Description and Evaluation of a Generic Design to Integrate CLP and Tabled Execution

Joaquín Arias
IMDEA Software Institute
joaquin.arias@imdea.org

Manuel Carro
IMDEA Software Institute
Technical University of Madrid
manuel.carro@{imdea.org,upm.es}

ABSTRACT

Logic programming systems with tabling and constraints (TCLP, *tabled constraint logic programming*) have been shown to be more expressive and in some cases more efficient than those featuring only either tabling or constraints. Previous implementations of TCLP systems which use entailment to determine call / answer subsumption did not provide a simple, uniform, and well-documented interface to facilitate the integration of additional constraint solvers in existing tabling systems, which would increase the application range of TCLP. We present the design and an experimental evaluation of Mod TCLP, a framework which eases this integration. Mod TCLP views the constraints solver as a client of the tabling system. The tabling system is generic w.r.t. the constraint solver and only requires a clear, small interface from the latter. We validate our design by integrating four constraint solvers: a re-engineered version of a previously existing constraint solver for difference constraints, written in C; the standard versions of Holzbauer's CLP(Q) and CLP(R), written in Prolog; and a new constraint solver for equations over finite lattices. We evaluate the performance of our framework in several benchmarks using the aforementioned constraint solvers. All the development work and evaluation was done in Ciao Prolog, a robust, mature, next-generation Prolog system.

Keywords

Constraints, Tabling, Prolog, Interface, Implementation.

1. INTRODUCTION

Constraint Logic Programming (CLP) [12] extends Logic Programming (LP) with variables which can belong to arbitrary constraint domains and the ability to incrementally solve the equations involving these variables. CLP brings additional expressive power to LP, since constraints can very concisely capture complex relationships between variables. Also, shifting from "generate-and-test" to "constraint-and-generate" patterns reduces the search tree and therefore brings additional performance, even if constraint solving is in general more expensive than unification.

Tabling [23, 26] is an execution strategy for logic programs which suspends repeated calls which would cause infinite loops. Answers from other, non-looping branches, are used to resume suspended calls which can in turn generate more answers. Only new answers are saved, and evaluation finishes when no new answers can be generated. Tabled evaluation always terminates for calls / programs with the bounded term depth property and can improve efficiency for programs which repeat computations, as it automatically implements a variant of dynamic programming. Tabling has been successfully applied in a variety of contexts, including deductive databases, program analysis, semantic Web reasoning, and model checking [27, 4, 29, 18].

The combination of CLP and tabling [25, 20, 3, 2] brings several advantages. It improves termination properties and increases speed in a range of programs. It has been applied in several areas, including constraint databases [15, 25], verification of timed automata and infinite systems [1], and abstract interpretation [24].

The theoretical basis of TCLP [25] were established in the framework of bottom-up evaluation of Datalog systems and presents the basic operations (projection and entailment checking) that are necessary to ensure completeness w.r.t. the declarative semantics. However, some previous implementations did not fully include these two operations [20, 3], allegedly due to performance issues and also to the implementation difficulty.

On the other hand, previous TCLP frameworks featuring a complete treatment of constraint projection and entailment [2] focused on adapting a tabling algorithm and its implementation to be used with constraints. As a result, and although being generic frameworks, they were are not easily extensible. Adding new constraint domains to them is a difficult task that requires deep knowledge about the particular tabling implementation and the constraint solver. The modifications done to the tabling implementation for one particular constraint solver may very well be difficult to adapt to another constraint solver; in turn, constraint solvers had to be modified in order to make then aware of data structures and capabilities of the tabling algorithm. These drawbacks generate a *technical debt* that makes using the full potential of TCLP very difficult.

In this work we, on one hand, generalize the design of a tabling implementation so that it can use the projection and entailment operations provided by a constraint solver[1] presented to the tabling engine as a *server*. On the other hand, we define a set of operations that the constraint solver has to provide to the tabling engine. These operations are natural to the constraint solver, and when they are not already present, they should be easy to implement.

We have validated our design (termed Mod TCLP) with an im-

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

[1]Note that parts of the algorithm, such as the SCC-based completion detection procedure, remain untouched.

plementation in Ciao Prolog [10][2] where we interfaced four non-trivial constraint solvers to provide four different TCLP systems. We have experimentally evaluated these implementations with programs exploiting TCLP execution on several benchmarks.

