Towards a Complete Scheme for Tabled Execution Based on Program Transformation*

Pablo Chico de Guzman[1] Manuel Carro[1] Manuel V. Hermenegildo[1,2]
pchico@clip.dia.fi.upm.es {mcarro,herme}@fi.upm.es

[1] School of Computer Science, Univ. Politécnica de Madrid, Spain
[2] IMDEA Software, Spain

Abstract. The advantages of tabled evaluation regarding program termination and reduction of complexity are well known —as are the significant implementation, portability, and maintenance efforts that some proposals (especially those based on suspension) require. This implementation effort is reduced by program transformation-based continuation call techniques, at some efficiency cost. However, the traditional formulation of this proposal [1] limits the interleaving of tabled and non-tabled predicates and thus cannot be used as-is for arbitrary programs. In this paper we present a complete translation for the continuation call technique which, while requiring the same runtime support as the traditional approach, solves these problems and makes it possible to execute arbitrary tabled programs. We also present performance results which show that the resulting `CCall` approach offers a useful tradeoff that can be competitive with other state-of-the-art implementations.
Keywords: Tabled logic programming, Continuation-call tabling, Implementation, Performance, Program transformation.

1 Introduction

Tabling [2–4] is a strategy for executing logic programs which uses *memoization* of already processed calls and their answers to improve several of the limitations of SLD resolution. It brings termination for bounded term-size programs and improves efficiency in programs which perform repeated computations. It has been successfully applied to deductive databases [5], program analysis [6, 7], reasoning in the semantic Web [8], model checking [9], etc.

However, tabling also has certain drawbacks, including that predicates to be tabled have to be carefully selected[3] in order not to incur in undesired slowdowns and, specially relevant to our discussion, that its efficient implementation is generally complex. In *suspension-based tabling* the computation state of suspended tabled subgoals has to be preserved to avoid backtracking over them. This is done either by *freezing* the stacks, as in the SLG-WAM [10], by copying to another area, as in CAT [11], or by using an intermediate solution as in CHAT [12]. *Linear tabling* maintains instead a single execution tree without

* This work was funded in part by EU FET project IST-15905 *MOBIUS*, and FP7 grant agreement 215483 *S-Cube*, Spanish MEC project TIN2008-05624 DOVES, ITEA2/PROFIT FIT-340005-2007-14 *ES_PASS*, and by Madrid Regional Government program S-0505/TIC/0407 *PROMESAS*.

[3] Note that XSB includes an `auto_table` declaration to automatically select which predicates are to be tabled in order to ensure termination. This declaration triggers a conservative analysis which may mark more predicates than strictly needed.

requiring suspension and resumption of sub-computations. The computation of the (local) fixpoint is performed by making subgoals "loop" in their alternatives until no more solutions are found. This may force some computations to be repeated. Examples of this method are the linear tabling of B-Prolog [13, 14] and the DRA scheme [15]. Suspension-based mechanisms achieve very good performance but, in general, require deeper changes to the underlying implementation. Linear mechanisms, on the other hand, can usually be implemented on top of existing sequential engines without major modifications.

The Continuation Call (`CCall`) approach to tabling [1] tries to combine the best of both worlds: it is a suspension-based mechanism (and, therefore, it does not need recomputation) which requires relatively simple additions to the Prolog implementation / compiler,[4] thus making maintenance and porting much easier. In [16] we proposed a number of optimizations to the `CCall` approach and showed that with such optimizations performance could be competitive with traditional implementations. However, this was only partially satisfactory since the `CCall` tabling approach is restricted to programs with no interleaving of tabled and non-tabled predicate calls, and thus cannot execute general tabled programs.

In this paper we present an extension of the `CCall` translation which, while requiring the same runtime support of the traditional proposal, overcomes the problem pointed out above. We also present a complexity comparison with CHAT and performance results comparing with state-of-the-art implementations.

2 The Continuation Call Technique

We sketch now how tabled evaluation [4, 10] works from a user point of view and we briefly describe the Continuation Call technique, on which we base our work.

2.1 Tabling Basics

We will use as example the program in Figure 1, whose purpose is to determine the reachability of nodes in a graph. Since the graph contains a cycle, the query `path(1,Z)` will make the program loop forever under the standard SLD resolution strategy, regardless of the order of the clauses. In this case, tabling changes the operational semantics of the `path/2` predicate to distinguish the first occurrence of a `path/2` goal (the *generator*) and subsequent calls which are identical up to variable renaming (the *consumers*). The generator applies resolution using the program clauses to derive answers for the goal. The consumer (the first recursive call in our example) *suspends* the current execution path (using implementation-dependent means) and starts execution on the second clause of predicate `path/2`. When this branch finally succeeds, the answer generated for the initial query, `path(1,1)`, is inserted in the table entry associated with its generator. This makes it possible to reactivate the consumer and to continue execution at the point where it was stopped. Thus, consumers do not use SLD resolution, but obtain instead the answers from the table where they were previously inserted by the generator. Predicates not marked as tabled are executed according

[4] As an example, no modification to the underlying engine is needed.

```
:- table path/2.

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

edge(1,1).
```

Fig. 1. A sample program.

```
path(X, Y):- slg(path(X, Y)).
slg_path(path(X, Z), Id):-
    edge(X, Y),
     slgcall (path_cont(Id, [X], path(Y, Z))).
slg_path(path(X, Z), Id):-
    edge(X, Z),
    answer(Id, path(X, Z)).
path_cont(Id, [X], path(Y, Z)):-
    answer(Id, path(X, Z)).
```

Fig. 2. The program in Figure 1 after being transformed for tabled execution.

to SLD resolution, hopefully with minimal overhead due to the availability of tabling.

