Implementation of Suspension-Based Tabling in Prolog using External Primitives

Ricardo Rocha, Cláudio Silva, and Ricardo Lopes*

DCC-FC & LIACC
University of Porto, Portugal
ricroc@dcc.fc.up.pt ccaldas@dcc.online.pt rslopes@dcc.fc.up.pt

Abstract. We present the design, implementation and evaluation of a suspension-based tabling mechanism that supports tabled evaluation by applying source level transformations to a tabled program. The transformed program then uses external tabling primitives that provide direct control over the evaluation strategy. To implement the tabling primitives we took advantage of the C language interface of the Yap Prolog system. Initial results show that our suspension-based mechanism is comparable to the state-of-the-art YapTab system, that implements tabling support at the low-level engine. This is an interesting result because YapTab also implements a suspension-based mechanism, uses the same data structures to implement the table space and is implemented on top of Yap. We thus argue that our approach is a good alternative to incorporate tabling into any Prolog system. It requires neither advanced knowledge of the implementation details of tabling nor time consuming or complex modifications to the low-level engine.

1 Introduction

Tabling [1] is a technique of resolution that overcomes some limitations of traditional Prolog systems in dealing with redundant sub-computations and recursion. Tabling has become a popular and successful technique thanks to the ground-breaking work in the XSB Prolog system and in particular in the SLG-WAM [2], the most successful engine of XSB. The success of SLG-WAM led to several alternative implementations that differ in the execution rule, in the data-structures used to implement tabling, and in the changes to the underlying Prolog engine. Tabling mechanisms are now widely available in systems like YapTab [3], ALS-Prolog [4], B-Prolog [5] and Mercury [6].

The common approach used by all these systems to support tabled evaluation is to modify and extend the low-level engine. Although this approach is ideal for run-time efficiency, it is not easily portable to other Prolog systems as engine level modifications are rather complex and time consuming. A different approach to

* This work has been partially supported by Myddas (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC.

incorporate tabled evaluation into existing Prolog systems is to apply source level transformations to a tabled program and then use external tabling primitives to provide direct control over the search strategy. This idea was first explored by Fan and Dietrich [7] that implemented a form of linear tabling using source level program transformation and tabling primitives implemented as Prolog built-ins.

In this work, we present the design, implementation and evaluation of a suspension-based tabling mechanism based on program transformation, but we use the C language interface, available in most Prolog systems, to implement the tabling primitives. In particular, we use the C interface of the Yap Prolog system [8] to build an external Prolog module implementing the support for tabled evaluation. To implement our mechanism, that we named *tabled evaluation with continuation calls*, we follow a *local scheduling* strategy [9] and we use *tries* [10] to implement the table space data structures. To implement the program transformation step, we have extended the original program transformation module of Ramesh and Chen [11] to include tabling primitives for our approach.

Initial results comparing our mechanism with the state-of-the-art YapTab system, that implements tabling support at the low-level engine, are very promising. This is an interesting result because YapTab also implements a suspension-based mechanism, uses tries to implement the table space and is implemented on top of Yap. This is thus a first and fair comparison between the approach of supporting tabling at the low-level engine and the approach of supporting tabling by applying source level transformations coupled with tabling primitives. We thus argue that our approach is a good alternative to incorporate tabling into any Prolog system. It requires neither advanced knowledge of the implementation details of tabling nor time consuming or complex modifications to the low-level engine.

The remainder of the paper is organized as follows. First, we show how a tabled program is transformed to include specific tabling primitives and present an evaluation example showing the interaction with Prolog execution. We then provide the details for implementing our mechanism as an external Prolog module written in C and discuss how completion is detected. At last, we present some experimental results and we end by outlining some conclusions.

2 Tabled Evaluation with Continuation Calls

Whenever a tabled subgoal is first called, a new entry is allocated in the table space and the evaluation begins with a generator node exploring the first clause for the corresponding tabled predicate. On the other hand, variant 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. When a consumer exhausts the set of available answers, the computation state for the variant call is *suspended*. In this model, suspension is implemented by leaving a *continuation call* for the current computation in the table entry corresponding to the variant call being suspended. During this process and as further new answers are found, they are stored in their tables and returned to all variant

calls by calling the previously stored continuation calls. We then follow a local scheduling approach [9] and execution *fails* for the new answer operation. New answers are only returned to the calling environment when all program clauses for the subgoal at hand were resolved.