2. BACKGROUND AND MOTIVATION

In order to highlight some of the advantages of TCLP versus LP, tabling, and CLP with respect to declarativeness and logical reading, we will compare the behavior of these paradigms and strategies using different versions of a program to compute distances between nodes in a graph. Each version will be adapted to a different paradigm, but trying to stay as close as possible to the original code, so that the additional expressiveness can be attributed to the semantics of the programming language and not to differences in the code itself. It is worth noting that with enough programming effort, a program written in Prolog (or any other language) can exhibit the behavior of a tabled, CLP, or TCLP program, but we precisely want to avoid having to spend resources in making this costly and bug-prone adaptation for every program.

```
dist(X, Y, D) :-
    dist(X, Z, D1),
    edge(Z, Y, D2),
    D is D1 + D2.
dist(X, Y, D) :-
    edge(X, Y, D).
```

```
:- use_package(clpq).

dist(X, Y, D) :-
    D1 #> 0, D2 #> 0,
    D #= D1 + D2,
    dist(X, Z, D1),
    edge(Z, Y, D2).
dist(X, Y, D) :-
    edge(X, Y, D).
```

Figure 1: Versions of distance in a graph: Prolog (left) and CLP (right).

2.1 LP vs. CLP

The initial code in Fig. 1, left is the Prolog version of a program to find nodes in a graph within a distance K from each other.[3] Fig. 1, right, is the CLP version of the same code where we have moved the constraint which expresses distance addition to the beginning of the body and added constraints stating that distances are assumed to be non-negative. In order to find the nodes X and Y within a distance K from each other we use the queries ?- dist(X,Y,D), D $<$ K. and ?- D #$<$ K, dist(X,Y,D). in Prolog and CLP, respectively. None of these two queries terminates due to the left recursion for graphs with or without cycles.

If we convert the program to a right-recursion version by just swapping the calls to edge/3 and dist/3, the LP execution will still not terminate in a graph with cycles. This conversion is easy in this case, but in other cases, such as certain parsing algorithms or language interpreters, left recursion is much more natural and the conversion to right recursion requires extra arguments to implement stacks — precisely the kind of program adaptation that we would like to avoid. The right-recursive version of the CLP program terminates because the initial distance bound eventually causes an inconsistency in the constraint store and provokes a failure in the search. In other words, the nodes explored can be at most K distance units apart. In order to ensure that this is a real bound we need a lower bound to the distance between two nodes, and hence the constraints on D1 and D2. If that constraint were not there, the possibility of a *negative distance* would make it impossible to set the aforementioned bounds. This behavior is summarized in columns "LP" and "CLP" of Table 1.

[2]Stable versions of Ciao Prolog are available at http://www.ciao-lang.org. The Ciao Prolog version and benchmarks used in this paper are available at http://goo.gl/vWRV15.

[3]This is a typical query for the analysis of social networks [21].

	LP	CLP	TAB	TCLP	
Left rec.	×	×	✓	✓	Without cycles
Right rec.	✓	✓	✓	✓	
Left rec.	×	×	×	✓	With cycles
Right rec.	×	✓	×	✓	

Table 1: Comparison of termination properties.

2.2 LP vs. Tabling

Tabling records the first occurrence of a tabled predicate call (the *generator*) and its answers. In variant tabling (the most usual form of tabling) when a call equal to a previous generator up to variable renaming is found, its execution is suspended, and it is marked as a *consumer* of the generator . When a generator finitely finishes exploring all of its clauses and its answers are collected, its consumers are resumed and are fed the generator's answers. This may make consumers to produce answers which may resume them again.

Tabling is complete for programs with the bounded term-depth property (which have a finite model), no matter whether they are left or right recursive. Therefore, left- or right-recursive *reachability* terminates in finite graphs with or without cycles. However, the program in Fig. 1, left, has an infinite model for cyclic graphs: every cycle can be traversed an unbound number of times, giving rise to an unlimited number of answers with a different distance each. The query ?- dist(X,Y,D), D $<$ K. will therefore not terminate under variant tabling.

