Towards a Generic Interface to Integrate CLP and Tabled Execution (Extended Abstract)

Joaquín Arias[1]

Manuel Carro[1,2]

joaquin.arias@imdea.org, manuel.carro@{imdea.org,upm.es}
[1] *IMDEA Software Institute,* [2] *Technical University of Madrid*

submitted 29 April 2015; accepted 5 June 2015

Logic programming systems featuring Constraint Logic Programming (Jaffar and Maher 1994) and tabled execution (Tamaki and Sato 1986; Warren 1992) have been shown to increase the declarativeness and efficiency of Prolog, while at the same time making it possible to write very expressive programs. Previous implementations fully integrating both capabilities (i.e., forcing suspension, answer subsumption, etc. where it is necessary in order to avoid recomputation and terminate whenever possible) did not have a simple, well-documented, easy-to-understand interface which made it possible to integrate arbitrary CLP solvers into existing tabling systems. This clearly hinders a more widespread usage of this combination.

In our work, we examine the requirements that a constraint solver must fulfill to be easily interfaced with a tabling system. We propose a minimal set of operations which the constraint solver has to provide to the tabling engine. These operations are based in only four objects (Vars, Dom, ProjStore and Store). Vars is a list with the constrained variables of a call. Dom and ProjStore are the representation of the projection of the constraint store of a call. Store is the representation of the constraint store of a generator and is used by the external constraint solver to reinstall it when the generator is complete.

The two main operations to be provided by the solver are: (i) entailment, entail(+Vars_A, +Dom_A, +Dom_B, +$\mathsf{ProjStore}_B$), which checks if the call/answer constraint store (Vars_A and Dom_A) is entailed by the previous call/answer constraint store (Dom_B and $\mathsf{ProjStore}_B$) and (ii) projection, executed in two steps: project_domain(+Vars, -Dom), that pre-computes an object (Dom) used during the entailment, and project_gen_store(+Vars, +Dom, -ProjStore), which is executed when the entailment fails.

The compiler performs a shallow program transformation adding tabled_call/1 to control the tabled execution and new_answer/0 to collect the answers. These two predicates invoke the operations of the interface during tabled execution. Fig. 1 contains a Prolog version of tabled_call/1 which specifies its implementation and the control flow of the execution. This specification shows that when a new call is entailed by a previous generator, its execution is suspended, unlike in usual tabling, where suspension happens only when variant calls are found.

We validate experimentally our design with three use cases. First we re-engineer a

```
tabled_call(Call) :-
 (
  lookup_table(Call, Gen, Vars)
 ;
  save_generator(Call, Gen, Vars)
 ),
 project_domain(Vars, Dom),
 (
  member(Lgen,~l_generators(Gen)),
  entail(Vars, Dom,~dom(Lgen),~projStore(Lgen))->
  suspend_consumer(Call)
 ;
  current_store(Store),
  project_gen_store(Vars, Dom, ProjStore),
  save_Lgen(Gen, Lgen, Store, Dom, ProjStore),
  push(Lgen),
  execute_generator(Call)
 ),
 consume_answer(Ans,~answers(Lgen)),
 member(Lans,~l_answers(Ans)),
 reinstall_store(~store(Lgen)),
 apply_answer(Vars,~dom(Lans),~projStore(Lans)).
```

Fig. 1. Prolog version of Tabled_call/1.
Notation: p(~dom(Lgen)) ≡ dom(Lgen, DomLgen), p(DomLgen).

previously existing tabled constrain domain (**difference constraints** (Chico de Guzmán et al. 2012)) in Ciao (Hermenegildo et al. 2012). This solver is implemented in C, so the arguments of the interface represent the memory address of C structures. Then we integrate **Holzbauer's CLP(Q)** (Holzbaur 1995; Holzbaur 1992) implementation with Ciao Prolog's tabling engine. In this case existing CLP(Q) predicates already provide the necessary functionality so we only need to write simple bridge predicates (see Fig. 2).

```
project_domain(_, _).
project_gen_store(V, _, (F, St)) :-
        clpqr_dump_constraints(V, F, St).
project_answer_store(V, _, (F, St)) :-
        clpqr_dump_constraints(V, F, St).
entail(V, _, _, (V, St)) :-
        clpq_entailed(St).

current_store(_).
reinstall_store(_, _, _).
apply_answer(V, _, (V, St)) :-
        clp_meta(St).
```

Fig. 2. Interface for CLP(Q).