Therefore, when the execution backtracks to a generator node, we first exhaust the remaining clauses and only then, when no more clauses are available, we start consuming the available answers for the subgoal. After consuming all answers, if the generator depends on older subgoals, we then proceed as for consumers and we fail by leaving the continuation call for the current computation in the table entry corresponding to the generator. It is only when a tabled subgoal is marked as complete that the corresponding continuation calls are deleted.

2.1 Program Transformation

Program transformation only applies to clauses of tabled predicates. Figure 1 shows how a `p/2` tabled predicate, that defines a right recursive relation in a graph, is transformed to implement our mechanism.

```
% original tabled predicate
p(X,Z) :- e(X,Y), p(Y,Z).
p(X,Z) :- e(X,Z).

% transformed predicate
p(X,Z) :- tabled_call(p(X,Z),Sid,_,p0,true), consume_answer(p(X,Z),Sid).
p0(p(X,Z),Sid) :- e(X,Y), tabled_call(p(Y,Z),Sid,[X,Z,Y],p0,p1).
p1(p(Y,Z),Sid,[X,Z,Y]) :- new_answer(p(X,Z),Sid).
p0(p(X,Z),Sid) :- e(X,Z), new_answer(p(X,Z),Sid).
```

Fig. 1. Program transformation for a tabled predicate

A `p/2` clause is maintained so that tabled predicate `p/2` can be called from other predicates without any change. The tabling primitive `tabled_call/5` in the body of `p/2` starts the tabled evaluation process and ensures that the subgoal will be completely evaluated. The `consume_answer/2` primitive then returns each computed answer one at a time.

Each clause in the original definition of `p/2` becomes a clause for a new distinct predicate, `p0/2` in the example, with 2 arguments. The first argument is the previous head clause; the second is the *subgoal id*. The id for a subgoal call is generated by the `tabled_call/5` primitive and is used to guarantee that the answers found for a call will be properly saved within that call. When the `tabled_call/5` primitive is called for a new subgoal (the first argument), a new entry is allocated in the table space and an unique id is returned (`Sid`). This id is then used by the `tabled_call/5` primitive to start the evaluation process by calling the predicate of arity 2 represented in the forth argument, `p0` in this case, with the appropriate arguments. To fully implement this idea, each tabled predicate in the body of the `p0/2` clauses is also replaced by a call to the `tabled_call/5` primitive. In addition, a `new_answer/2` primitive, responsible to check for redundant answers and to return new answers to the stored continuation calls, is added to the end of each clause's body.

A major issue with this transformation step is how to deal with continuation calls. Intuitively, the continuation call for a generator or consumer node is the code to be executed when an answer is returned to the node. With respect to a clause and an occurrence of a tabled predicate in the clause body, the continuation call is the portion of the clause body after the tabled predicate literal. The problem here is that to execute a continuation call, we cannot start a clause from the literal after the corresponding tabled predicate. Our program transformation captures the continuation of a tabled predicate in a clause body by introducing a new predicate symbol, which has a single clause and whose body is the continuation to be executed when an answer is returned. In the example of Fig. 1, we only have one such predicate, `p1/3`. The first two arguments of `p1/3` are the same as the arguments of `p0/2`. The third argument is a list of variables used to pass variable bindings across continuation calls.

Continuations calls are added by the `tabled_call/5` primitive when a computation is being suspended. A continuation call is a triplet formed by the second, third and fifth arguments passed to the `tabled_call/5`. The second argument stores the id of the call that is calling the `tabled_call/5` in the body of a clause. The third argument stores the bindings for the variables appearing in the head and in the body of the clause. The fifth argument stores the predicate of arity 3 to be called in the continuation. When a new answer is found, each continuation triplet is then used in conjunction with the new answer to construct the calls that continue the suspended computations.

Figure 2 shows the evaluation sequence for the query goal `p(1,Z)` if considering a small directed graph defined by two `e/2` facts, `e(1,2)` and `e(2,1)`. At the top, the figure illustrates the program code and the final state of the table space at the end of the evaluation. The bottom sub-figure shows the resulting forest of trees with the numbering of nodes denoting the evaluation sequence.