Operationally, if *local scheduling* [5] is used, the tabling engine tries to find all the answers (maybe infinitely many) before returning them. If *batch scheduling* [5] is used, some answers may be returned but the tabling engine will not finitely terminate, as it will eventually try to unsuccessfully generate new answers under the distance bound K. Note that ignoring the distance argument for tabling (which some systems can do [28] makes tabling to keep the first solution found, but it breaks completeness.

2.3 TCLP vs. Tabling and CLP

Under our TCLP framework, the program in Fig. 1, right, can be executed with tabling and using constraint entailment to suspend more particular calls and to store only the more general answers. These enhance termination properties and speed, as we will see in Section 4.1. Using constraint entailment in TCLP is related to using subsumption in regular tabling [21], which was also shown to enhance termination and performance. The column "TCLP" of Table 1 summarizes the termination properties of dist/3 under TCLP, and shows that a full integration of tabling and CLP makes it possible to find all the solutions and finitely terminate in all the cases. To execute the CLP program under TCLP it is only necessary to import the package with the tabled constraint solver interface instead of the regular constraint solver.

In order to understand why entailment is advantageous, we will very summarily describe how our implementation works. We assume some knowledge about tabling. Under our implementation, a generator *Gen* is identified by a tuple $\langle C_{gen}, ProjStore_{gen}\rangle$ where C_{gen} is the call pattern and $ProjStore_{gen}$ is the projection of the current constraint store onto the constrained variables of the call. A call $\langle C_{call}, ProjStore_{call}\rangle$ is a consumer of *Gen* if C_{call} is a variant call of C_{gen} and the constraint store $ProjStore_{call}$ is entailed by $ProjStore_{gen}$. In that case $\langle C_{call}, ProjStore_{call}\rangle$ can be suspended because every answer Ans_{call} in its answer set is at least as particular as an answer in the set of answers Ans_{gen} of the generator:

$$\langle C_{call}, ProjStore_{call}\rangle \sqsubseteq \langle C_{gen}, ProjStore_{gen}\rangle$$
$$\Longrightarrow variant(C_{call}, C_{gen}) \wedge ProjStore_{call} \sqsubseteq ProjStore_{gen}$$
$$\Longrightarrow Ans_{call} \sqsubseteq_S Ans_{gen}$$

where $S_1 \sqsubseteq_S S_2$ iff $\forall c_1 \in S_1\ \exists\, c_2 \in S_2\ .\ c_1 \sqsubseteq c_2$

In the Mod TCLP implementation, an answer is identified by a tuple $\langle S_{ans}, ProjStore_{ans}\rangle$, where S_{ans} is a variable substitution and $ProjStore_{ans}$ is the projection of the constraint store onto the set of variables constrained when the answer is collected. The answer Ans_0 is more particular that the answer Ans_1 if:

$$\langle S_{ans_0}, ProjStore_{ans_0}\rangle \sqsubseteq \langle S_{ans_1}, ProjStore_{ans_1}\rangle$$
$$\Longrightarrow variant(S_{ans_0}, S_{ans_1}) \wedge ProjStore_{ans_0} \sqsubseteq ProjStore_{ans_1}$$

When the consumers are resumed, as in tabling, they are fed with answers from its generator. Unlike in variant tabling, where these answers are a substitution which can be directly applied to the consumers (both have variables in the same position), in TCLP with entailment not all the generator answers are be valid for its consumers, since a consumer could be more particular than its generator due to the constraints active when it was called.

So in order to check if an answer is valid for a consumer, the tabling engine first directly applies the variable substitution (they also have variables in the same position), and then calls the constraint solver to calculate the resulting constraint store $Store_{res}$ once the projected constraint store of the answer $ProjStore_{ans}$ is added to the constraint store of the consumer $Store_{cons}$

$$Store_{res} = Store_{cons} \wedge ProjStore_{ans}$$

If $Store_{res} = \bot$, that answer is not valid since the resulting constraint store is not consistent, otherwise the execution continues with $Store_{res}$ as the consumer constraint store.

Before an answer *Ans* produced by the generator *Gen* is added to its answer set Ans_{gen}, a double entailment check is executed to discard *Ans* if there is a more general answer in Ans_{gen} or to remove from Ans_{gen} the answers which are more particular than *Ans*. This process takes time, but it brings advantages from saving resumptions: when an answer is added to a generator, consumers are resumed by that answer. These consumers generate more answers and cause further resumptions in cascade. Reducing the number of redundant answers reduces the number of redundant resumptions and execution time.

