Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI-16)

Tabling as a Library with Delimited Control

Benoit Desouter, Marko van Dooren
Ghent University, Belgium
benoit.desouter@ugent.be
marko.vandooren@ugent.be

Tom Schrijvers, Alexander Vandenbroucke
KU Leuven, Belgium
tom.schrijvers@kuleuven.be
alexander.vandenbroucke@kuleuven.be

Abstract

The logic programming language Prolog uses a resource-efficient SLD resolution strategy for query answering. Yet, its propensity for non-termination seriously detracts from the language's declarative nature. This problem is remedied by *tabling*, a modified execution strategy that allows a larger class of programs to terminate. Unfortunately, few Prolog systems provide tabling, because the documented implementation techniques are complex, low-level and require a prohibitive engineering effort.

To enable more widespread adoption, this paper presents a novel implementation of tabling for Prolog that is both high-level and compact. It comes in the form of a Prolog library that weighs in at under 600 lines of code, is based on delimited control and delivers reasonable performance.

1 Introduction

The essence of programming is crisply captured in Kowalski's adage ALGORITHM = LOGIC + CONTROL [1979]. The ideal of logic programming is that the programmer should be able to focus only on the first part, the problem logic, while the control should be supplied by the programming system.

Unfortunately, traditional Prolog systems fall short of reaching this ideal as programmers need to be constantly aware of Prolog's SLD resolution strategy [Kowalski, 1974], which processes rules in a "top-down, left-to-right" fashion. Consider for example the logic rules for computing the transitive closure of the `e/2` relation:

```
p(X,Y) :- p(X,Z), e(Z,Y).
p(X,Y) :- e(X,Y).

e(1,2). e(2,3).
```

Although the logic is correct, Prolog diverges when resolving any `p(X,Y)` query. Unfortunately, it is up to the programmer to diagnose the left recursion in the first rule as the culprit, and to address the issue by eliminating the left recursion and by reordering the rules. Moreover, to handle a cycle in the graph,[1] the programmer needs to pollute his declarative model more thoroughly with control logic. This clearly goes against the grain of declarative programming.

Tabling is a variation on SLD resolution that does not get stuck in cyclic derivations, and thus captures the declarative least fixed-point semantics of logic programs more completely. It works by storing the intermediate answers to predicates (in a data structure known as a table), and reusing them instead of recomputing them, whenever possible. At the same time cyclic derivations are suspended and only resumed when new answers are available. This approach not only improves the termination behavior, but may also drastically improve performance—be it at the cost of increased memory usage.

Given these advantages, it may come as a surprise that not many Prolog systems support tabling. The reason for this is that existing implementations, such as those of Yap [Santos Costa *et al.*, 2012] and XSB [Swift and Warren, 2012], require pervasive changes to the Prolog engine, the Warren Abstract machine (WAM) [Warren, 1983; Aït-Kaci, 1999] or one of its variants. This is a substantial engineering effort that is beyond most systems [Santos Costa *et al.*, 2012].

Several attempts have been made to tame the complexity by means of transformations, calls to C routines and very specific changes to the WAM [Ramesh and Chen, 1994; Zhou *et al.*, 2000; Guo and Gupta, 2001; 2004]. Nevertheless, these approaches incur substantial technical debt, have a high maintenance and porting cost, and the development effort cannot be amortised over other features. In contrast, extension tables [Fan and Dietrich, 1992] provide a high-level tabling mechanism that is implemented directly in Prolog. However, the approach cannot achieve satisfactory performance as suspended goals are always re-evaluated.

We improve upon the current state of the art with a novel lightweight implementation of tabling based on delimited control. Our approach comes in the form of a Prolog library that weighs in at less than 600 lines of code, delivers acceptable performance, and requires only a minimal extension to the Prolog system: *delimited control* [Schrijvers *et al.*, 2013b]. Moreover, the development effort of delimited control can be amortized over the range of high-level language features they enable, such as *effect handlers* [Plotkin and Pretnar, 2013].

The remainder of this paper provides a high-level overview of our contribution. We refer to the extended paper [Desouter *et al.*, 2015] for more details.