The evaluation begins by calling the `tabled_call/5` primitive for the `p(1,Z)` subgoal. The `p(1,Z)` subgoal is a new tabled subgoal call and thus, a new entry is allocated in the table space for it, with id `sid1`, and a new generator node is created (node 1). Generator nodes are represented by black oval boxes. In the continuation, `p0(p(1,Z),sid1)` is called, creating node 2. The execution then proceeds with the first alternative of `p0/2`, calling `e(1,Y)` that binds `Y` with `2` and with primitive `tabled_call/5` being called for the `p(2,Z)` subgoal (step 4). As this is the first call to `p(2,Z)`, we add a new entry for it, with id `sid2`, and proceed by allocating a new generator node as shown in the bottommost tree.

Again, `p(2,Z)` is resolved against the first clause for `p0/2` (step 5). In the continuation, the first clause for `e(2,Y)` fails (step 7), but the second succeeds by binding `Y` with `1` (step 8). The `tabled_call/5` primitive is then called again for `p(1,Z)`. Since `p(1,Z)` is a variant call and no answers are still available for it, the current evaluation is *suspended*. A triplet formed by the id `sid2`, the list `[2,Z,1]`, and by the predicate symbol `p1` is then stored in the table entry for `p(1,Z)` as a continuation call (step 9).

We then return to node 5 and try the second clause for `p0/2` obtaining, in the continuation, a first answer for `p(2,Z)` (step 12). The answer is inserted

```
p(X,Z) :- tabled_call(p(X,Z),Sid,_,p0,true), consume_answer(p(X,Z),Sid).

p0(p(X,Z),Sid) :- e(X,Y), tabled_call(p(Y,Z),Sid,[X,Z,Y],p0,p1).
p1(p(Y,Z),Sid,[X,Z,Y]) :- new_answer(p(X,Z),Sid).
p0(p(X,Z),Sid) :- e(X,Z), new_answer(p(X,Z),Sid).

e(1,2).   e(2,1).
```

Sid	Subgoal	Answers	Continuation calls
sid1	1. p(1,Z)	15. p(1,1) 23. p(1,2) 32. complete	9. p1(?ANS?,sid2,[2,Z,1])
sid2	4. p(2,Z)	12. p(2,1) 25. p(2,2) 32. complete	20. p1(?ANS?,sid1,[1,Z,2])

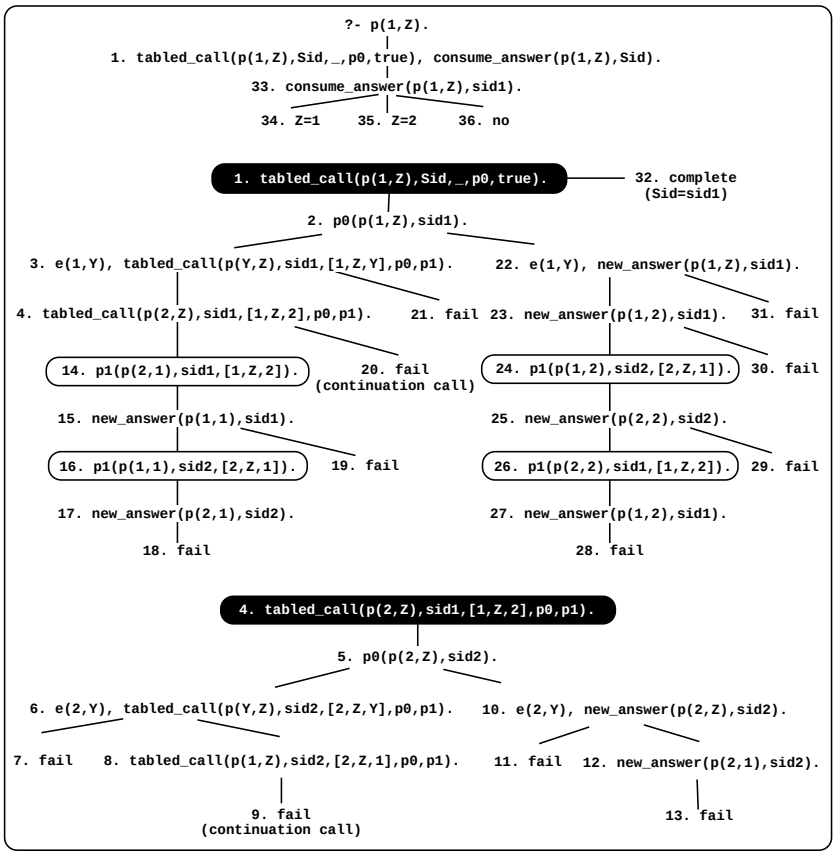

Fig. 2. Tabled evaluation with continuation calls

in the table and, because there are no continuation calls for `p(2,Z)`, the execution fails. Remember that the new answer operation always fails when using a local scheduling approach. The execution then backtracks to node 4 and we check whether `p(2,Z)` can be completed. It can not, because it depends on the continuation call left by subgoal `p(1,Z)` at step 9. If that continuation call gets executed, further answers for `p(2,Z)` can be found.

