A Sketch of a Complete Scheme for Tabled Execution Based on Program Transformation

Pablo Chico de Guzmán[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. Tabled evaluation has proved to be an effective method to improve several aspects of goal-oriented query evaluation, including termination and complexity. "Native" implementations of tabled evaluation offer good performance, but also require significant implementation effort, affecting compiler and abstract machine. Alternatively, program transformation-based implementations, such as the original *continuation call* (`CCall`) technique, offer lower implementation burden at some efficiency cost. A limitation of the original `CCall` proposal is that it limits the interleaving of tabled and non-tabled predicates and thus cannot be used for arbitrary programs. In this work we present an extension of the `CCall` technique that allows the execution of arbitrary tabled programs, as well as some performance results. Our approach offers a useful trade-off that can be competitive with state-of-the-art implementations, while keeping implementation effort relatively low.
Keywords: Tabled logic programming, Continuation-call tabling, Implementation, Performance, Program transformation.

1 Introduction

Tabling [1–3] is a strategy for executing logic programs which *remembers* already processed calls and their answers to overcome several limitations of SLD resolution: non-termination due to repeated subgoals can sometimes be avoided (tabling ensures termination of bounded term-size programs) and some cases of recomputation can also be automatically optimized. The first occurrence (the *generator*) of a call to a predicate marked as *tabled* and subsequent calls which are identical up to variable renaming (the *consumers*) are recognized. The generator applies resolution using program clauses to derive answers for the goal. Conceptually, consumers *suspend* their current execution path and take on a different branch; this can be repeated several times. When some alternative branch eventually succeeds the answer generated for the initial query is inserted in a table associated with the original goal. This makes it possible to reactivate suspended calls and to continue execution at the point where they were stopped.

Implementing tabling is a complex task. In *suspension-based tabling* (e.g., XSB [4] and CHAT [5], among others) the execution state of suspended tabled subgoals is preserved to avoid unnecessary recomputations, but they usually require deep changes to the underlying implementation. *Linear tabling* schemes (as exemplified by B-Prolog [6, 7] and the DRA scheme [8]) does not require suspension and resumption of sub-computations, and then, they can usually be implemented on top of existing sequential engines with relatively simple modifications. However, their efficiency is affected by subgoal recomputation.

2 The Continuation Call Technique

The `CCall` approach to tabling [9, 10] is a suspension-based mechanism which requires much simpler modifications to the Prolog implementation or compiler than other suspension-based techniques. A number of low-level optimizations to existing implementations of the `CCall` approach were proposed in [11] and it was shown that performance could be competitive with other implementations.

The `CCall` technique 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. Consumer suspension and resumption is performed by operations which are visible at Prolog level.

Roughly speaking, the original `CCall` approach calls tabled predicates through the `slgcall` primitive, which receives a goal and analyzes if it is a generator or a consumer call. When it is a consumer, suspension has to be performed by saving the current environment and program counter in order to resume execution later on. The body goals after the tabled call are associated with a new predicate symbol, which takes the role of the program counter at that particular place. The bindings performed before the tabled call make up the environment of the consumer. Consequently, `slgcall` takes the name of the auxiliary predicate and a list of bindings as arguments, in order to be able to perform resumption at Prolog level. Answers are inserted in the table by `answer/2` primitive, which is added at the end of each clause of the original tabled predicate.

The original transformation is not general because the environments are only correctly saved when tabled calls are themselves in the body of a tabled predicate (except for the first one). If there are non-tabled, SLD predicates between the generator and some consumer, the code after that consumer is not associated with any predicate symbol, and it is not considered for tabled execution (see [12] for more details and examples). Tabling *all* predicates between generators and consumers works around this problem, but it can seriously impact efficiency.

3 A Complete Tabling Translation for General Programs

We have extended the translation to work around the issue presented in the previous section by bringing into the scene a new kind of predicates – *bridge predicates*. Predicate `B` is a bridge if for some tabled predicate `T`, `T` depends on `B` (i.e., `B` is called in the subtree rooted at `T`) and `B` depends on `T`. Figure 1, which uses a sugared Prolog-like language,[3] shows the rules for the new translation.

