On Applying Program Transformation to Implement Suspension-Based Tabling in Prolog (Extended Abstract)

Ricardo Rocha, Cláudio Silva, and Ricardo Lopes*

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

Tabling is a technique of resolution that overcomes some limitations of traditional Prolog systems in dealing with redundant sub-computations and recursion. We can distinguish two main categories of tabling mechanisms: *suspension-based tabling mechanisms* and *linear tabling mechanisms*. Suspension-based tabling mechanisms need to preserve the state of suspended tabled subgoals in order to ensure that all answers are correctly computed. A tabled evaluation can be seen as a sequence of sub-computations that suspend and later resume. On the other hand, linear tabling mechanisms use iterative computations of tabled subgoals to compute fix-points. The main idea of linear tabling is to maintain a single execution tree where tabled subgoals always extend the current computation without requiring suspension and resumption of sub-computations.

A common approach used to include tabling support into existing Prolog systems 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 and require changing important components of the system such as the compiler, the code generator, and the data structures that support Prolog execution. A different approach to incorporate tabled evaluation into existing Prolog systems is to apply source level transformations to a tabled program. The transformed program then uses external tabling primitives that provide direct control over the search strategy to implement tabled evaluation. This idea was first explored by Fan and Dietrich [1] 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 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 to build external Prolog modules implementing the support for tabled evaluation. We can distinguish two main modules in our implementation: the module that implements the specific control primitives and the module that implements the table space data structures. The table space was implemented using *tries* [2]. To implement our mechanism, that we

* 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.

named *tabled evaluation with continuation calls*, we followed a *local scheduling* strategy [3]. 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. To implement the program transformation step, we have extended the original program transformation module of Ramesh and Chen [4] to include the tabling primitives for our approach.

To evaluate the impact of our approach, we ran it against the state-of-the-art YapTab system, that implements tabling support at the low-level engine. YapTab also implements a suspension-based mechanism, uses tries to implement the table space and is implemented on top of Yap. This was 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. As expected, YapTab outperformed our mechanism in all programs tested. On average, YapTab was about 7.50 faster than the continuation calls mechanism. Best performance was achieved for left recursive tabled predicates with the recursive clause first, with an average overhead between 2 and 3. The results obtained also suggested that there is a cost in the execution time that is proportional to the number of redundant answers, variant 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 because continuation calls are not compiled, they are constructed and called in run-time using the C language interface.

Considering that Yap and YapTab are respectively two of the fastest Prolog and tabling engines currently available, the results obtained are very interesting and very promising. 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. Moreover, 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.

References

1. Fan, C., Dietrich, S.: Extension Table Built-Ins for Prolog. Software Practice and Experience **22** (1992) 573–597
2. 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
3. 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
4. Ramesh, R., Chen, W.: Implementation of Tabled Evaluation with Delaying in Prolog. IEEE Transactions on Knowledge and Data Engineering **9** (1997) 559–574