2.4 Some Application Fields for TCLP

Facilitating the integration of tabling and constraint solvers makes it possible to exploit the synergy between them in several application fields, already mentioned in the literature, of which we highlight a few:

Abstract interpretation: Tabling can be use not only to reach the fixpoint [14, 13] but also by implementing the abstract domain operations as a constraint system, the entailment will automatically detect more particular calls and suspend execution to reuse answers from most general ones. Constraints can also be used to state *preconditions* to the analysis results before the analysis starts. These preconditions can propagate and solve some verification problems faster. In Section 4.3 we show and evaluate an example.

Reasoning on ontologies: An ontology formalizes types, properties, and interrelationships of entities. These can be expressed as a constraint system and with TLCP, evaluation in ontologies can benefit from entailment of instances which are more particular than other entities in a similar fashion to OWL [9] but in richer domains.

Constraint-based verification: Verification rules can be encoded as constraint systems, and the tabling engine can use the entailment to guaranty termination and saves execution time.

store_projection(+Vars, -ProjStore) Returns in ProjStore the projection of the current constraint store onto the list of variables Vars and extra information needed in the others phases.

call_entail(+Vars, +ProjStore, +ProjStore$_{gen}$) Success if the projection of current constraint store, ProjStore, is entailed by the projected store, ProjStore$_{gen}$, of a previous generator. Otherwise it fails.

answer_compare(+Vars, +ProjStore, +ProjStore$_{ans}$, -Res) Returns Res='$\leq$' if the projected store of the current answer, ProjStore, is entailed by the projected store of a previous answer, ProjStore$_{ans}$ or Res='$>$' if ProjStore entails ProjStore$_{ans}$ but ProjStore $\neq$ ProjStore$_{ans}$. Otherwise it fails.

apply_answer(+Vars, +ProjStore) Adds the projected constraint store ProjStore of the answer to the current constraint store and success if the resulting constraint store is consistent.

current_store(+Vars, +ProjStore, -CuSt) Returns in CuSt the current constraint store.

reinstall_store(+Vars, +CuSt) Reinstall the original constraint store CuSt.

Figure 2: Generic interface specification.

We note that these application examples may need to use different constraint systems. Even more, for the abstract interpreter every domain is a different constraint system. Therefore here is where the power of making it easy to plug-in different constraint solvers is relevant.

3. THE GENERIC INTERFACE

In this section we describe the interface operations, the program transformation, and the implementation which together enable tabled execution with constraints. We illustrate them with a flowchart and a trace of a TCLP execution.

3.1 Design of the Generic Interface

The basic generic interface (Fig. 2) specifies the predicates that a constraint solver has provide to enable the tabling engine to interact with it in our framework. Four of these predicates correspond to three main operation in constraint systems:

Projection The projection of a constraint store CSt onto a set of variables V $\subseteq$ vars(CSt) is another constraint store PSt involving only variables in V such that any solution of CSt is also a solution of PSt and a valuation over V which is a solution of PSt is a partial solution of CSt.
The predicate store_projection/2 should return an object representing the projection of the current constraint store onto the variables in the call plus the information which the constraint solver will need later to reinstall these constraints. For example, since the tabling engine will recover this term with a variable renaming, it is necessary to be able to identify to which variables in the then-current constraint store correspond the variables in the recovered projection.

Entailment A constraint store St_1 is entailed by a constraint store St_2, ($St_1 \sqsubseteq St_2$), if any solution of St_1 is also a solution of St_2. Predicate call_entail/3 is used to check if the constraint store of the current call is entailed by the constraint store

of a previous generator ($Call \sqsubseteq Gen$), and fails otherwise. Predicate answer_compare/4 is used to check entailment in both directions: it returns $\leq$ in its last argument when $Call \sqsubseteq Gen$ or $>$ when $Call \sqsupset Gen$, and fails otherwise.

Add answers apply_answers/2 is used to add the projected constraint store corresponding to the variables contained in an answer to the current constraint store. If the resulting constraint store is consistent, it will succeed, and fail otherwise.

