *SF* and we mark the alternatives being evaluated by each call as looping alternatives.

Finally, we discuss in more detail how completion is detected with batched scheduling. Remember that after exploring the last matching clause for a tabled call, we execute the *fix-point check* operation. Figure 8 shows the pseudo-code for its implementation.

The fix-point check operation starts by verifying if the subgoal at hand is a leader call. If it is leader and has found new solutions during the last round, then the current SCC is scheduled for a re-evaluation. For batched scheduling, this is the same situation as for a first call in a re-evaluating round in the *tabled call* operation, i.e., it resets the *SgFr_last_consumed* field to the first solution and executes the *consume_solutions_and_execute()* procedure. If

```
fix-point_check(subgoal frame SF) {
  if (SgFr_is_leader(SF)){
    if (SgFr_new_solutions(SF)) {                              // start a new round
      SgFr_new_solutions(SF) = FALSE
      for each (subgoal in current SCC)
        SgFr_state(subgoal) = loop_ready
      SgFr_state(SF) = loop_evaluating
      if (batched_scheduling_mode(SF)) {
        SgFr_last_consumed(SF) = SgFr_first_solution(SF)
        if (DRA_mode(SF))
          goto consume_solutions_and_execute(SF,first_looping_alternative())
        else
          goto consume_solutions_and_execute(SF,first_alternative())
      } else { ... }                                            // local scheduling
    } else {                                   // complete subgoals in current SCC
      for each (subgoal in current SCC)
        SgFr_state(subgoal) = complete
      if (batched_scheduling_mode(SF))
        fail()
      else { ... }                                              // local scheduling
  } else {                                                     // not a leader call
    if (SgFr_new_solutions(SF))                          // propagate new solutions
      SgFr_new_solutions(current_leader(SF)) = TRUE
    SgFr_new_solutions(SF) = FALSE                           // reset new solutions
    if (batched_scheduling_mode(SF))
      fail()
    else { ... }                                                // local scheduling
  }
}
```

Figure 8 Pseudo-code for the *fix-point check* operation.

the subgoal is leader but no new solutions were found during the current round, then we have reached a fix-point and thus, the subgoals in the current SCC are marked as completed and the evaluation fails.

Otherwise, if the subgoal is not a leader call, then it propagates the new solutions information to the current leader of the SCC and the evaluation fails. Note that, with batched scheduling, we can safely fail since all the solutions were already propagated to the context of the calling environment. Moreover, since the *SgFr_new_solutions* flag is propagated to the leader of the SCC, the leader will mark the SCC for a new evaluation round, which means that the current subgoal will be called again, and so it will start by consuming all its solutions.

6 Experimental Results

To the best of our knowledge, Yap is now the first tabling engine that integrates and supports the combination of different linear tabling strategies using batched scheduling. We have thus the conditions to better understand the advantages and weaknesses of each strategy when used solely or combined. In what follows, we present initial experiments comparing linear tabled evaluation with and without DRA and DRE support, using batched scheduling. To put our results in perspective, we have also included experiments for the B-Prolog linear tabling system [13] and for the YapTab suspension-based tabling system [10], both using batched scheduling. In fact, for B-Prolog, we used its *eager scheduling mode*, which is similar to batched scheduling. The environment for our experiments was a PC with a 2.83 GHz Intel(R) Core(TM)2 Quad CPU and 8 GBytes of memory running the Linux kernel

3.0.0-16-generic. We used B-Prolog version 7.5 and Yap version 6.0.7.

For benchmarking, we used three sets of programs. The **Model Checking** set includes three different specifications and transition relation graphs usually used in model checking applications. The **Path Right** set implements the right recursive definition of the well-known *path*/2 predicate, that computes the transitive closure in a graph, using three different edge configurations. Figure 9 shows an example for each configuration. We experimented the **Pyramid** and **Cycle** configurations with depths 1000, 2000 and 3000 and the **Grid** configuration with depths 20, 30 and 40. We chose this set of experiments because the *path*/2 predicate implements a relatively easy to understand pattern of computation and its right recursive definition creates several inter-dependencies between tabled subgoals. The **Warren** set is a variation of the left recursive definition of the path problem for a linear graph (see Fig. 9), where the *path/2* clauses are duplicated to be used with the labels *a* and *b*. This problem was kindly suggested by David S. Warren as a way to stress the performance of a linear tabling system. All benchmarks find all the solutions for the problem.