At that point, the answers in the table entry for `p(2,Z)` should be consumed. Remember that answers are only returned to a generator node when all program

clauses for it were resolved. Consuming an answer corresponds to the execution of the continuation call for the `tabled_call/5` at hand. The continuation call for the `tabled_call/5` primitive at node 4 is thus executed for the answer `p(2,1)` (step 14). Continuation calls are represented by white oval boxes.

A first answer, `p(1,1)`, is then found for `p(1,Z)` (step 15) and, because there is a continuation call for `p(1,Z)`, the execution continues by calling it with the newly found answer (step 16). A redundant answer is then found for `p(2,Z)` (step 17), so we fail and backtrack again to node 4. We then proceed as for consumers and we fail by leaving the continuation call for the current computation in the table entry for `p(2,Z)` (step 20). The evaluation then explores the second clause at node 2 obtaining, in the continuation, a second answer for `p(1,Z)` (step 23) and a second answer for `p(2,Z)` (step 25). When backtracking to node 1, the subgoals `p(1,Z)` and `p(2,Z)` are now fully exploited. So, we declare the two subgoals to be completed (step 32). Again, at that point, the answers in the table entry should be consumed. However, the continuation call for the `tabled_call/5` primitive at node 1 is the predicate symbol `true`. This means that the `tabled_call/5` was executed from the clause representing the original `p/2` predicate. In such situations, we simply succeed by binding `Sid` with the current subgoal id, `sid1` in this case, and leave to the `consume_answer/2` primitive the process of returning by backtracking the computed answers.

2.2 Implementation Details

A key data structure in the table space organization is the *subgoal frame*. Subgoal frames are used to store information about the tabled subgoals and to act like entry points to the trie structures where answers are stored. In our implementation, a subgoal frame includes eight fields with the following meaning:

`SgFr_state:` indicates the state of the subgoal. During evaluation, a subgoal can be in one of the following states: *ready*, *evaluating* or *complete*.

`SgFr_dfn:` is the *depth-first number* of the subgoal call. Calls are numbered incrementally and according to the order in which they appear in the evaluation.

`SgFr_dep:` is the *depth-first number* of the older call in which the current call depends. It is initialized with the same number as its depth-first number, meaning that no dependencies exist. It is critical to the fix-point check procedure that we discuss later.

`SgFr_answers:` is the entry point to the answer trie structure.

`SgFr_first_answer:` is the pointer to the first inserted answer in the answer trie or `NULL` if no answers are available.

`SgFr_last_answer:` is the pointer to the last inserted answer in the answer trie or `NULL` if no answers are available.

`SgFr_cont_calls:` is the pointer to continuation calls associated with the subgoal or `NULL` if no continuation calls are available.

`SgFr_previous:` is the pointer to the previously allocated subgoal frame that is still not completed. A global variable, `TOP_SF`, always points to the youngest subgoal frame being evaluated.

Next we show the pseudo-code for the `new_answer/2` and `tabled_call/5` primitives. The `new_answer/2` primitive (see Fig. 3) starts by calling a procedure to insert the given `ANSWER` in the answer trie structure for the `SF` subgoal frame. If the answer is redundant, it simply fails. Otherwise, the answer is new and each continuation triplet stored in `SF` is then used in conjunction with the new answer to construct the calls that continue the suspended computations.

```
new_answer(YAP_Term ANSWER, SgFr SF) {
  if (insert_answer(ANSWER, SF) == TRUE)
    for each (cont_pred, cont_sf, cont_vars) in SgFr_cont_calls(SF) do {
      cont_call = construct_call(cont_pred, ANSWER, cont_sf, cont_vars);
      YAP_CallProlog(cont_call);
    }
  return FALSE;                                    // always fail at the end
}
```