2.2 `CCall` by Example

`CCall` implements tabling by a combination of program transformation and side effects in the form of insertions into and retrievals from a table which relates calls, answers, and the continuation code to be executed after consumers read answers from the table. We will now sketch how the mechanism works using the path/2 example (Figure 1). The original code is transformed into the program in Figure 2, whose execution is shown in Figure 3.

Roughly speaking, the transformation for tabling is as follows: the predicate to be actually tabled is a variation (slg_path/2) of the initial predicate (path/2). In order to preserve the previous interface, path/2 calls slg_path/2 through a primitive, slg/1, which keeps track of which invocation is a generator or a consumer and makes sure that its argument is executed to completion. After completion, it will return, on backtracking, all the solutions found for the tabled predicate. To this end, slg/1 checks if the call has already been executed. If so, all of its answers are returned on backtracking. Otherwise, slg/1 passes control to the transformed version of its argument, slg_path/2 (step 2).[5] slg_path/2 receives in its first argument the original call to path/2 and in the second argument the identifier of its generator, which is used to relate operations on the table with this initial call. Each clause of slg_path/2 is derived from a clause of the original path/2 predicate by:

- Adding an answer/2 primitive at the end of each clause of the original tabled predicate. answer/2 inserts answers in the entry of the table identified by its first argument (step 7) after checking for redundant answers (i.e., step 10 does not insert the redundant answer) and fails.
- Instrumenting calls to tabled predicates using the slgcall/1 primitive (step 4). If this tabled call is a consumer, path_cont/3, along with its arguments, is recorded as (one of) the continuation(s) of its generator and execution suspends (step 5). If the tabled call is a generator, it is associated with

[5] A unique name has been created by simply prepending slg_ to the original predicate name. Any means of constructing a unique predicate name can be used.

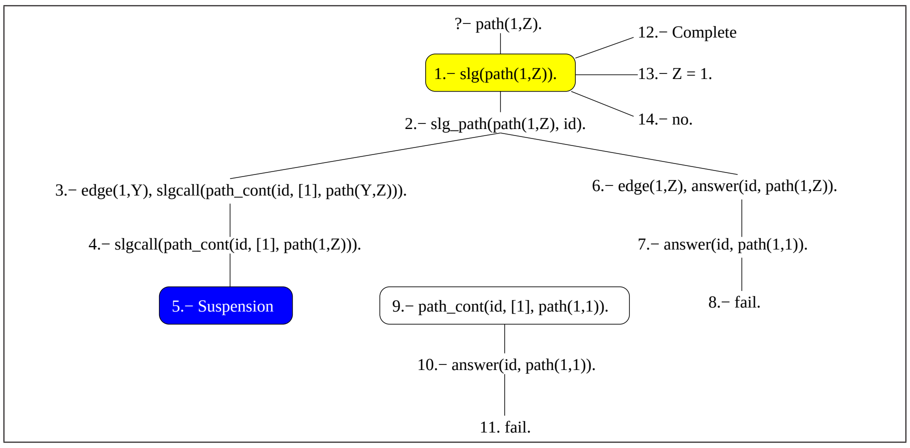
?– path(1,Z).
1.– slg(path(1,Z)).
12.– Complete
13.– Z = 1.
14.– no.
2.– slg_path(path(1,Z), id).
3.– edge(1,Y), slgcall(path_cont(id, [1], path(Y,Z))).
4.– slgcall(path_cont(id, [1], path(1,Z))).
5.– Suspension
6.– edge(1,Z), answer(id, path(1,Z)).
7.– answer(id, path(1,1)).
8.– fail.
9.– path_cont(id, [1], path(1,1)).
10.– answer(id, path(1,1)).
11. fail.

Fig. 3. Tabling execution of Figure 1.

a new call identifier and execution follows using slg_path/2 to derive new answers (as done by slg/1 (step 1)). Besides, path_cont/3 will be recorded as a continuation of the generator identified by Id if the tabled call cannot be completed (because there may be dependencies on previous generators). The path_cont/3 continuation will be called to consume found answers (step 9) or erased upon completion of its generator.

– Encoding the remaining of the clause body of path/2 after the recursive call using path_cont/3. This is constructed in a similar way to slg_path/2, i.e., applying the same transformation as for the initial clauses and calling slgcall/1 if this clause contains another call to the tabled predicate.

The second argument of path_cont/3 is a list of bindings needed to recover the environment of the continuation call; in other words, the variables which are reachable before a consumer is suspended and which can be necessary when the consumer is resumed. In our example, when the execution suspends (step 5), the value of X has to be saved since it will be used by the answer/2 primitive when the consumer is resumed (step 10).

A safe approximation of the variables which should appear in this list is the set of variables which appear in the clause before the tabled goal and which are used in the continuation, including the answer/2 primitive. Variables appearing in the tabled call itself do not need to be included, as they will be passed along anyway.