The last two predicates defined in the interface, current_store/3 and reinstall_store/2, are necessary in the cases where the tabling engine (and the underlying Prolog machinery) cannot restore the constraint store state on backtracking to retrieve the answers to the generator. This can be the case for constraint solvers provided by external libraries which can not use memory areas managed by the tabling engine, and a representation of the external constraint store needs to be saved and retrieved explicitly (as in the case of $D_{\leq}$, see Section 4.1). This makes it possible to handle the objects managed by the solver in a separate, transparent way.

3.2 Implementation Sketch

We describe now the implementation of Mod TCLP, including the global table where generators, consumers, and answers are saved, the transformation we perform to the tabled predicates, and a (simplified) flowchart highlighting how the predicates from the interface are integrated in the execution of the tabling engine.

Global Table.

Tries are the data structure of choice for the call / answer global table [19]. In variant tabling every generator C_{gen} is uniquely associated, modulo variable renaming, to a leaf from where the substitution for every answer hangs. In TCLP, however, generators are identified using as well the constraint store at call time — more precisely, the projection on the variables of the generator. Thus we represent generators with a tuple $\langle C_{gen}, ProjStore_{gen} \rangle$, and in every leaf we have (i) the call pattern C_{gen} and (ii) a list pointing to the frames of the generators having C_{gen} as call pattern. Each frame identifies: (a) the projected constraint store ($ProjStore_{gen}$), (b) the answer table with the answers of this generator, and (c) a list which maintains the consumers of this generator.

Similarly, answers are not merely a Herbrand substitution, but also constraints attached to the variables in the answer. Different answers can have the same substitution but different constraints. The answer table is then a trie where each leaf identifies a variable substitution S_{ans} and for each of these substitutions, S_{ans} a list of the projected constraint stores $ProjStore_{ans}$ of the answer.

Program Transformation.

Fig. 3 shows the transformation of the predicate dist/3 from the program in Fig. 1 (right). The original entry point to the predicate is rewritten to call an auxiliary predicate through the tabled_call/1 meta-predicate. The auxiliary predicate corresponds to the original one with a renamed head and with an additional new_answer/0 at the end of the body to collect the answers. We illustrate how the execution internally proceeds in the next section.

```
dist(A, B, C) :-    tabled_call(dist_aux(A,B,C)).

dist_aux(X, Y, D) :- D1 #> 0, D2 #> 0, D #= D1 + D2,
                     dist(X, Z, D1), edge(Z, Y, D2),
                     new_answer.
dist_aux(X, Y, D) :- edge(X, Y, D),
                     new_answer.
```

Figure 3: Transformation of dist/3.

Execution Flow.

Figs. 5 and 6 show the implementations of tabled_call/1 and new_answer/0. In our implementation these predicates are written in C for performance reasons, but for conciseness and to help understanding we show a Prolog version. This translation generalizes previous tabling algorithms to deal with constraints and related operations. This adaptation is transparent to the constraint solver, which only has to provide the interface functionality.

Fig. 4 uses a (simplified) flowchart to show how execution of a tabled call proceeds. We explain below the steps, following each of the labels in the Figure.

A call to a tabled predicate Call starts the tabled execution invoking **tabled_call/1**, which takes the control of the execution and performs the following steps:

1. lookup_table/3 returns in Gen the variant call pattern of Call and in Vars the list of variables with constraints that appear in the call.
2. The tabling engine calls store_projection/2 in the constraint solver. It returns the projection of the current constraint store onto Vars in ProjStore.
3. The tabling engine retrieves, using member/2, a projected constraint store from the list of frames of Gen. If it succeeds, ProjStore_G contains the projected constraint store and the execution continues in step 6. Otherwise it fails, which happens because all frames have been retrieved or because Gen is the first occurrence of this call pattern.
4. If the constraint solver implements current_store/3, it returns a representation of the current constraint store in CuSt.
5. The tabling engine calls save_generator/4 to add a frame identifying a new generator to the list of frames of Gen. In this new frame, the projected store ProjStore_G of the new generator and the current constraint store CuSt (if available), are also saved. The answer trie and the consumers list are initialized and the execution continues in the step 8.
6. The constraint solver checks if the current store ProjStore is entailed by the retrieved projected constraint store ProjStore_G using call_entail/3. If it is the case, Call is suspended in step 7. Otherwise the tabling engine tries to retrieve another projected constraint store in the step 3.
7. If the generator is not complete, the tabling engine suspends the execution of Call with suspend_consumer/1 and adds Call to the list of consumers of the generator <Gen, ProjStore_G>. Execution then continues by backtracking over the youngest generator. Otherwise Call continues the execution in step 16. A suspended consumer is resumed when its generator produces a new answer, and continues the execution in step 16 also.
8. The generator <Gen, ProjStore_G> is executed with execute_generator/2 which calls the renamed tabled predicate. The execution of **new_answer/0** means that a new answer has been found and execution continues in step 9.
9. The tabling engine returns the variable substitution of the current answer in Ans and the list of (constrained) variables that appeared in the call and are now/still (constrained) variables in Vars using lookup_table/1.
10. The tabling engine invokes store_projection/2. This returns in ProjStore the projection of the current constraint store onto the variables in the answer, Vars.
11. The tabling engine retrieves projected constraint stores from the corresponding list of Ans one at a time using member/2. If it succeeds the projected constraint store is returned in ProjStore_A, and the execution continues in step 13. If it fails the execution continues in step 12. Failure can happen because all