With these two constraints solvers we evaluate on one hand the cost of adopting a more modular framework versus the previous non-modular implementation of difference constraints, and on the other hand we highlight the benefits of being able to interface easily more constraint solvers: using TCLP(Q) gives more expressiveness and in some cases better performance that TCLP(Diff) (see results of the reverse Fibonacci benchmarks in Table 1) since by using TCLP(Q) we can write programs in way which exploits better the advantages of constraint programming.

Last, we implement a **constraint solver over (finite) lattices** that is parametrized by the lattice domain. The lattice domain defines the elements and its operations, including at least join and meet, which define the partial order ($\sqsubseteq$) relation used to check en-

	Diff Constraints			CLP(Q)	
	CLP	TCLP	Mod TCLP	CLP	Mod TCLP
fibonacci(P, 89)	494	11	13	67	12
fibonacci(P, 610)	–	21	25	628	20
fibonacci(P, 4181)	–	36	42	6001	30
fibonacci(P, 28657)	–	56	69	153813	40
fibonacci(P, 196418)	–	85	111	> 5 min.	53
fibonacci(P, 832040)	–	113	158	> 5 min.	64

Table 1. *Run time results (in ms.) for the fibonacci/2 program in two versions.*

tailment. To evaluate this constraint solver in the context of tabled execution, we implemented a simple abstract analyzer whose fix-point is reached by means of tabled execution. Its domain operations are implemented using the lattice domain and the constraint solver, which avoids recomputation of subsumed abstractions and attains better accuracy and considerable speedups. We evaluate its performance by comparing this implementation with an abstract interpreter without the constraint solver. Table 2 shows the results in terms of execution time of the analysis of a program which is parametrized by the number of arguments.

	Tabling	Mod TCLP
permute/10	2788	3
permute/8	563	2
permute/6	112	2
permute/4	21	1

Table 2. *Run time results (in ms.) for* `?- analyze(permute(A1, ... , An), P).`

References

CHICO DE GUZMÁN, P., CARRO, M., HERMENEGILDO, M., AND STUCKEY, P. 2012. A General Implementation Framework for Tabled CLP. In *FLOPS'12*, T. Schrijvers and P. Thiemann, Eds. Number 7294 in LNCS. Springer Verlag, 104–119.

HERMENEGILDO, M. V., BUENO, F., CARRO, M., LÓPEZ, P., MERA, E., MORALES, J., AND PUEBLA, G. 2012. An Overview of Ciao and its Design Philosophy. *TPLP 12,* 1–2, 219–252. http://arxiv.org/abs/1102.5497.

HOLZBAUR, C. 1992. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In *1992 International Symposium on Programming Language Implementation and Logic Programming*. LNCS 631, Springer Verlag, 260–268.

HOLZBAUR, C. 1995. OFAI clp(q,r) manual, edition 1.3.3. Tech. Rep. TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna.

JAFFAR, J. AND MAHER, M. 1994. Constraint Logic Programming: A Survey. *Journal of Logic Programming 19/20*, 503–581.

TAMAKI, H. AND SATO, M. 1986. OLD Resolution with Tabulation. In *Third International Conference on Logic Programming*. Lecture Notes in Computer Science, Springer-Verlag, London, 84–98.

WARREN, D. S. 1992. Memoing for Logic Programs. *Communications of the ACM 35*, 3, 93–111.