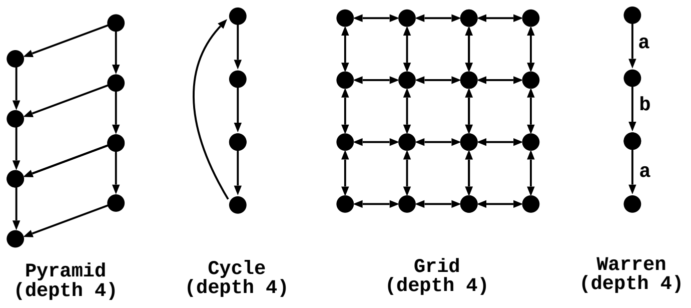


Figure 9 Edge configurations used with the second and third set of problems.

In Table 1, we show the execution time, in milliseconds, for standard linear tabling (column **Std**), DRA and DRE evaluation, solely and combined (column **DRA+DRE**), B-Prolog and YapTab, using batched scheduling, and the respective performance ratios when compared with standard linear tabling, for the **Model Checking**, **Path Right** and **Warren** sets of problems. Ratios higher than 1.00 mean that the respective strategy has a positive impact on the execution time. The ratio marked with *n.c.* for B-Prolog means that we are *not considering* it in the average results (for some reason, we failed in executing this benchmark). The results are the average of five runs for each benchmark.

In addition to the results presented in Table 1, we also collected several statistics regarding important aspects of the evaluation (not fully presented here due to lack of space). In Table 2, we show some of these statistics for standard linear tabling and the respective performance ratios when compared with the other models, for a subset of the benchmarks. We used the **Leader** specification for the **Model Checking** set, the configurations **Pyramid** and **Cycle** with depth 2000 and **Grid** with depth 30 for the **Path Right** set, and the configuration with depth 600 for the **Warren** set.

The statistics in Table 2 measure how the mixing with SLD (non-tabled) computations can affect the base performance of our benchmarks. For that, we extended the tabled predicates, at the beginning and at the end of each clause, with dummy SLD (non-tabled) predicates, which we named *sldi/0*, with $0 < i \leq 2n$, where n is the number of tabled clauses. For example, the extended definition for the *path/2* predicate is:

```
path(X,Z) :- sld1, edge(X,Y), path(Y,Z), sld2.
path(X,Z) :- sld3, edge(X,Z), sld4.
```

Table 1 Execution time, in milliseconds, for standard linear tabling, DRA and DRE evaluation, solely and combined, B-Prolog and YapTab, using batched scheduling, and the respective ratios when compared with standard linear tabling (for the linear tabling models, best ratios are in bold).

Bench	Std	DRA		DRE		DRA+DRE		B-Prolog		YapTab	
Model Checking											
IProto	2,874	2,879	**(1.00)**	5,766	(0.50)	3,098	(0.93)	8,087	(0.36)	1,201	(2.39)
Leader	5,355	5,288	**(1.01)**	13,423	(0.40)	5,430	(0.99)	40,624	(0.13)	1,891	(2.83)
Sieve	35,218	35,350	**(1.00)**	76,927	(0.46)	38,048	(0.93)	217,155	(0.16)	11,046	(3.19)
	Average		(1.00)		(0.45)		(0.95)		(0.22)		(2.80)
Path Right - Pyramid											
1000	983	526	**(1.87)**	1,102	(0.89)	658	(1.49)	948	(1.04)	517	(1.90)
2000	3,897	2,071	**(1.88)**	4,380	(0.89)	2,611	(1.49)	5,630	(0.69)	2,013	(1.94)
3000	9,043	4,740	**(1.91)**	10,110	(0.89)	5,920	(1.53)	—	(*n.c.*)	4,561	(1.98)
Path Right - Cycle											
1000	687	539	**(1.27)**	713	(0.96)	563	(1.22)	540	**(1.27)**	362	(1.89)
2000	2,793	2,198	**(1.27)**	2,891	(0.97)	2,286	(1.22)	3,079	(0.91)	1,534	(1.82)
3000	6,048	4,681	**(1.29)**	6,343	(0.95)	4,949	(1.22)	8,678	(0.70)	2,956	(2.05)
Path Right - Grid											
20	221	166	**(1.33)**	227	(0.97)	174	(1.27)	202	(1.09)	105	(2.10)
30	1,344	1,015	**(1.32)**	1,362	(0.99)	1,036	(1.30)	1,318	(1.02)	605	(2.22)
40	4,578	3,508	**(1.31)**	4,697	(0.97)	3,630	(1.26)	5,995	(0.76)	1,958	(2.34)
	Average		**(1.50)**		(0.94)		(1.33)		(0.93)		(2.03)
Warren											
400	2,673	2,632	(1.02)	42	**(64.26)**	42	**(64.26)**	7,861	(0.34)	21	(126.09)
600	9,496	9,564	(0.99)	109	**(87.28)**	109	**(87.28)**	27,302	(0.35)	58	(162.61)
800	23,163	23,086	(1.00)	205	(112.88)	198	**(116.98)**	67,049	(0.35)	107	(216.88)
	Average		(1.00)		**(87.93)**		**(89.51)**		(0.35)		(168.53)