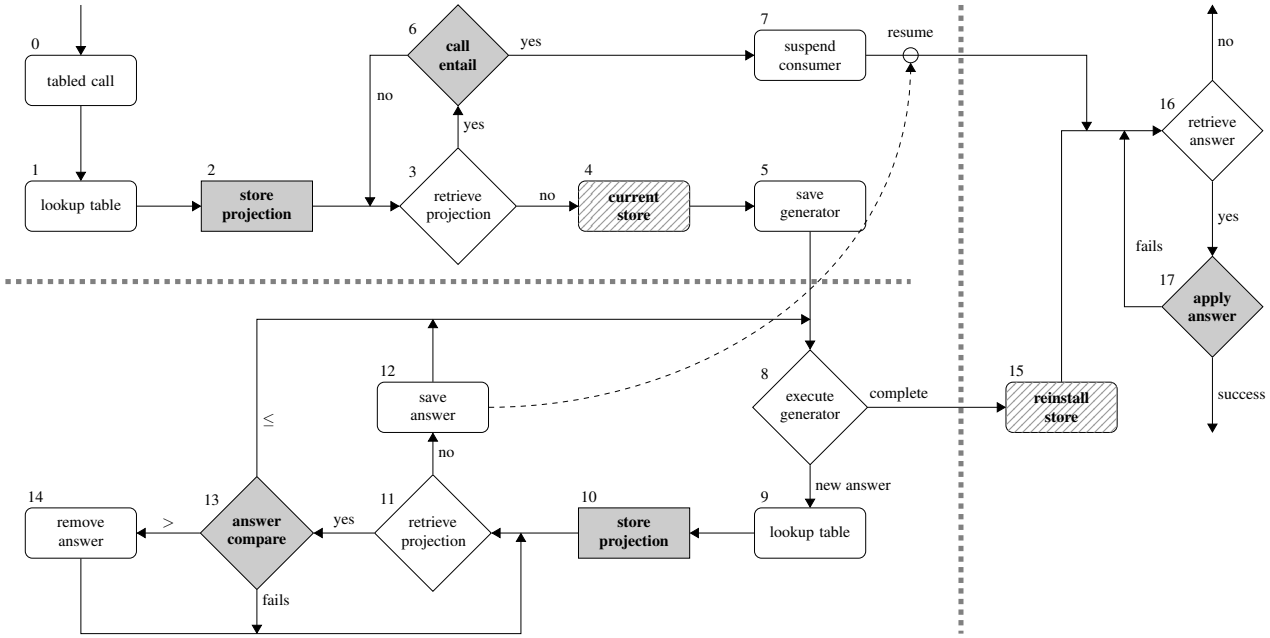

Figure 4: Flowchart of the execution algorithm of Mod TCLP.

```
tabled_call(Call) :-
    lookup_table(Call, Vars, Gen),
    store_projection(Vars, ProjStore),
    (
       member(ProjStore_G, ~projstore_Gs(Gen)),
       call_entail(Vars, ProjStore, ProjStore_G) ->
       suspend_consumer(Call)
    ;
       current_store(Vars, ProjStore, CuSt),
       save_generator(Gen, ProjStore_G, ProjStore, CuSt),
       execute_generator(Gen, ProjStore_G),
       reinstall_store(Vars, CuSt)
    ),
    member(Ans, ~answers(Gen, ProjStore_G)),
    member(ProjStore_A, ~projstore_As(Ans)),
    apply_answer(Vars, ProjStore_A).
```