Fig. 3. Pseudo-code for the `new_answer/2` primitive

The `tabled_call/5` primitive (see Fig. 4) starts by calling a procedure to insert the given `SUBGOAL_CALL` in the table space. Then, it checks if the resulting subgoal frame is new, that is, if the `SgFr_state` is `READY`. Being this the case, it changes the subgoal's state to `EVALUATING`, constructs the call that starts the evaluation process and calls the Prolog engine to execute it. When returning, it checks for completion. If the `SgFr_dfn` and `SgFr_dep` fields are equal then we know that no dependencies exist and, therefore, all younger subgoals can be completed. Otherwise, if the subgoal is not new or if the `SgFr_dfn` and `SgFr_dep` fields are different, we propagate the dependency in the `SgFr_dep` field of the current subgoal to the `SgFr_dep` field of the `CONT_SF` frame that continues the execution. We discuss completion in more detail in the next subsection.

In both situations, we then consume the available answers, leave a continuation call if the subgoal is not completed, and fail. The only situation where this primitive does not fails is when it is executed from the clause representing the original tabled predicate, that is, where the `CONT_PRED` is `true`. In such situations, the procedure succeeds by unifying the `CONT_SF` argument with the current subgoal frame pointer.

2.3 Detecting Completion

When checking for completion, we use the `SgFr_dfn` and the `SgFr_dep` fields of the subgoal frames to quickly determine whether a subgoal is a *leader node*, that is, whether it does not depend on older generators. If the `SgFr_dfn` and `SgFr_dep` fields are equal then we know that no dependencies exist and thus the subgoals starting from the frame referred by `TOP_SF` up to the current subgoal can be completed. On the other hand, if the `SgFr_dep` field holds a number less than its depth-first number then we cannot perform completion. Instead, we must propagate the current dependency to the subgoal call that continues the evaluation. Figures 5 and 6 show two examples illustrating how completion is detected. At the top, each figure shows the subgoal dependencies and the leader nodes (nodes filled with a black background). The black dots in the sub-figures below indicate the fields being updated at each step of the example.

```
tabled_call(YAP_Term SUBGOAL_CALL, SgFr CONT_SF, YAP_Term CONT_VARS,
            YAP_Atom INITIAL_PRED, YAP_Atom CONT_PRED) {
  sf = insert_tabled_call(SUBGOAL_CALL);
  if (SgFr_state(sf) == READY) {                        // new subgoal call
    SgFr_state(sf) = EVALUATING;
    initial_call = construct_call(INITIAL_PRED, SUBGOAL_CALL, sf);
    YAP_CallProlog(initial_call);
    if (SgFr_dfn(sf) == SgFr_dep(sf))              // check for completion
      do {
        SgFr_state(TOP_SF) = COMPLETE;
        delete_continuation_calls(TOP_SF);
        TOP_SF = SgFr_previous(TOP_SF);
      } while (TOP_SF != SgFr_previous(sf))   // complete frames up to sf
  }
  if (SgFr_state(sf) != COMPLETE)    // propagate dependencies to CONT_SF
    SgFr_dep(CONT_SF) = minimum(SgFr_dep(CONT_SF), SgFr_dep(sf));
  else if (SgFr_state(sf) == COMPLETE && CONT_PRED == true) {  // succeed
    YAP_Unify(CONT_SF, sf);    // we use the sf pointer as the subgoal id
    return TRUE;
  }
  answer = load_answer(SgFr_first_answer(sf));        // get first answer
  while (answer != NULL) {
    cont_call = construct_call(CONT_PRED, answer, CONT_SF, CONT_VARS);
    YAP_CallProlog(cont_call);
    answer = load_answer(AnswerTrie_next(answer));     // get next answer
  }
  if (SgFr_state(sf) != COMPLETE)            // leave a continuation call
    add_continuation_call(sf, CONT_PRED, CONT_SF, CONT_VARS);
  return FALSE;                                     // always fail at the end
}
```