[1] e.g., obtained by adding the fact `e(2,1)`

2 Denotational Semantics, SLD Resolution and Tabling

This section reviews the denotational semantics of logic programs [Lloyd, 1984] and explains the connection to tabling.

Consider a definite clause program P. The *Herbrand base* H_P of P is the set of all ground atoms in P. A Herbrand interpretation I states which ground atoms are true and which are false. By convention, we represent I by the set of true atoms (i.e. $\forall a \in \mathrm{H}_P : I \models a$ iff $a \in I$).

The immediate consequence operator $\mathrm{T_P}(I)$ of P captures which atoms follow directly from the given interpretation I by one of the rules in the program.

$$\mathrm{T_P}(I) = \{\alpha \in \mathrm{H}_P \mid \alpha \leftarrow B_1, \ldots, B_n \text{ is a ground instance of a clause in } P \wedge \{B_1, \ldots, B_n\} \subseteq I\}$$

The conventional denotational semantics for P is the unique interpretation I that is the least fixed-point of $\mathrm{T_P}$, also known as the *least Herbrand model* of the program. This interpretation contains those and only those atoms that follow from the program and that are not self-supported.

Example 1 *Consider the following program P:*

```
p(a). p(b).
q(X) :- p(X).
```

Its Herbrand base is $\{p(a), p(b), q(a), q(b)\}$ and its fixpoint semantics is lpf$(\mathrm{T_P})$ = $\{p(a), p(b), q(a), q(b)\}$.

It can be show that the least fixed-point of $\mathrm{T_P}$ is $\mathrm{T_P}{\uparrow}^{\omega}$ where $\mathrm{T_P}{\uparrow}^{\omega}$ is defined as:

$$\begin{aligned} \mathrm{T_P}{\uparrow}^{0} &= \varnothing \\ \mathrm{T_P}{\uparrow}^{n} &= \mathrm{T_P}\left(\mathrm{T_P}{\uparrow}^{n-1}\right), n > 0 \\ \mathrm{T_P}{\uparrow}^{\omega} &= \bigcup_{n \geq 0} \mathrm{T_P}{\uparrow}^{n} \end{aligned}$$

This definition suggests a naive bottom-up evaluation strategy, which is used in an improved *semi-naive* form by Datalog systems. However, this strategy is impractical for query answering in the general Prolog setting. Firstly, compound terms give rise to both an infinite Herbrand model and an infinite least Herbrand model which cannot be practically computed. Secondly, the bottom-up strategy can be overly expensive because it derives more facts than necessary for answering the query at hand.

Hence, Prolog uses the top-down strategy of SLD resolution, essentially based on $\mathrm{T_P}^{-1}$ to reason backwards from the query and only consider relevant facts. Unfortunately, this backwards chaining strategy easily gets trapped in cyclic derivations. In contrast, tabling combines the best elements of both approaches: the efficiency of top-down SLD resolution and the cycle-insensitivity of bottom-up least fixed-point computation. Tabling's backbone is top-down resolution, but paired with active cycle detection. It replaces infinite cycles with a forward-chaining least fixed-point strategy, not unlike the immediate consequence operator $\mathrm{T_P}$, but switches back to top-down resolution for previously unexplored queries. Like in the bottom-up strategy tabling comes at the cost of storing the answers to intermediate queries. To mitigate this cost, most systems use SLD resolution by default and allow the programmer to enable tabling for individual predicates.

The hybrid top-down/bottom-up strategy of tabling requires complex control to deviate from the default SLD resolution. This control is typically implemented at a low level in the Prolog abstract machine, where it cross-cuts the existing architecture in a very intricate manner. In this work, we propose an alternative high-level implementation approach, based on a high-level language feature for manipulating SLD resolution control flow from within the program: *delimited control*.

3 Delimited Continuations

Delimited control [Felleisen, 1988; Danvy and Filinski, 1990] is the key ingredient of our lightweight tabling approach. This technique originates in functional programming and was recently introduced in Prolog by Schrijvers et al. [2013b; 2013a] in the form of two built-ins: `reset/3` and `shift/1` for delimiting and capturing the continuation respectively.