Figure 5: Implementation of tabled_call/1.
Note: p(~q(X)) ≡ q(X,Y), p(Y), with Y a fresh variable.

```
new_answer :-
    lookup_table(Vars, Ans)
    store_projection(Vars, ProjStore),
    (
      member(ProjStore_A, ~projstore_As(Ans)),
      answer_compare(Vars, ProjStore, ProjStore_A, Res),
      (
          Res == '=<'
       ;
          Res ==  '>',
          remove_answer(ProjStore_A), fail
      ), !
    ;
      save_answer(Ans, ProjStore)
    ), !,
    fail.

new_answer :-
    complete.
```

Figure 6: Implementation of new_answer/0.
Note: p(~q(X)) ≡ q(X,Y), p(Y), with Y a fresh variable.

projected constraint stores were already retrieved from the list or because Ans is the first occurrence of this variable substitution.

12. The tabling engine adds ProjStore to the list of projected constraint stores of Ans with save_answer/2, and resumes, one by one, the consumers of the current generator which were suspended in step 7. Since new_answer/0 always fails, the execution backtracks to complete the execution of the generator (step 8).
13. The constraint solver checks if the current store (ProjStore) is entailed by the retrieved projected constraint store (ProjStore_A) using answer_compare/4. If this is the case, it returns Res = '=<', which makes new_answer/0 discard the current answer, and the generator is re-executed in step 8. If ProjStore entails ProjStore_A but they are not equal, it returns Res = '>' and ProjStore_A is removed in step 14. Otherwise it fails and the execution continues in step 11, where the tabling engine tries to retrieve another projected constraint store.
14. The tabling engine flags the entailed answer as removed using remove_answer/1. Then the execution continues in step 11.
15. Once the generator has exhausted all the answers and does not have more dependencies, it is marked as complete using complete/0. When the interface implements reinstall_store/2, the constraint solver updates the current constraint store by reinstalling CuSt.
16. The tabling engine retrieves answers <Ans, ProjStore_A> from the generator <Gen, ProjStore_G> using member/2. If it succeeds and the answer is not flagged as removed, the answer will be applied in step 17. Otherwise the execution backtracks to retrieve another answer.
17. Applying the variable substitution Ans always succeeds, because the generator and its consumers have the same call pattern. Then the constraint solver adds the projected constraint store of the answer to the current constraint store with apply_answer/2, and checks if the resulting constraint store is consistent. If so, execution continues; otherwise the execution continues to step 16.

```
:- active_tclp.

store_projection(V, (F, St)) :-
            clpqr_dump_constraints(V, F, St).

call_entail(V, _, (V, StGen)) :- clpq_entailed(StGen).

answer_compare(V,        _, (V, StAns), =<) :-
            clpq_entailed(StAns), !.
answer_compare(_, (F, St), (F, StAns), >) :-
            clpq_meta(StAns), clpq_entailed(St).

apply_answer(V, (V, St)) :- clpq_meta(St).
```

Figure 7: Mod TCLP interface for CLP(Q), a bridge to existing predicate

3.3 Interface implementation of TCLP(Q)

As an example, Fig. 7 shows the implementation of the TCLP interface for Holzbauer's CLP(Q) solver [11]. We will refer to this code when we explore the step-by-step execution of a program in Section 3.4. The existing CLP(Q) implementation already provides most of the functionality required by the tabling engine, so that the TCLP(Q) interface act mostly as a bridge to existing predicates.

A Mod TCLP constraint interface starts with the declaration :- active_tclp. which makes the compiler check which interface predicates are available in order to adjust the program transformation, and instructs the runtime to activate the TCLP framework when loading the program. The functionality required by the interface is implemented as follows:

Projection the predicate clpqr_dump_constraints(+V, -F, -St), provided by the constraint solver, performs projection. It returns in St the current store projected onto the variables in V. The variables in St are fresh and are contained in the list F, following the same order as those in V.

Call entailment CLP(Q) provides a predicate clpq_entailed(+St) which checks entailment of St w.r.t. the current constraint store, with which it has to share variables. In order to check entailment with a projected constraint store retrieved from the table, the fresh variables in the retrieved store have to be unified with the constrained variables in the call V. That is why store_projection/2 returns a structure containing both the projected store and the variables in this store.