Key Contribution of `CCall`: a new predicate name is created for all points where suspension can happen. Suspension is performed by saving this predicate name (equivalent to saving a program counter), a list of bindings (equivalent to protecting the environment from backtracking), and a generator identifier (to relate answers in the table with the generator). Resumption is performed by constructing a Prolog goal with the information saved on suspension plus the answer which raised the resumption. This mechanism is significantly simpler to implement than other approaches such as SLG-WAM or CHAT, where non-trivial

```
:- table t/1.

t(A):-
    p(B),
    A is B + 1.
t(0).

p(B):- t(B), B < 1.
```

Fig. 4. A program for which the original `CCall` transformation fails.

```
t(A):- slg(t(A)).
slg_t(t(A), Id):-
    p(B), A is B + 1,
    answer(Id, t(A)).

slg_t(t(0), Id):-
    answer(Id, t(0)).

p(B):- t(B), B < 1.
```

Fig. 5. The program in Figure 4 after being wrongly transformed for tabled execution.

extensions to the SLD abstract machine had to be introduced. Consequently, porting and maintainability are simpler too, since `CCall` is independent of the compiler. Creating a Prolog term on the heap is the only low-level operation to be implemented.

3 Mixing Tabled and Non-Tabled Predicates

The `CCall` approach to tabling, as originally proposed, has a serious limitation which shows up when non-tabled predicates appear between a generator and its consumers: the variables created during the execution of these non-tabled predicates may be needed to correctly suspend and resume consumers. However, `CCall` just saves the environment of the parent call.

3.1 Problems in the Original Transformation

As an example of the problem, Figure 4 shows a tabled program where tabled and non-tabled execution (t/1 and p/1) are mixed. The translation of the program is shown in Figure 5, following the rules in Section 2.2.

The execution of the program for query t(A) is shown in Figure 6. Execution proceeds correctly until slg/1 is called again from p/1. At that point, execution should suspend (and later resume), but slg/1 does not have any associated continuation, and it does not have any pointer to the code to be executed on resumption (partially in p/1 and partially in slg_t/2): B < 1, A **is** B + 1, answer(Id,t(A)) is lost on backtracking and it is not reachable when resuming. Consequently, the second answer to the query, t(1), is lost.

The call to t(B) made by p(B) could have been translated using the slgcall/1 primitive, generating a continuation for the remaining code of p/1, but, even in that case, the code segment "A **is** B + 1, answer(Id, t(A))" in the first clause of slg_t/1 would be lost anyway. This is an example of why all the frames between a consumer and its nearest generator have to be saved when suspending, and it is not enough to save just the last one, as in the original `CCall` proposal.[6]

[6] Which does work, however, when all the calls to the tabled predicates appear in the body of a clause of a tabled predicate.

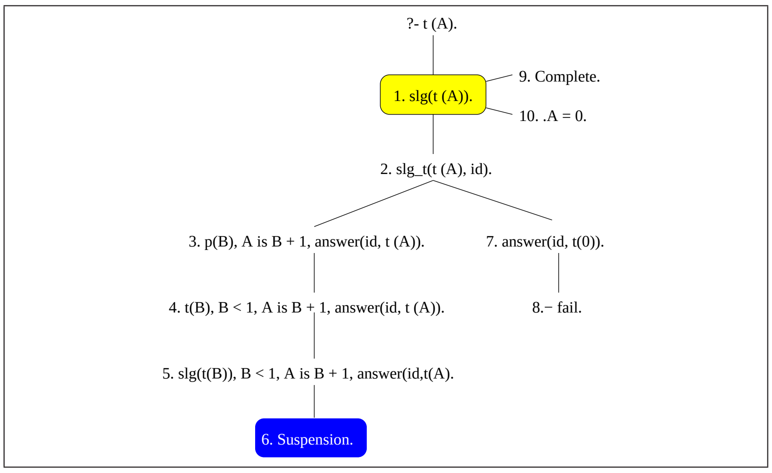
?- t (A).
1. slg(t (A)).
9. Complete.
10. .A = 0.
2. slg_t(t (A), id).
3. p(B), A is B + 1, answer(id, t (A)).
7. answer(id, t(0)).
4. t(B), B < 1, A is B + 1, answer(id, t (A)).
8.− fail.
5. slg(t(B)), B < 1, A is B + 1, answer(id,t(A).
6. Suspension.

Fig. 6. Tabling execution of example of Figure 1.

3.2 Marking Predicates as Bridges

To solve this problem, we have extended the translation to take into account a new kind of predicates, named *bridges*. A bridge predicate is a non-tabled predicate whose clauses generate frames which have to be saved in the continuation of a consumer. In the example of Figure 4, `p/1` would be a bridge predicate.

Bridge predicates are all the non-tabled predicates which can appear in the execution tree of a query between a generator and each of its consumers, i.e., the predicates whose environments lie in the local stack between the environment of the generator and that of each of its consumers. Note that tabled predicates do not need to be included as bridge predicates as their environment will be saved already by the translation.

Thus, in order to determine a minimal set of bridge predicates, B_{min}, we need to locate the points where a consumer will appear. Detecting that a call will definitely be a consumer is an undecidable problem (because it would need identifying where infinite failures happen). Therefore, generating B_{min} is also undecidable and a *safe approximation*, which may mark as bridges some predicates which do not need to be marked, has to be applied.