The `tr/2` predicate takes a clause to be translated and returns the list of clauses resulting from the translation. Its last clause ensures that predicates which are non-tabled and non-bridge are not transformed. The first one generates the interface with the rest of the code for each tabled predicate. The second and third cases translate clauses of tabled and bridge predicates, respectively.[4] They

[3] Functional syntax is implicitly assumed where needed. The '$\circ$' operator is a general `append` function which can either join (linear) structures or concatenates atoms.

[4] Predicates `table/1` and `bridge/1` check if their argument corresponds to a tabled or bridge predicate, respectively.

```
tr((:- table P/N),
   (P(X1..Xn) :- !,slg(P(X1..Xn)))).
tr((H :- B),LC) :- !,
   table(H),
   H_tr =.. ['slg_' ∘ H, H, Id],
   End = answer(Id, H),
   tr_B(H_tr, B, Id, [], End, LC).
tr((H :- B), (H :- B ∘ LC)) :- !,
   bridge(H),
   H_tr =.. [H ∘ '_bridge', H, Id, Cont],
   End = (arg(3, Cont, H), call(Cont)),
   tr_Body(H_tr, B, Id, Cont, End, LC).
tr(C, C).

tr_Body([], [], _, _, [], []).
tr_Body(H, B, Id, CCPrev, End,
        (H :- B_tr ∘ RestB_tr)) :-
   following(B, Pref, Pred, Suff),
   getLBinds(Pref, Suff, LBinds),
   up_Body(Pred, End, Id, Pref, LBinds,
           CCPrev, Cont, B_tr),
   tr_Body(Cont, Suff, Id, CCPrev, End, RestB_tr).

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

up_Body([], End, _Id, Pref, _LBinds,
        _CCPrev, [], Pref ∘ End).
up_Body(Pred, _End, Id, Pref, LBinds,
        CCPrev, Cont, Pref ∘ slgcall(Cont)) :-
   table(Pred),
   getNameCont(NameCont),
   Cont = NameCont(Id, LBinds, Pred, CCPrev).
up_Body(Pred, _End, Id, Pref, LBinds,
        CCPrev, Cont, Pref ∘ Bridge_call) :-
   bridge(Pred),
   getNameCont(NameCont),
   Cont = NameCont(Id, LBinds, Pred, CCPrev),
   Bridge_call =.. [Pred ∘ '_bridge', Cont].
```

Fig. 1. The Prolog code of the translation rules.

generate the new head of the clause, `H_tr`, and the code which has to be appended at the end of the body, `End`, before calling `tr_Body/6` with these arguments. The original clauses are maintained in case bridge predicates are called outside a tabled call.

`tr_Body/6` generates, in its last argument, the translation of the body of a clause by taking care, in each iteration, of the code until the next tabled or bridge call, or until the end the clause, and appending the translation of the rest of the clause to this partial translation.

`following/4` splits a clause body in three parts: a prefix, until the first time a tabled or bridge call appears, the tabled or bridge call itself, and a suffix from this call until the end of the clause. `getLBinds/3` obtains the list of variables which have to be saved to recover the environment of the consumer.

The `up_Body/8` predicate completes the body prefix until the next tabled or bridge call. Its first sixth arguments are inputs, the seventh one is the head of the continuation for the suffix of the body, and the last argument is the new translation for the prefix. The first clause takes care of the base case, when there are no calls to bridge or tabled predicates left, the second clause generates code for a call to a tabled predicate, and the last one does the same with a bridge predicate. `getNameCont/1` generates a unique name for the continuation.

```
:- table t/1.
t(A):-
    p(B), A is B + 1.
t(0).
p(B):- t(B), B < 1.
```

Fig. 2. A program which needs bridge predicates.

An example of a tabled program which needs our extended translation is presented in the figure right above this paragraph (and, at more length, in [12]). If the query `?- t(A).` is issued, `t(B)` is called in a consumer position inside `p/1`. This simple combination would incorrectly be dealt by [9]. However, the translation proposed in Figure 1 generates the code in Figure 3, which transforms `p/1` so that the information necessary to resume `t/1` is available where needed, at the cost of some duplicated code and an extra argument when `p/1` is called from inside a tabled execution.

```
t(A) :- slg(t(A)).
slg_t(t(A), Id) :-
   p_bridge(Id,
            slg_t0(Id,[A],p(B),[])).
slg_t0(Id, [A], 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.
p_bridge(Id, Cont) :-
   slgcall(
      p_bridge0(Id,[],t(B),Cont)).
p_bridge0(Id, [], t(B), Cont) :-
   B < 1,
   arg(3, Cont, p(B)),
   call(Cont).
```

Fig. 3. The program in Figure 3 after being transformed for tabled execution.