Fig. 4. Pseudo-code for the `tabled_call/5` primitive

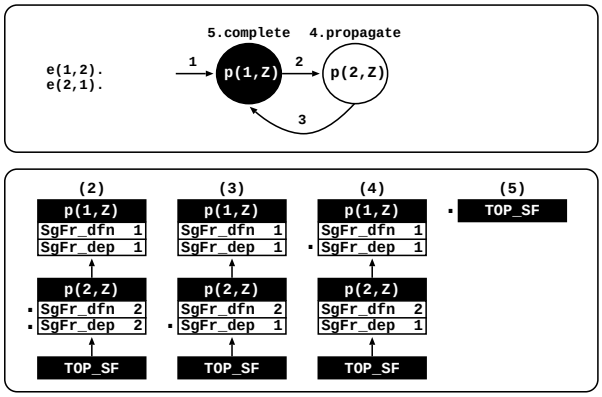

Fig. 5. Completion in the evaluation of Fig. 2

Figure 5 represents again the evaluation of Fig. 2. Initially, `p(1,Z)` and `p(2,Z)` are called and two subgoal frames are initialized respectively with fields `SgFr_dfn` and `SgFr_dep` as 1 and 2 (step 2 in Fig. 5). In the continuation, `p(2,Z)` calls again `p(1,Z)`, which creates a dependency between subgoals `p(2,Z)` and `p(1,Z)`. The `SgFr_dep` field of `p(2,Z)` is updated to represent this dependency to `p(1,Z)` (step 3 in Fig. 5). Next, when the computation returns to `p(2,Z)`, we cannot perform completion because the `SgFr_dfn` and `SgFr_dep` fields are different. We thus propagate the dependency in the `SgFr_dep` field of `p(2,Z)` to `p(1,Z)` (step 4 in Fig. 5). This has no effect because the `SgFr_dep` field of `p(1,Z)` is already 1. At the end, when the computation returns to `p(1,Z)`, the two subgoals are marked as complete and the `TOP_SF` global variable is updated (step 5 in Fig. 5).

Figure 6 represents a different example for a graph with more edges (see the e/2 facts in the sub-figure at the top), where subgoals are not necessarily completed at the end of the evaluation. As in the previous example, the computation starts with subgoals p(1,Z) and p(2,Z). Next, p(3,Z) and p(4,Z) are also called and two new frames are initialized respectively with SgFr_dfn and SgFr_dep as 3 and 4. In the continuation, p(4,Z) calls p(2,Z) again, and the SgFr_dep field of subgoal p(4,Z) is updated to represent this dependency (step 5 in Fig. 6). Then, the computation returns to p(4,Z) and we propagate the dependency in the field SgFr_dep of p(4,Z) to p(3,Z) (step 6 in Fig. 6). The same happens when the computation returns to p(3,Z) and we propagate its dependency to p(2,Z) (step 7 in Fig. 6). The interesting case occurs when the computation returns to p(2,Z) as the SgFr_dfn and SgFr_dep fields are equal meaning that p(2,Z) is the leader of the subgoals starting from TOP_SF up to it, that is, subgoals p(4,Z), p(3,Z) and p(2,Z). The three subgoals are thus marked as complete and the TOP_SF variable is updated to the previous subgoal in evaluation (step 8 in Fig. 6). At the end, the computation returns to p(1,Z) and its subgoal frame is also marked as completed as no more dependencies exist.

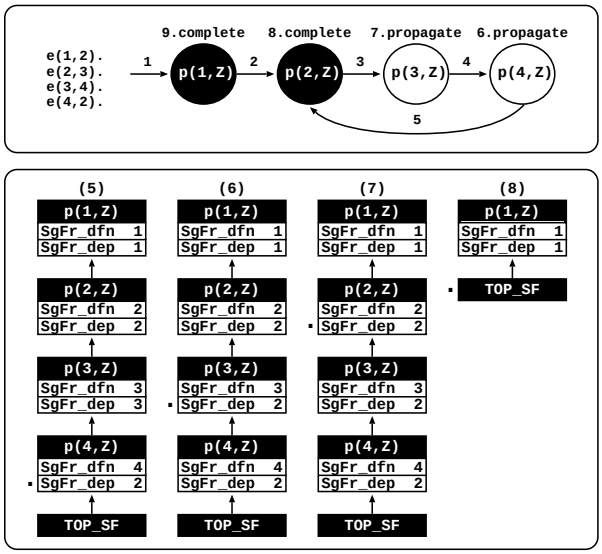

Fig. 6. Completion in the middle of the evaluation

3 Experimental Results

To evaluate the impact of our approach, we ran it against the YapTab system using six different versions of the p/2 predicate combined with several different configurations of the e/2 facts, for a total number of 54 programs. The six versions of the p/2 predicate include two right recursive, two left recursive and two doubly recursive p/2 definitions. Each pair has one definition with the recursive clause first and another with the recursive clause last. Regarding the e/2 facts, we used three configurations: a binary tree, a cycle and a grid configuration (Fig. 7 shows an example for each configuration). We have experimented the binary tree configuration with depths 12, 14 and 16; the cycle configuration with depths 200, 300 and 400; and the grid configuration with depths 10x10, 15x15 and 20x20. The environment for our experiments was a Pentium M 1600MHz processor with 768 MBytes of main memory and running the Linux kernel 2.6.11.