As we will see in Section 4, the disadvantages of such an over-approximation are minor. Bridge predicates (with an extra argument) can be called when not needed, incurring a small overhead, and some code may be duplicated (to accept a new argument for the case where a bridge predicate is called from a tabled execution). The algorithm we have implemented (Figure 7) only detects tabled predicates which can recursively call themselves. For the examples used for performance evaluation in Section 6, using the safe approximation algorithm produces, on average, a slowdown of only 3% with respect to a perfect (manual) characterization of bridge predicates.

```
Make a graph $G$ with an edge $(p1/n1,\ p2/n2) \Leftrightarrow p2/n2$ is called from $p1/n1$
$Bridges = \emptyset$
FOR each predicate $T$ in TABLED PREDICATES
  $Forward =$ All predicates reached from $T$ in $G$
  $Backward =$ All predicates from which $T$ is reached in $G$
  $Bridges = Bridges \cup (Forward \cap Backward)$
$Bridges = Bridges -$ TABLED PREDICATES
```

Fig. 7. Safe approximation to mark bridge predicates.

4 A General Translation for Tabled Programs

In this section we present program transformation rules which take into account bridge predicates. This transformation assumes that all the bridge predicates (and possibly some more) have been marked by adding :− bridge P/N declarations in the program.

As seen in Section 2.2, a continuation saves all the information needed to resume a consumer, including environment variables and continuation code. Consequently, the goal of the new translation is to associate a continuation with each of the bridge calls within the scope of tabled execution (Figure 9). Continuations for tabling will have a new argument (the continuation to be executed) and new continuations are pushed onto this argument as they appear, in much the same way as environments are pushed onto the local stack.

4.1 Translation Rules

The new translation rules are shown in the metaprogram in Figure 8, where we have used a sugared Prolog-like language. We use for conciseness functional syntax where needed [17]. Infix '$\circ$' is a generic concatenation function which joins either atoms or (linear) structures. It may appear in an output head position with the expected semantics.

The trans/2 predicate receives in its first argument the program clauses one by one and returns in its second argument a list of clauses resulting from the translation of the input clause. The first clause of trans/2 ensures that predicates which are neither tabled nor bridges are not transformed.[7] The second one generates, for each tabled predicate, a single-clause predicate to maintain the interface of the new predicate with the rest of the code (i.e., the first predicate in Figure 9, left). The third clause of trans/2 translates clauses of tabled predicates, and the fourth one translates clauses of bridge predicates, keeping the original clauses as well so that they can be called from non-tabled predicates.

A new predicate head (Head_tr) is generated, and its body will result from transforming the body literals appearing after a call to a tabled or a bridge predicate. The variable End holds the code to appear as last goal of the body corresponding to Head_tr. This code can be answer/2, for clauses of tabled predicates, or **call**(Cont), for clauses of bridge predicates. The latter will be used to

[7] Predicates table/1 and bridge/1 (generated by the compiler from the corresponding declarations) are used to check if their argument is a predicate head or a clause of a tabled or bridge predicate, respectively.

```
trans(C, C) :- \+ table(C), \+ bridge(C).
trans(( :- table P/N ), ( P(X1..Xn) :- slg(P(X1..Xn)) )).
trans(( Head :- Body ), LC) :-
    table(Head),
    Head_tr =.. ['slg_' ∘ Head, Head, Id],
    End = answer(Id, Head),
    transBody(Head_tr, Body, Id, [], End, LC).
trans(( Head :- Body ), [( Head :- Body ) | LC]) :-
    bridge(Head),
    Head_tr =.. [Head ∘ '_bridge', Head, Id, Cont],
    End = call(Cont),
    transBody(Head_tr, Body, Id, Cont, End, LC).

transBody([], [], _, _, [], []).
transBody(Head, Body, Id, ContPrev, End, [( Head :- Body_tr ) | RestBody_tr]) :-
    following(Body, Pref, Pred, Suff),
    getLBinds(Pref, Suff, LBinds),
    updateBody(Pred, End, Id, Pref, LBinds, ContPrev, Cont, Body_tr),
    transBody(Cont, Suff, Id, ContPrev, End, RestBody_tr).

following(Body, Pref, Pred, Suff) :-
    member(Body, Pred),
    (table(Pred); bridge(Pred)), !,
    Body = Pref ∘ Pred ∘ Suff.

updateBody([], End, _Id, Pref, _LBinds, _ContPrev, [], Pref ∘ End).
updateBody(Pred, _End, Id, Pref, LBinds, ContPrev, Cont, Pref ∘ EndClause) :-
    getNameCont(NameCont),
    Cont = NameCont(Id, LBinds, Pred, ContPrev),
    ( bridge(Pred) ->
          EndClause =.. [Pred ∘ '_bridge', Pred, Id, Cont]
     ;
          EndClause = Call(Cont)
     ).
```

Fig. 8. The Prolog code of the translation rules.

call a continuation which will be received as fourth argument of the generated bridge predicate.

transBody/6 generates, in its last argument, the translation of the body of a clause by processing, in each iteration, the code remaining until either the next tabled / bridge call or the end the clause. In order to do that, following/4 splits a clause body into three parts: a *prefix*, from the beginning of the body up to the first occurrence of a tabled or bridge call, the tabled / bridge call itself, and the rest of the clause (the *suffix*).