Prog.	Ciao+CCall	XSB	YapTab	BProl.
path	517.92	231.4	151.12	206.26
tcl	96.93	59.91	39.16	51.60
tcr	315.44	106.91	90.13	96.21
tcn	485.77	123.21	85.87	117.70
sgm	3151.8	1733.1	1110.1	1474.0
atr2	689.86	602.03	262.44	320.07
pg	15.240	13.435	8.5482	36.448

Prog.	Ciao+CCall	XSB	YapTab	BProl.
kalah	23.152	19.187	13.156	28.333
gabriel	23.500	19.633	12.384	40.753
disj	18.095	15.762	9.2131	29.095
cs_o	34.176	27.644	18.169	85.719
cs_r	66.699	55.087	34.873	170.25
peep	68.757	58.161	37.124	150.14

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

4 Performance Evaluation

We have implemented the proposed technique as an extension of the Ciao system [13] with the efficiency improvements presented in [11] and the new translation for general programs explained in this poster.

Table 1 aims at determining how the proposed implementation of tabling compares with state-of-the-art systems —namely, the available versions of XSB, YapTab, and B-Prolog at the time of writing. We provide the raw time (in milliseconds) taken to execute several tabling benchmarks. Measurements have been made with Ciao-1.13, using the standard, unoptimized bytecode-based compilation, and with the `CCall` extensions loaded, as well as in XSB 3.0.1, YapTab 5.1.1, and B-Prolog 7.0. 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 the systems (except B-Prolog, which is available as a binary), and we executed them on a machine with Fedora Core Linux, kernel 2.6.9, and an Intel Xeon *Deschutes* processor.

While the performance of `CCall` is clearly affected by the fragment of the execution performed at Prolog level, its efficiency is in general not too far away from than XSB's, whose abstract machine is about half the speed of Ciao's for SLD execution. The relationship with B-Prolog is not so clear, as it features a fast abstract machine, but its tabling implementation sometimes suffers from recomputation. Last, Yap, which has a fast abstract machine which implements SLG resolution, easily beats the rest of the systems. We plan to improve the performance of our implementation by making the `CCall` primitives closer to the abstract machine. More details about the execution times and a comparison of the `CCall` time execution complexity with CHAT can be found in [12].

Acknowledgments: This work was funded in part by EU FP6 FET project IST-15905 *MOBIUS*, FP7 grant 215483 *S-Cube*, Spanish MEC project TIN2005-09207-C03 *MERIT-COMVERS*, ITEA2/PROFIT FIT-340005-2007-14 *ES_PASS*, and by Madrid Regional Government project S-0505/TIC/0407 *PROMESAS*. M. Hermenegildo was also funded in part by the Prince of Asturias Chair in IST at UNM. Pablo Chico de Guzmán was also funded by a UPM doctoral grant.

References

1. 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)
2. Warren, D.: Memoing for logic programs. Communications of the ACM **35**(3), pp. 93–111 (1992)
3. Chen, W., Warren, D.S.: Tabled Evaluation with Delaying for General Logic Programs. Journal of the ACM **43**(1), pp. 20–74 (January 1996)
4. 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)
5. Demoen, B., Sagonas, K.F.: Chat: The copy-hybrid approach to tabling. In: Practical Applications of Declarative Languages. pp. 106–121. (1999)
6. 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)
7. 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)
8. 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)
9. Ramesh, R., Chen, W.: A Portable Method for Integrating SLG Resolution into Prolog Systems. In Bruynooghe, M., ed.: International Symposium on Logic Programming, pp. 618–632. MIT Press (1994)
10. Rocha, R., Silva, C., Lopes, R.: On Applying Program Transformation to Implement Suspension-Based Tabling in Prolog. In Dahl, V., Niemelä, I., eds.: 23rd International Conference on Logic Programming. Number 4670 in LNCS, Porto, Portugal, pp. 444–445. Springer-Verlag (September 2007)
11. 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)
12. de Guzmán, P.C., Carro, M., Hermenegildo, M.V.: Bridge Program Transformation for the CCall Tabling Scheme. Technical Report CLIP6/2008.0, Technical University of Madrid (UPM), Computer Science School, UPM (September 2008)
13. 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`.