The rows in Table 2 show the number of times each dummy SLD predicate is called for the corresponding benchmark. We can read these numbers as an estimation of the performance ratios that we will obtain if the execution time of the corresponding SLD predicate clearly overweights the execution time of the other computations. Note that the odd SLD predicates (such as **sld1** and **sld3**) correspond to re-executions of a clause and that the even SLD predicates (such as **sld2** and **sld4**) correspond to new answer operations. In our experiments, the **sld2** predicate (placed at the end of the first tabled clause) is the one that can potentially have a greater influence in the performance ratios as it clearly exceeds all the others in the number of times it is called (see Table 2).

Analyzing the general picture of Table 1, the results show that DRA evaluation is able to reduce the execution time for the **Path Right** problem set (1.50 times faster, on average) but has no impact for the other two sets, when compared with standard evaluation. The results also indicate that DRE evaluation has a negative impact in the execution time for the **Model Checking** and **Path Right** sets but, on the other hand, it can significantly reduce the execution time for the **Warren** set (more than 80 times faster, on average). We next discuss in more detail each strategy.

Table 2 Number of calls to the dummy SLD predicates for standard linear tabling and the respective ratios when compared with DRA and DRE evaluation, solely and combined, B-Prolog and YapTab, using batched scheduling (for the linear tabling models, best ratios are in bold).

Bench	Std	DRA	DRE	DRA+DRE	B-Prolog	YapTab
Model Checking - Leader						
sld1	3	1.00	0.75	1.00	1.00	3.00
sld2	1,153,026	1.00	0.40	1.00	1.00	2.00
sld3	3	**3.00**	0.75	**3.00**	**3.00**	3.00
sld4	3	**3.00**	0.75	**3.00**	**3.00**	3.00
Path Right - Pyramid 2000						
sld1	7,999	**2.00**	1.00	**2.00**	**2.00**	2.00
sld2	37,951,017	**2.38**	0.86	1.73	**2.38**	2.38
sld3	7,999	**2.00**	1.00	**2.00**	**2.00**	2.00
sld4	23,988	**2.00**	1.00	**2.00**	**2.00**	2.00
Path Right - Cycle 2000						
sld1	6,002	1.00	1.00	1.00	1.00	3.00
sld2	18,003,000	**1.29**	1.00	**1.29**	**1.29**	2.25
sld3	6,002	**3.00**	1.00	**3.00**	**3.00**	3.00
sld4	10,000	**2.50**	1.00	**2.50**	**2.50**	2.50
Path Right - Grid 30						
sld1	2,702	1.00	1.00	1.00	0.18	3.00
sld2	13,851,534	1.29	1.00	**1.30**	0.30	2.21
sld3	2,702	**3.00**	1.00	1.02	**3.00**	3.00
sld4	17,400	**2.50**	1.00	1.27	**2.50**	2.50
Warren - 600						
sld1/sld3	302	1.00	**100.67**	**100.67**	1.00	302.00
sld2/sld4	18,044,650	1.00	66.98	**100.42**	1.00	201.17
sld5/sld7	302	**302.00**	100.67	**302.00**	**302.00**	302.00
sld6/sld8	90,600	**302.00**	100.67	**302.00**	**302.00**	302.00

DRA: the results for DRA evaluation show that the strategy of avoiding the exploration of non-looping alternatives in re-evaluation rounds is quite effective in general and does not add extra overheads when not used. The results also show that, for the **Path Right** set, DRA is more effective for programs without loops, like the **Pyramid** configurations, than for programs with larger SCCs, like the **Cycle** and **Grid** configurations. On Table 2, we can observe that the number of dummy SLD computations is, in fact, effectively reduced with DRA evaluation.

DRE: for the **Model Checking** set, DRE evaluation is around two times slower than standard evaluation and, for the **Path Right** set, DRE has no significant impact for all the configurations. Table 2 confirms that, the strategy of allocating follower nodes, adds an extra complexity to the evaluation for the **Model Checking** set (the number of dummy SLD calls is higher) and that it has no impact for the **Path Right** set (the number of dummy SLD calls is identical to standard evaluation). For the **Warren** set, DRE evaluation produces the most interesting results. Note that, this is the set of benchmarks where suspension-based tabling (the YapTab system) is far more faster than standard linear tabling (168.53 times faster, on average) and the difference increases as the depth of the problem also increases. However, DRE evaluation is able to reduce this huge difference to a minimum. On average, DRE evaluation is 87.93 times faster than standard evaluation and the scalability, as the depth of the problem increases, is similar to the one observed for YapTab. Table 2 confirms this behavior for DRE and YapTab evaluations (the number of dummy SLD calls is clearly lower than standard evaluation).