The updateBody/8 predicate returns, in its last argument, the translation for the prefix identified by following/4; the list of variables which have to be saved in order to recover the environment of the consumer was already obtained by getLBinds/3. The suffix will be transformed into a continuation to be associated

```
t(A) :- slg(t(A)).
slg_t (t(A), Id) :-
   p_bridge(p(B), Id,                         p(B) :- t(B), B < 1.
            slg_t0(Id, [A], p(B), [])).
                                              p_bridge(p(B), Id, Cont) :-
slg_t (t(0), Id) :- answer(Id, t(0)).            slgcall (p_bridge0(Id, [], t(B), Cont)).

slg_t0(Id, [A], p(B), []) :-                  p_bridge0(Id, [], t(B), Cont) :-
   A is B + 1,                                   B < 1,
   answer(Id, t(A)).                             call(Cont).
```

Fig. 9. The program in Figure 4 after being transformed for tabled execution.

with a new predicate symbol, generated by getNameCont/1. The body of this new predicate is generated by recursively calling transBody/6.

The first clause of updateBody/8 takes care of the base case, when there are no calls to bridge or tabled predicates left, and the End of the clause, generated by trans/2, is appended at the end of the body. Its second clause has two cases which, respectively, generate code for a call to a bridge and a table predicate.

We will now refer to the example in Figure 4, assuming that a :- bridge p/1 declaration has been added to show how a translation would take place.

4.2 Correct Transformation of the Example

The translation of the first clause of t/1 is performed by the third clause of trans/2, which makes the head of the translated clause, Head_tr, to be slg_t (t(A), Id) and states that the final call of that clause has to be answer(Id, t(A)) —i.e., when the clause successfully finishes, it adds the answer to the table.

transBody/6 then takes care of the rest of the body. It identifies the variables which have to be saved (A, in this case) and classifies the body literals as follows:

Pref	Pred	Suff
(none)	p(B)	A **is** B + 1

updateBody/8 generates the body for the predicate associated with Head_tr to give the first clause of slg_t /2, and generates the head (slg_t0 /4) of the clause which corresponds to the translation of Suff. The body of Suff is generated in the recursive call to the trans/6 predicate.

The translation of the second clause of t/1 is simpler, as it only has to add answer(Id, t(0)) at the end of the body of the new predicate.

The original clause for the bridge predicate p/1 is kept to maintain its interface. The translation for the single clause of p/1 is made by the fourth clause of trans/2 where Head_tr is unified with p_bridge(p(B), Id, Cont) and End is unified with **call**(Cont) to resume the pushed continuation. transBody/6 finds an empty list of environment variables and unifies Pref, Pred and Suff with [], t(B) and $B < 1$, respectively. The second clause of updateBody/8 generates the body for the head Head_tr and also the head of the continuation clause which translates Suff (p_bridge0/3). Its body is generated in the recursive call to the trans/6 predicate by the first clause of updateBody/8, after appending Suff and End, generated by trans/2.

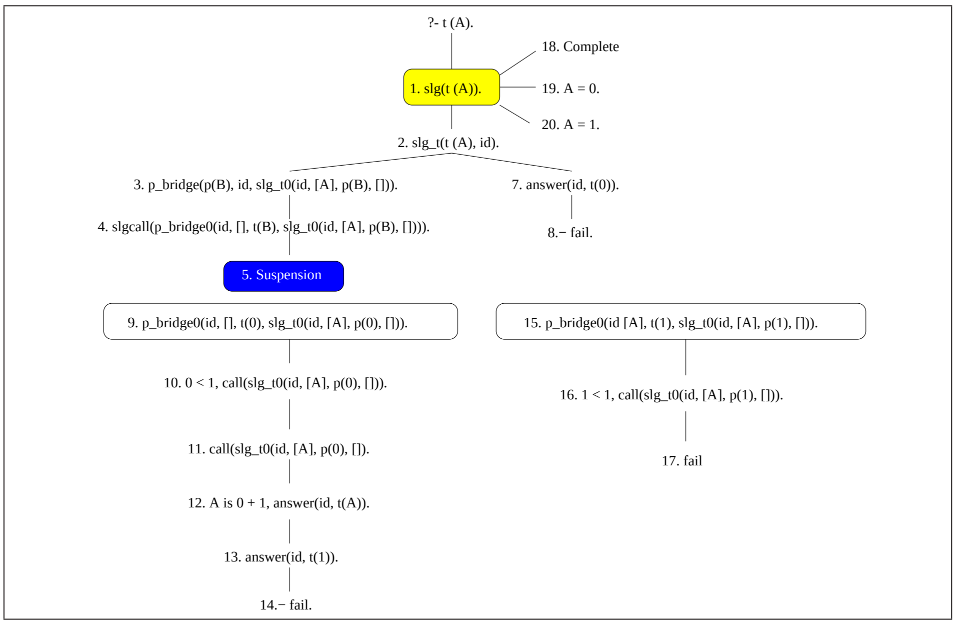
?- t (A).
18. Complete
1. slg(t (A)).
19. A = 0.
20. A = 1.
2. slg_t(t (A), id).
3. p_bridge(p(B), id, slg_t0(id, [A], p(B), [])).
7. answer(id, t(0)).
4. slgcall(p_bridge0(id, [], t(B), slg_t0(id, [A], p(B), []))).
8.− fail.
5. Suspension
9. p_bridge0(id, [], t(0), slg_t0(id, [A], p(0), [])).
15. p_bridge0(id [A], t(1), slg_t0(id, [A], p(1), [])).
10. 0 < 1, call(slg_t0(id, [A], p(0), [])).
16. 1 < 1, call(slg_t0(id, [A], p(1), [])).
11. call(slg_t0(id, [A], p(0), []).
17. fail
12. A is 0 + 1, answer(id, t(A)).
13. answer(id, t(1)).
14.− fail.