- `reset(Goal,Cont,Term1)` executes `Goal`. If `Goal` calls `shift(Term2)`, its further execution is suspended and unified with continuation `Cont`. A continuation is an unspecified Prolog term, which can be resumed using `call/1`. It can be called, saved, copied and compared like any other term, but it is opaque: from its representation we cannot determine anything about the actual goals it represents.
- `shift(Term2)` unifies the remainder of `Goal` up to the nearest call to `reset/3` (i.e., the delimited continuation) with `Cont`, and its return value `Term2` with `Term1`. Finally, it returns control to just after the `reset/3` goal.

The following example illustrates these two built-ins.

Example 2 *Consider the following variation on the transitive closure program:*

```
p(X,Y) :- e(X,Y).
p(X,Y) :- shift(t(X,Z)), e(Z,Y).

e(1,2). e(3,4).
```

If we delimit the query `p(1,Y)` with `reset/3`, we get two answers.

```
?- reset(p(1,Y),Cont,Term).
Y = 2, Cont = 0, Term = 0 ;
Cont = ..., Term = t(1,Z) .
```

The first is a proper answer obtained via the first rule.[2] *The second answer follows from the second rule: `shift` has ended the resolution prematurely without a proper answer and captured the pending subgoal `e(Z,Y)` in `Cont`.*

If we happen to know of an alternative edge from node 1 to node 3, we can instantiate `Z` accordinly, through the unification of `Term`. Then we can resume the continuation to get another proper answer.

```
?- reset(p(1,Y),Cont,Term),
   Term = t(1,3),
   call(Cont).
Cont = ..., Term = t(1,3), Y = 4 .
```

[2] The dummy value 0 indicates that `Cont` and `Term` are not used.

```
delim(Wrapper,Worker,Table) :-
   reset(Worker,Continuation,SourceCall),
   ( Continuation == 0 ->
     store_answer(Table,Wrapper)
   ;
     SourceCall = call_info(_,SourceTable),
     TargetCall = call_info(Wrapper,Table),
     Dependency = dependency(SourceCall,
       Continuation,TargetCall),
     store_dependency(SourceTable,Dependency)
   ).
```

Figure 1: Delimited execution.

4 Implementation

This section provides a high-level overview of our tabling implementation. For reasons of brevity, we leave out the non-essential details, such as the definitions of the data structures involved, which are included in the full version of this paper [Desouter *et al.*, 2015].

The main execution strategy is Prolog's native SLD-resolution. However, delimited control allows us to interrupt this process when a cycle is detected, set the ongoing derivation aside for the time being, and instead explore alternatives.

Cycle-Free Phase Our implementation introduces tabling by means of a shallow source-to-source program transformation. Here is the result for our running example:

```
p(X,Y) :- table(p(X,Y),p_aux(X,Y)).

p_aux(X,Y) :- p(X,Z), e(Z,Y).
p_aux(X,Y) :- e(X,Y).
```

The body of the original predicate has been moved into an auxiliary *worker* predicate `p_aux/2` and `p/2` now wraps the generic tabling logic `table/2` around this worker.

The tabling logic intercepts all calls to the predicate and distinguishes three different scenarios.

1. The call has not previously been encountered. Then control is passed to the worker to compute the answers, which are stored in the table and finally returned. This scenario is covered by the first four lines of Figure 1.
2. A cycle is detected where the current call is a variant of an ancestor call. Clearly the results of the current call are needed to compute the results for a tabled parent call. With `shift/1` this parent computation is suspended. This scenario is covered by the remaining lines of code in Figure 1, where the `reset/3` delimits the parent computation and the suspended computation is stored for later reactivation in the form of a `dependency`. This dependency records the source (= child) and target (=parent) calls alongside the continuation, which turns answers to the former into answers to the latter.
3. The answers for the call are already available in a table datastructure. Instead of recomputing them, they are simply read from the table.