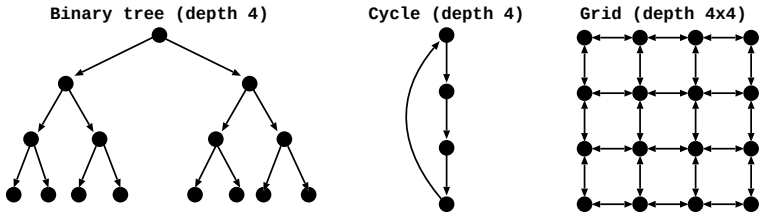

Fig. 7. The `e/2` configurations

Table 1 shows the overhead of the continuation calls running times over the YapTab running times using local scheduling. Each result corresponds to the average overhead obtained in a set of 3 runs. Each run finds all the solutions for the problem. All experiments were performed using the YapTab system based on the Yap Prolog 5.1.1.

Predicate	Binary tree			Cycle			Grid		
	12	**14**	**16**	**200**	**300**	**400**	**10x10**	**15x15**	**20x20**
`p_right_first/2`	4.00	3.73	3.62	4.36	3.99	3.89	7.75	6.41	6.11
`p_right_last/2`	3.73	3.59	3.70	4.56	4.00	3.98	8.55	6.27	6.42
`p_left_first/2`	2.65	2.39	2.34	3.05	2.65	2.26	3.11	2.46	2.12
`p_left_last/2`	5.00	4.31	4.25	5.13	4.34	4.24	5.67	4.73	4.15
`p_doubly_first/2`	8.13	7.72	7.68	10.45	11.57	11.22	10.34	9.66	10.40
`p_doubly_last/2`	15.05	13.96	13.68	20.36	22.23	21.72	19.74	18.25	19.53

Table 1. Overheads over the YapTab running times

The results in Table 1 clearly show that the YapTab tabling engine outperforms the tabled evaluation with continuation calls mechanism in all configurations. However, considering that Yap and YapTab are respectively two of the fastest Prolog and tabling engines currently available, the results obtained with our approach are very interesting and very promising results. A closer analysis of the results indicate that our approach scales well when we increase the complexity of the problem being tested. In general, the overhead over YapTab is almost the same when we compare the binary tree configurations (depths 12, 14 and 16), the cycle configurations (depths 200, 300 and 400) or the grid configurations (depths 10x10, 15x15 and 20x02) between themselves. In particular, for some configurations, the overhead over YapTab shows a generic tendency to decrease as the complexity of the problem increases. The results also show that the right recursive definitions achieve similar overheads when we use the versions with the recursive clause first or last. For the left and doubly recursive definitions, the versions with the recursive clause first obtain better results, about half the overhead of the corresponding last versions. Globally, best performance is always achieved by the left recursive definition with the recursive clause first, `p_left_first/2`, with an excellent average overhead between 2 and 3.

To better understand these results, we next present in Table 2 several statistics gathered during execution for three specific configurations: the binary tree with depth 16, the cycle with depth 400 and the grid with depth 20x20. The rows in Table 2 have the following meaning:

AnsUni: is the number of non-redundant answers found. It corresponds to the total number of answers stored in the table space.
AnsRed: is the number of redundant answers found.
CallsUni: is the number of first calls to tabled subgoals. It corresponds to the total number of subgoal frames allocated.
CallsRep: is the number of repeated calls to tabled subgoals.
ContCalls: is the number of continuation calls executed by the primitives `tabled_call/5` and `new_answer/2`.
OvHd: is the overhead over YapTab from Table 1.