Fig. 10. New `CCall` tabling execution.

4.3 Execution of the Transformed Program

The execution tree for the transformed program is shown in Figure 10. It is similar to that in Figure 6, but a continuation slg_t0 (id , [A], p(B), []) is passed to the bridge call to p/1 (step 3). This continuation contains the code to be executed after the execution of p(B) and the list [A] needed to recover the environment of this remaining code. Consequently, there are two nested continuations associated with the suspension (step 4): one continuation to execute the rest of the code of p(B), p_bridge0/4, and another one to execute the rest of the code of t(A), slg_t0 /4. As we can see, bridge predicates push continuations which are called when a consumer is resumed.

After the first answer is found (step 7), this nested continuation is resumed (step 9). After executing the remaining code of p(B) (step 10), the next pushed continuation (fourth argument of p_bridge0/4) is called to execute the remaining code of t(A), and the second answer, t(1), is found (step 13). Again, the (nested) continuation is resumed, but it fails at step 16. Finally, the tabled call can be completed (step 18), and each of its answers are returned by backtracking (steps 19 and 20).

5 Θ(CHAT) is not Comparable with Θ(`CCall`)

In this section we present a comparative analysis of the complexity of `CCall` and CHAT, which is an efficient implementation of tabling with a comparatively simple machinery. Since it is known that Θ(CHAT) is Θ(SLG-WAM) [18], the final conclusion applies to the SLG-WAM as well.

The complexity analysis focuses on the operations of suspension and resumption. The environment of a consumer has to be protected when suspending to reinstall it when resuming. `CCall` achieves that by copying the continuation associated with the consumer in a special memory area to be protected on backtracking. In the original implementation [1] this continuation is copied from the heap to a separate table (when suspending) and back (when resuming). Alternatively, continuations can be saved in a special memory area with the same data format as the heap [16]. This makes it possible to use WAM code directly on them and, when resuming, they can be directly used as normal Prolog data, without having to copy them each time a consumer is resumed.

On the other hand, CHAT freezes the heap and the frame stack when suspending. These areas are frozen by traversing the choice point stack. For all choice points between the consumer choice point and its generator, their pointers to the end of the heap and frame stack are changed the consumer choice point values. By doing that, heap and frame stack are preserved on backtracking. However, the consumer choice point and the trail segment between consumer and generator (with its associated values) have to be copied onto a special memory area. This makes it possible to reinstall the values of the variables which were bound when suspending (and which backtracking will unbind) when resuming.

Each consumer is suspended only once, and it can be resumed several times. The rest of the operations, i.e., checking if a tabled call is a generator or a consumer, are not analyzed, because they are common to both systems. In addition, we will ignore the cost of working at the Prolog level, since this is an orthogonal issue: `CCall` primitives could be compiled to WAM instructions and working at Prolog level does not increase the complexity. Finally, for simplicity, we assume that both systems use the same scheduling strategy and that the *leader*[8] does not change between the suspension and the resumptions of a consumer.

Θ(`CCall`): when suspending, `CCall` has to copy all the environments until the last generator and the structures in the heap which hang from them. Let E be the size of all the environments and H the size of the structures in the heap. The time consumption when suspending is $\Theta(E + H)$. When resuming, `CCall` only needs to perform pattern matching of the continuation against its clause. The time taken by this matching depends on the size of the list of bindings, which is known to be $\Theta(E)$. Since each consumer can be resumed N times, the time consumption of resuming consumers is $\Theta(N \times E)$.

Θ(CHAT): when suspending, CHAT has to traverse the frame and choicepoint stacks, but with the improvements presented in [18], this time can be neglected because a choice point is only traversed once for all the consumers, and only the trail and the last choice point have to be copied. Let T be the trail size and C the choice point size, which is bound by a constant for a given program. The time consumption when suspending is: $\Theta(T)$. When resuming, CHAT has to reinstall the values of the frame and the choice point. Since each consumer can be resumed N times, the time consumption of resuming is $\Theta(N \times T)$.

[8] The leader of a given consumer, C, is the generator which execute the completion algorithm of the generator of C.

Analyzing the worst cases of both systems: we can conclude $E + H \geq T$, because each variable can be only once in the trail, and then `CCall` is worse than CHAT when suspending. On the other hand, if $E < T$, than `CCall` is better than CHAT when resuming. Consequently, for a plausible general case, the more resumptions there are, the better `CCall` behaves in comparison with CHAT, and conversely. In any case, the worst and best cases for each implementation are different, which makes them difficult to compare. For example, if there is a very large structure pointed to from the environments, and none of its elements are pointed to from the trail, `CCall` is slower than CHAT, since it has to copy all the structure in a different memory area when suspending and CHAT does nothing both when suspending and when resuming.

On the other hand, if all the elements of the structure are pointed to from the trail, `CCall` has to copy all the structure on suspension in a different memory area to protect it on backtracking, but it is ready to be resumed without any other operation (just a unification with the pointer to the structure). CHAT has to copy all the structure on suspension too, because all the structure is in the trail. In addition, each time the consumer is resumed, all the elements of the structure have to be reinstalled using the trail, and CHAT has to perform more operations than `CCall`, and then, the more resumptions there are, the worse CHAT would be in comparison with `CCall`. Anyway, as the trail is usually smaller than the heap, we expect CHAT to outperform `CCall` in most cases.