For instance, when first calling `p(X,Y)`, `delim/3` calls `p_aux(X,Y)`. In turn `p_aux(X,Y)` calls `p(X,Z)` which invokes `shift/1`, suspending resolution of the remainder of the rule. The dependency is recorded that the continuation `e(Z,Y)` yields results for `p(X,Y)` given results for `p(X,Z)`. Finally, through backtracking the non-cyclic answers `p(1,2)` and `p(2,3)` are found.

Completion Phase After all alternatives have been exhausted through backtracking, our implementation enters the *completion* phase where it computes a fixed point of stored results and dependencies. To make this more concrete, suppose we have solution $\{X = 1, Z = 2\}$ for `p(X,Z)` as well as the previously mentioned dependency for `p(X,Y)`, and the program contains the fact `e(2,1)`. Then, solving the continuation `e(Z,Y)` using our modified SLD-resolution results in the solution $\{X = 1, Y = 1\}$ for `p(X,Y)`.

In general, resuming a continuation may lead both to new results and new dependencies. Hence, a fixed-point computation is required that stops when no new results or dependencies are generated.

Implementation Support In addition to the two delimited control primitives, our implementation requires support from the Prolog system for mutable terms, non-backtrackable mutations[3] and global variables. These features are available in many Prolog systems and generally easy to add to others.

5 Evaluation

Implementation Effort The control flow part for our tabling implementation comprises 60 LoC, which is about 10% of the whole implementation. This is quite unprecedented and clearly attests to the high-level nature of the approach. The majority of the code is made up by two kinds of data structures: the tables (233 LoC or 40%) and the fixed-point worklists (259 LoC or 45%). Adding 25 lines of glue code, this amounts to an implementation in 577 Prolog LoC.

Performance While raw performance is not the main objective of our lightweight implementation, it is nevertheless important to compare reasonably to the existing state-of-the-art. In order to evaluate this, we compare our implementation in hProlog 3.2.38 against XSB 3.4.0 [Swift and Warren, 2012], B-Prolog 8.1 [Zhou, 2012], Yap 6.3.4 [Santos Costa *et al.*, 2012] and Ciao 1.15-2731-g3749edd [Hermenegildo *et al.*, 2012] on a number of benchmarks.[4] Table 1 summarizes the results (in ms) obtained on a Dell PowerEdge R410 server (2.4 GHz, 32 GB RAM) running Debian 7.6.

Discussion The XSB system is the reference system for tabling; it has invested the most time and resources in the development of its tabling infrastructure. We see that it is 8 to 38 times faster than our implementation, but 45 to 78 times faster for two outliers (path right last: binary tree 18 and 10k pingpong).

[3]Essential to retain the stored answers and dependencies across backtracking.

[4]The description and code of the benchmarks can be found at http://users.ugent.be/~bdsouter/tabling/.

Benchmark	Size	hProlog	$\frac{\text{hProlog}}{\text{XSB}}$	$\frac{\text{hProlog}}{\text{B-Prolog}}$	$\frac{\text{hProlog}}{\text{Ciao}}$
recognize	20,000	205	26	0.003	4
	50,000	503	30	0.001	4
n-reverse	500	767	38	11	45
	1,000	2,800	31	6	34
shuttle	2,000	44	∞	0.1	9
	5,000	138	23	0.08	12
	20,000	582	24	0.02	10
	50,000	1,586	29	0.01	12
ping pong	10,000	271	45	0.07	14
	20,000	490	35	0.03	8
path double first loop	50	653	19	13	7
	100	4,638	17	10	6
path double first	50	162	27	15	14
	100	989	20	12	10
	200	6,785	18	16	10
	500	110,463	25	19	14
path right last: pyramid 500	500	1,914	35	29	27
path right last: binary tree 18	18	108,662	78	50	42
test large joins 2	12	3,001	10	4	12
joins mondial		6,444	8	7	6

Table 1: Results of the performance benchmarks.

B-Prolog is only half as fast as XSB on many benchmarks, but is architecturally different: BProlog implements linear tabling and uses a hashing-based table. Moreover, in several cases B-Prolog is notably slower than XSB (i.e., n-reverse) and even much slower than our own implementation (recognize, shuttle, ping pong). All in all the results are mixed and point out several weaknesses in the B-Prolog implementation compared to our all Prolog implementation.