Predicate	AnsUni	AnsRed	CallsUni	CallsRed	ContCalls	OvHe
Binary tree 16						
p_right_first/2	1,769,478	0	65,535	65,532	1,638,412	3.62
p_right_last/2	1,769,478	0	65,535	65,532	1,638,412	3.70
p_left_first/2	917,506	0	1	1	917,506	2.34
p_left_last/2	917,506	786,440	1	1	1,769,478	4.25
p_doubly_first/2	1,769,478	9,568,232	65,535	1,769,479	12,976,122	7.68
p_doubly_last/2	1,769,478	19,136,464	65,535	3,407,891	24,182,766	13.68
Cycle 400						
p_right_first/2	320,000	800	401	400	320,000	3.89
p_right_last/2	320,000	800	401	400	320,000	3.98
p_left_first/2	160,000	400	1	1	160,000	2.26
p_left_last/2	160,000	160,000	1	1	319,600	4.24
p_doubly_first/2	320,000	127,680,519	401	320,001	128,320,000	11.22
p_doubly_last/2	320,000	255,358,774	401	639,200	256,319,200	21.72
Grid 20x20						
p_right_first/2	320,000	899,040	401	2,640	1,216,000	6.11
p_right_last/2	320,000	899,040	401	2,640	1,216,000	6.42
p_left_first/2	160,000	449,520	1	1	160,000	2.12
p_left_last/2	160,000	1,051,672	1	1	318,480	4.15
p_doubly_first/2	320,000	127,683,040	401	320,001	128,320,000	10.40
p_doubly_last/2	320,000	254,467,040	401	636,961	255,420,960	19.53

Table 2. Statistics gathered during execution for three specific configurations

Globally, the statistics presented in Table 2 suggest that there is a cost in the execution time that is proportional to the number of redundant answers, repeated calls and continuation calls executed during an evaluation. In particular, the number of continuation calls seems to be the most relevant factor that contributes to this cost. Remember that continuation calls are not compiled, they are constructed and called in run-time using the C language interface.

The statistics also show why the right recursive definitions obtain similar overheads for the versions with the recursive clause first and last. In the three configurations tested, they show exactly the same statistics. For the left and doubly recursive definitions, the number of redundant answers, repeated calls and continuations calls in the versions with the recursive clause first is about half the number of the statistics for the last versions, which justifies why they have, on average, half the overhead of the corresponding last versions. The statistics for the left recursive definitions with the recursive clause first also show the smallest number of redundant answers, repeated calls and continuations calls.

4 Conclusions

We have presented the design, implementation and evaluation of a suspension-based tabling mechanism based on program transformation coupled with tabling primitives. Initial results comparing the state-of-the-art YapTab system with our approach were very interesting and very promising. We thus argue that our approach is a good alternative to incorporate tabling into any Prolog system. Both source level transformations and tabling primitives can be easily ported to other Prolog systems with a C language interface. Currently, we are already working with the Ciao group to include our implementation as a module of the Ciao Prolog system [12]. Further work also includes extending our approach to support linear tabling mechanisms, such as the DRA [4] and SLDT [5] mechanisms.

References

1. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43** (1996) 20–74
2. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems **20** (1998) 586–634
3. Rocha, R., Silva, F., Santos Costa, V.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction. (2000) 77–87
4. Guo, H.F., Gupta, G.: 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, Springer-Verlag (2001) 181–196
5. Zhou, N.F., Shen, Y.D., Yuan, L.Y., You, J.H.: Implementation of a Linear Tabling Mechanism. In: Practical Aspects of Declarative Languages. Number 1753 in LNCS, Springer-Verlag (2000) 109–123
6. Somogyi, Z., Sagonas, K.: Tabling in Mercury: Design and Implementation. In: International Symposium on Practical Aspects of Declarative Languages. Number 3819 in LNCS, Springer-Verlag (2006) 150–167
7. Fan, C., Dietrich, S.: Extension Table Built-Ins for Prolog. Software Practice and Experience **22** (1992) 573–597
8. Santos Costa, V., Damas, L., Reis, R., Azevedo, R.: (YAP User's Manual) Available from `http://www.ncc.up.pt/~vsc/Yap`.
9. Freire, J., Swift, T., Warren, D.S.: 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, Springer-Verlag (1996) 243–258
10. Ramakrishnan, I.V., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient Access Mechanisms for Tabled Logic Programs. Journal of Logic Programming **38** (1999) 31–54
11. Ramesh, R., Chen, W.: Implementation of Tabled Evaluation with Delaying in Prolog. IEEE Transactions on Knowledge and Data Engineering **9** (1997) 559–574
12. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López, P., Puebla, G.: (Ciao Prolog System Manual) Available from `http://clip.dia.fi.upm.es/Software/Ciao`.