6 Performance Evaluation

We have implemented the proposed technique as an extension of the Ciao system [19], using the improvements presented in [16]. Tabled evaluation is provided to the user as a loadable *package* [20] that implements the new directives and user-level predicates, performs the program transformations, and links in the low-level support for tabling.

Table 1 compares the proposed implementation of tabling with the latest versions of state-of-the-art systems, namely, XSB 3.1 (SLG-WAM), YapTab 5.1.3 (SLG-WAM) [21], and B-Prolog 7.1 on benchmarks also used in other similar performance evaluations. We provide the raw time (in milliseconds) taken to execute these benchmarks and the number of bridge predicates which appear in the new translation. Measurements have been made with Ciao-1.13, using the standard, unoptimized bytecode-based compilation, and with the `CCall` extensions loaded. Note that we did not compare with CHAT, which was available as a configuration option in the XSB system but which was removed in recent XSB versions, since it was experimentally found to be overall slower than the SLG-WAM [22]. All the executions were performed using local scheduling and disabling garbage collection; in the end this did not impact execution times very much. We used `gcc 4.1.1` to compile all systems (when necessary), and executed on Fedora Core Linux, kernel 2.6.9, on an Intel Xeon *Deschutes* processor.

The first benchmark is `path`, the same as Figure 1, which has been executed with a linear (each node follows, and is followed by, only one node, as in a chain) graph. Since this is a tabling-intensive program with no consumers in its execution, the difference with other systems is mainly due to large parts of

Program	`CCall`	XSB	YapTab	BProlog	# Bridges
path	517.92	231.4	151.12	206.26	0
tcl	96.93	59.91	39.16	51.60	0
tcr	315.44	106.91	90.13	96.21	0
tcn	485.77	123.21	85.87	117.70	0
sgm	3151.8	1733.1	1110.1	1474.0	0
atr2	689.86	602.03	262.44	320.07	0
pg	15.240	13.435	8.5482	36.448	6
kalah	23.152	19.187	13.156	28.333	20
gabriel	23.500	19.633	12.384	40.753	12
disj	18.095	15.762	9.2131	29.095	15
cs_o	34.176	27.644	18.169	85.719	14
cs_r	66.699	55.087	34.873	170.25	15
peep	68.757	58.161	37.124	150.14	10

Table 1. Comparing Ciao+`CCall` with XSB, YapTab, and B-Prolog.

the execution being done at Prolog level. The following five benchmarks, until `atr2`, are also tabling intensive. As their associated environments are very small, `CCall` is far from its worst case (see Section 5), and the difference with other systems is similar to that in `path` and for a similar reason. The worst case in this set is `tcn` because there are two calls to `slgcall/1` per generator, and the overhead of working at the Prolog level is duplicated.

B-Prolog, which uses linear tabling, suffers from performance problems when costly predicates have to be recomputed: this is what happens in benchmarks from `pg` until `peep`, where tabled and non-tabled execution is mixed. This is a well-known disadvantage of linear tabling techniques which does not affect suspension-based approaches. It has to be noted, however, that the latest versions of B-Prolog implement an optimized variant of its original linear tabling mechanism [14] which tries to avoid reevaluation of looping subgoals.

The difference in speed for SLD execution, at least in those cases where the program execution is large enough to be really significant, must also be taken into account in order to compare the efficiency of our implementation. XSB was shown to be between 1.8 and 2 times slower than Ciao (partially due to being always prepared for tabling execution) and YapTab was about 1.5 times faster.[9]

In non-trivial benchmarks, from `pg` until `peep`, which at least in principle should reflect more accurately what one might expect in larger applications using tabling, execution times are in the end competitive with XSB or YapTab. This is probably due to the fact that the raw speed of the basic engine in Ciao is higher than in XSB and closer to YapTab, rather than to factors related to tabling execution, but it also implies that the overhead of the approach to tabling used is reasonable after the optimizations in [16]. In this context it should be noted that in these experiments we have used the baseline, bytecode-based compilation and abstract machine. Turning on global analysis and using optimizing compilers and abstract machines [23–25] can improve the speed of both the SLD part of the computation and (the Prolog part of) tabling.

[9] Note that the tabling-enabled version of Yap is somewhat slower than regular Yap.

7 Conclusions

We have presented an extension of the continuation call technique which eliminates its limitations when interleaving tabled and non-tabled predicates. Our approach has the advantage of being easier to implement and maintain than other techniques, which usually require non-trivial modifications to low-level machinery. We expect the overhead caused by executing at Prolog level to be reduced as the speed of the source language improves by using global analysis, optimizing compilers, and better abstract machines. Accordingly, we expect the performance of `CCall` to improve in the future and thus gradually gain ground in the comparisons.

Although a non-optimal tabled execution is obviously a disadvantage, it is worth noting that, since our implementation does not (or only very slightly [16]) changes the WAM or the Prolog compiler, the speed at which regular Prolog is executed remains unchanged. In our case, executables which do not need tabling have very little tabling-related code, as the data structures (for tries, etc.) are handled by dynamic libraries loaded on demand, and only stubs are needed in the regular engine. Additionally, the modular design of our approach gives better chances of making it easier to port to other systems.