The performance of Ciao lies between that of XSB and B-Prolog. Performance of our implementation is within a factor 4 to 14 of Ciao, with reverse and path right last as outliers.

The Yap tabling implementation, which is based on that of XSB, is clearly the fastest: the underlying engine is much faster [Rocha *et al.*, 2000]. It outperforms our approach on all benchmarks, and the other systems on most. Six benchmarks take less than $1\,\mathrm{ms}$ (rounded down to $0\,\mathrm{ms}$). We refer the reader to [Desouter *et al.*, 2015] for the detailed Yap timings, which we have left out here for reasons of space.

Summary We consider the performance results of our implementation very reasonable, especially if we take into account the stark contrast between our concise and high-level implementation and the complex integration in other systems.

6 Related Work

Delimited Control Delimited control, well-known in functional programming, has not received much attention in the context of Prolog. Only recently have Schrijvers et al. provided an unobtrusive implementation in the WAM [2013b; 2013a]. In the continuation-passing implementation [Tarau and Dahl, 1994] of BinProlog [Tarau, 2012] this is even easier. Schrijvers et al. also illustrate the power of delimited control by porting various effect handlers [Plotkin and Pretnar, 2013] to Prolog. As far as we know, tabling as a library is the first Prolog-specific application.

Other Tabling Mechanisms XSB [Swift and Warren, 2012] is the best-known Prolog engine supporting tabling. Its foundation, SLG resolution, has been described by Chen and Warren [1996]. It is based on stack freezing, which has required deep changes to the architecture of the WAM.

Linear tabling and DRA [Zhou *et al.*, 2000; Guo and Gupta, 2001; 2004] implement tabled evaluation by stealing choicepoints or reordering alternatives at run time. They also require specific lowlevel WAM changes and have a worse time performance than XSB.

Ramesh and Chen [1994] extend Prolog with tabling primitives implemented in C. Calls to the primitives are introduced in a complex program transformation. More recently, Guzmán et al. [2008] have addressed the performance bottlenecks of Ramesh and Chen's approach using more fine-grained primitives. Hence, the approach does not lower the threshold for adopting tabling.

CAT is an alternative to the SLG-WAM used in XSB [Demoen and Sagonas, 1998a]. Rather than stack freezing, CAT uses incremental copies to preserve the execution state of suspended computations. CHAT is a hybrid between SLG and CAT [Demoen and Sagonas, 1998b]. Both CAT and CHAT acknowledge that the complexity and scope of WAM-changes should be kept limited.

7 Conclusion

We have presented a new high-level implementation of tabling. Our approach is implemented entirely as a Prolog library and requires no deep modifications to the WAM or complex program transformations. It weighs in at less than 600 LoC and, in particular, captures the complex control management of tabling in 60 LoC thanks to delimited control. We believe that the simplicity of this implementation makes tabling more accessible to a wider range of Prolog systems, while still delivering a reasonable performance.

Acknowledgments

We are grateful to Bart Demoen for maintaining hProlog. This work was partly supported by the Flemish Fund for Scientific Research (FWO), and partly by project GRACeFUL, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement № 640954.

References

[Aït-Kaci, 1999] Hassan Aït-Kaci. *Warren's Abstract Machine — a Tutorial Reconstruction*. http://wambook.sourceforge.net/, 1999. Online edition of the 1991 book published by MIT Press.

[Chen and Warren, 1996] Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996.

[Danvy and Filinski, 1990] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of LFP'90*, LFP '90, pages 151–160. ACM, 1990.

[de Guzmán *et al.*, 2008] Pablo Chico de Guzmán, Manuel Carro, Manuel V. Hermenegildo, Cláudio Silva, and Ricardo Rocha. An improved continuation call-based implementation of tabling. In *Proceedings of PADL'08*, volume 4902 of *LNCS*, pages 197–213. Springer, 2008.

[Demoen and Sagonas, 1998a] Bart Demoen and Konstantinos Sagonas. Cat: The copying approach to tabling. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *Principles of Declarative Programming*, volume 1490 of *LNCS*, pages 21–35. Springer, 1998.