Regarding the combination of both strategies (DRA+DRE), our experiments show that, in general, the best of both worlds is always present in the combination. The results in Table 1 show that, by combining both strategies, DRA is able to avoid DRE behavior for the **Model Checking** and **Path Right** sets. Still, the results for DRA+DRE are slightly worst than DRA used solely. For the **Warren** set, the results show that, by combining both strategies, it is possible to reduce even further the execution time when compared with DRE used solely. In particular, one can observe that, for depths 400 and 600, the execution times are the same but, for depth 800, DRA+DRE evaluation outperforms DRE used solely.

The statistics on Table 2 confirm that, in general, the best of both worlds is always present in the combination. The exceptions are the **sld2** predicate, for the **Pyramid 2000** configuration, and the **sld3** and **sld4** predicates, for the **Grid 30** configuration. On the other hand, for the **Warren 600** configuration, the **sld1/sld3** predicates are executed the same number of times as for DRE used solely, the **sld5** to **sld8** predicates are executed the same number of times as for DRA used solely, and the **sld2** and **sld4** predicates are executed less times than both strategies used solely, which is explained by the fact that the fix-point is achieved in less rounds (statistics not shown here due to lack of space).

Regarding the comparison with the B-Prolog linear tabling system, the results in Table 2 suggest that B-Prolog implements a DRA-based evaluation strategy since the statistics for B-Prolog and DRA evaluation are all the same, except for the **sld1** and **sld2** predicates in the **Grid 30** configuration. However, the execution times in Table 1 show that our DRA implementation is always faster than B-Prolog in these experiments and that, for almost all configurations, the ratio difference shows a generic tendency to increase as the depth of the problem also increases.

For all experiments, the results obtained for the YapTab suspension-based system clearly outperform the standard linear tabled evaluation but, for our DRA+DRE implementation, they are globally comparable. On average, YapTab is around 2 times faster than DRA+DRE evaluation, including the **Warren** problem set, where YapTab shows a huge difference for standard linear tabling. The results also indicate that our implementation scales as well as YapTab when we increase the depth of the problem being tested.

7 Conclusions

We have presented a new linear tabling framework that integrates and supports batched scheduling with DRA and DRE evaluation, solely or combined. We discussed how these strategies can optimize different aspects of a tabled evaluation and we presented the relevant implementation details for their integration on top of the Yap system.

Our experimental results were very interesting and very promising. In particular, the combination of DRA with DRE showed the potential of our framework to effectively reduce the execution time of the standard linear tabled evaluation. When compared with YapTab's suspension-based mechanism, the commonly referred weakness of linear tabling of doing a huge number of redundant computations for computing fix-points was not such a problem in our experiments. We thus argue that an efficient implementation of linear tabling can be a good and first alternative to incorporate tabling into a Prolog system without such support.

Further work will include adding new strategies/optimizations to our framework, and exploring the impact of applying our strategies to more complex problems, seeking real-world experimental results, allowing us to improve and consolidate our current implementation.

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 M. Areias and R. Rocha. An Efficient Implementation of Linear Tabling Based on Dynamic Reordering of Alternatives. In *International Symposium on Practical Aspects of Declarative Languages*, number 5937 in LNCS, pages 279–293. Springer-Verlag, 2010.

2 M. Areias and R. Rocha. On Combining Linear-Based Strategies for Tabled Evaluation of Logic Programs. *Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue*, 11(4–5):681–696, 2011.

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

4 P. Chico, M. Carro, M. V. Hermenegildo, C. Silva, and R. Rocha. An Improved Continuation Call-Based Implementation of Tabling. In *International Symposium on Practical Aspects of Declarative Languages*, number 4902 in LNCS, pages 197–213. Springer-Verlag, 2008.

5 J. Freire, T. Swift, and D. S. Warren. Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In *International Symposium on Programming Language Implementation and Logic Programming*, number 1140 in LNCS, pages 243–258. Springer-Verlag, 1996.

6 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.

7 J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

8 I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Access Mechanisms for Tabled Logic Programs. *Journal of Logic Programming*, 38(1):31–54, 1999.

9 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.

10 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.

11 Z. Somogyi and K. Sagonas. Tabling in Mercury: Design and Implementation. In *International Symposium on Practical Aspects of Declarative Languages*, number 3819 in LNCS, pages 150–167. Springer-Verlag, 2006.

12 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.

13 Neng-Fa Zhou. The Language Features and Architecture of B-Prolog. *Journal of Theory and Practice of Logic Programming*, 12(1 & 2):189–218, 2012.

14 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.