References

1. Ramesh, R., Chen, W.: Implementation of tabled evaluation with delaying in prolog. IEEE Trans. Knowl. Data Eng. **9**(4), pp. 559–574 (1997)
2. Tamaki, H., Sato, M.: OLD resolution with tabulation. In: Third International Conference on Logic Programming, London, pp. 84–98. Lecture Notes in Computer Science, Springer-Verlag (1986)
3. Warren, D.: Memoing for logic programs. Communications of the ACM **35**(3), pp. 93–111 (1992)
4. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43**(1), pp. 20–74 (January 1996)
5. Ramakrishnan, R., Ullman, J.D.: A survey of research on deductive database systems. Journal of Logic Programming **23**(2), pp. 125–149 (1993)
6. Warren, R., Hermenegildo, M., Debray, S.K.: On the Practicality of Global Flow Analysis of Logic Programs. In: Fifth International Conference and Symposium on Logic Programming, pp. 684–699. MIT Press (August 1988)
7. Dawson, S., Ramakrishnan, C., Warren, D.: Practical Program Analysis Using General Purpose Logic Programming Systems – A Case Study. In: Proceedings of PLDI'96, New York, USA, pp. 117–126. ACM Press (1996)
8. Zou, Y., Finin, T., Chen, H.: F-OWL: An Inference Engine for Semantic Web. In: Formal Approaches to Agent-Based Systems. Volume 3228 of Lecture Notes in Computer Science., pp. 238–248. Springer Verlag (January 2005)
9. Ramakrishna, Y., Ramakrishnan, C., Ramakrishnan, I., Smolka, S., Swift, T., Warren, D.: Efficient Model Checking Using Tabled Resolution. In: Computer Aided Verification. Volume 1254 of Lecture Notes in Computer Science., pp. 143–154. Springer Verlag (1997)
10. Sagonas, K., Swift, T.: An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. ACM Transactions on Programming Languages and Systems **20**(3), pp. 586–634 (May 1998)

11. Demoen, B., Sagonas, K.: CAT: The Copying Approach to Tabling. In: Programming Language Implementation and Logic Programming. Volume 1490 of Lecture Notes in Computer Science., pp. 21–35. Springer-Verlag (1998)
12. Demoen, B., Sagonas, K.F.: Chat: The copy-hybrid approach to tabling. In: Practical Applications of Declarative Languages. pp. 106–121. (1999)
13. Zhou, N.F., Shen, Y.D., Yuan, L.Y., You, J.H.: Implementation of a linear tabling mechanism. Journal of Functional and Logic Programming **2001**(10), (October 2001)
14. Zhou, N.F., Sato, T., Shen, Y.D.: Linear Tabling Strategies and Optimizations. Theory and Practice of Logic Programming **8**(1), pp. 81–109 (2008)
15. 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. pp. 181–196. (2001)
16. de Guzmán, P.C., Carro, M., Hermenegildo, M., Silva, C., Rocha, R.: An Improved Continuation Call-Based Implementation of Tabling. In Warren, D., Hudak, P., eds.: 10th International Symposium on Practical Aspects of Declarative Languages (PADL'08). Volume 4902 of LNCS., pp. 198–213. Springer-Verlag (January 2008)
17. Casas, A., Cabeza, D., Hermenegildo, M.: A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In: FLOPS'06, Fuji Susono (Japan). (April 2006)
18. Demoen, B., Sagonas, K.: CHAT is θ(SLG-WAM). In H. Ganzinger, D.M.A., Voronkov, A., eds.: International Conference on Logic for Programming and Automated Reasoning. Volume 1705 of Lectures Notes in Computer Science., pp. 337–357. Springer (September 1999)
19. Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., (Eds.), G.P.: The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM) (2006) Available at `http://www.ciaohome.org`.
20. Cabeza, D., Hermenegildo, M.: The Ciao Modular, Standalone Compiler and Its Generic Program Processing Library. In: Special Issue on Parallelism and Implementation of (C)LP Systems. Volume 30(3) of Electronic Notes in Theoretical Computer Science. Elsevier - North Holland (March 2000)
21. Rocha, R., Silva, F., Costa, V.S.: YapTab: A Tabling Engine Designed to Support Parallelism. In: Conference on Tabulation in Parsing and Deduction. pp. 77–87. (2000)
22. Castro, L., Swift, T., Warren, D.: Suspending and resuming computations in engines for SLG evaluation. In: Practical Applications of Declarative Languages. Volume 2257 of LNCS., pp. 332–346. Springer-Verlag (2002)
23. Carro, M., Morales, J., Muller, H., Puebla, G., Hermenegildo, M.: High-Level Languages for Small Devices: A Case Study. In Flautner, K., Kim, T., eds.: Compilers, Architecture, and Synthesis for Embedded Systems, pp. 271–281. ACM Press / Sheridan (October 2006)
24. Morales, J., Carro, M., Hermenegildo, M.: Improving the Compilation of Prolog to C Using Moded Types and Determinism Information. In: Proceedings of the Sixth International Symposium on Practical Aspects of Declarative Languages. Number 3057 in LNCS, Heidelberg, Germany, pp. 86–103. Springer-Verlag (June 2004)
25. Morales, J., Carro, M., Hermenegildo, M.: Comparing Tag Scheme Variations Using an Abstract Machine Generator. In: 10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08), pp. 32–43. ACM Press (July 2008)