[Demoen and Sagonas, 1998b] Bart Demoen and Konstantinos Sagonas. Chat: The copy-hybrid approach to tabling. In Gopal Gupta, editor, *Proceedings of PADL'98*, volume 1551 of *LNCS*, pages 106–121. Springer, 1998.

[Desouter *et al.*, 2015] Benoit Desouter, Marko van Dooren, and Tom Schrijvers. Tabling as a library with delimited control. *Theory and Practice of Logic Programming*, 15(4-5):419–433, 2015.

[Fan and Dietrich, 1992] Changguan Fan and Suzanne Wagner Dietrich. Extension table built-ins for Prolog. *Software: Practice and Experience*, 22(7):573–597, 1992.

[Felleisen, 1988] Mattias Felleisen. The theory and practice of first-class prompts. In *Proceedings of POPL'88*, POPL, pages 180–190. ACM, 1988.

[Guo and Gupta, 2001] Hai-Feng Guo and Gopal Gupta. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In *Proceedings of ICLP'01*, pages 181–196. Springer, 2001.

[Guo and Gupta, 2004] Hai-Feng Guo and Gopal Gupta. An efficient and flexible engine for computing fixed points. *CoRR*, abs/cs/0412041, 2004.

[Hermenegildo *et al.*, 2012] M. V. Hermenegildo, F. Bueno, M. Carro, P. Lípez-García, E. Mera, J. F. Morales, and G. Puebla. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming*, 12(1-2):219–252, January 2012.

[Kowalski, 1974] Robert Kowalski. Predicate logic as programming language. In *Proceedings of International Federation for Information Processing (IFIP)*, pages 569–574, 1974.

[Kowalski, 1979] Robert Kowalski. *Logic for Problem Solving*. North-Holland, 1979.

[Lloyd, 1984] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, New York, 1984.

[Plotkin and Pretnar, 2013] Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4), 2013.

[Ramesh and Chen, 1994] R. Ramesh and Weidong Chen. A portable method for integrating SLG resolution into Prolog systems. In *Proceedings of ILPS'94*, ILPS, pages 618–632, Cambridge, 1994. MIT Press.

[Rocha *et al.*, 2000] Ricardo Rocha, Fernando Silva, and Vítor Santos Costa. YapTab: A tabling engine designed to support parallelism. In *Conference on Tabulation in Parsing and Deduction*, pages 77–87, 2000.

[Santos Costa *et al.*, 2012] Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The YAP Prolog system. *Theory and Practice of Logic Programming*, 12(1-2):5–34, 2012.

[Schrijvers *et al.*, 2013a] Tom Schrijvers, Bart Demoen, and Benoit Desouter. Delimited continuations in Prolog: Semantics, use and implementation in the WAM. Report CW 631, Dept. of Computer Science, KU Leuven, 2013.

[Schrijvers *et al.*, 2013b] Tom Schrijvers, Bart Demoen, Benoit Desouter, and Jan Wielemaker. Delimited continuations for Prolog. *Theory and Practice of Logic Programming*, 13(4-5):533–546, 2013.

[Swift and Warren, 2012] Terrance Swift and David S. Warren. XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming*, 12(1-2):157–187, January 2012.

[Tarau and Dahl, 1994] Paul Tarau and Veronica Dahl. Logic programming and logic grammars with first-order continuations. In *Proceedings of LOPSTR '94*, volume 883. Springer, June 1994.

[Tarau, 2012] Paul Tarau. The BinProlog experience: Architecture and implementation choices for continuation passing Prolog and first-class logic engines. *Theory and Practice of Logic Programming*, 12(1-2):97–126, 2012.

[Warren, 1983] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI, 1983.

[Zhou *et al.*, 2000] Neng-Fa Zhou, Yi-Dong Shen, Li-Yan Yuan, and Jia-Huai You. Implementation of a linear tabling mechanism. In *Practical Aspects of Declarative Languages*, volume 1753 of *LNCS*, pages 109–123. Springer, 2000.

[Zhou, 2012] Neng-Fa Zhou. The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):189–218, 2012.