Answer entailment In order to return the entailment direction of the current constraint store w.r.t. a previous answer StAns, we wrap it using two clauses. The first one checks in the same direction as clpq_entailed/1, for which we only need to unify the fresh variables of the retrieved answer with those in the current environment. To check in the other direction we make StAns part of the current store by executing the constraints (which does not interact with the current store, because all of its variables are fresh) and we check entailment with the projected current answer, whose variables have been previously unified with those in the retrieved answer. If there is no entailment in any direction, the call fails.

Apply answer The predicate clpq_meta(+St) executes the constraints in St. If they are consistent with the current constraint store, execution continues, and fails otherwise. The variables in St have been previously unified with those in the pattern of the answer.

With this constraint solver, the predicates current_store/1 and reinstall_store/3 are not needed since the CLP(Q) is implemented in Prolog with attributed variables which restores its past state on backtracking when necessary.

3.4 Step by Step Execution of dist/3

We will now walk through a step by step execution of the query ?- D #< 150, dist(a,Y,D). under the program in Fig. 1 (right) and the graph in Fig. 8, where one edge length is defined with constraints. We will use TCLP(Q) for this example, which is imported using :- use_package(t_clpq).. The directive :- table dist/3. is used to enable tabled execution.

```
edge(a, b, 50).
edge(b, a, D) :-
      D #>= 25,
      D #=< 35.
```

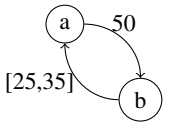

Figure 8: Graph definition. [25,35] is the closed interval from 25 to 35.

The query starts the execution adding the constraint D#=<150 to the current store. After dist(a,Y,D) is called, the tabling engine takes the control of the execution calling tabled_call(dist_aux(a,Y,D). Then lookup_table/3 initializes and saves Gen = dist_aux(a,A,V_1), the first occurrence of the call, and returns Vars=[D]. Then store_projection([D],(F,St)) is executed, which returns (F,St)=([V0],[V0#=<150]).

The list of projected constraint stores associated to dist_aux(a,A,V_1) is empty, so the execution continues in step 5 because current_store/3 is not implemented. The tuple ⟨dist_aux(a,A,V_1), ([V0],[V0#=<150])⟩ is saved as a generator and its execution starts in step 8.

During the evaluation of the first clause of dist_aux/3, the constraints {D1#>0, D2#>0, D#=D1+D2} are added to the constraint store. Then dist(a,Z,D1) is called and the tabling engine reenters the tabled execution with tabled_call(dist_aux(a,Z,D1)). The execution of lookup_table(dist_aux(a,Z,D1),Vars,Gen) returns Vars=[D1] and returns in Gen the previous variant call (dist_aux(a,A,V_1)) because it is a variant of dist_aux(a,Z,D1). At this point the list of projected constraint stores associated to Gen contains the projected constraint store of the previous call.

The constraint solver executes store_projection([D1], ProjStore) and returns ProjStore=([V0], [V0#>0, V0#<150]). For clarification, the projection of the current constraint store {D=<150, D1>0, D2>0, D=D1+D2} onto D1 is {D1>0, D1<150}.

Then the tabling engine retrieves the previously projected store ProjStore_G=([V0], [V0#=<150]) and checks if it entails the current constraint store with clpq_entailed([V0#=<150]),[4] which after the unification D1=V0 succeeds because

$$\{D{=}{<}150,\ D1{>}0,\ D2{>}0,\ D = D1{+}D2\} \sqsubseteq \{D1{=}{<}150\}.$$

The current call dist_aux(a,Z,D1) is suspended and the evaluation of the generator backtracks to continue executing its second clause. This execution starts with the initial constraint store D#=<150 and edge(a,Y,D) which unifies with edge(a,b,50). Then new_answer/0 (added at the end of the body by the compiler) collects the first answer, whose variable substitution is {Y=b, D=50}. This is stored in the table, and since the list of constrained variables is empty, store_projection/2 returns ([],[]) and the answer ⟨{Y=b, D=50}, ([],[])⟩ is saved.

The tabling engine then resumes the suspended goal, which consumes this answer. Since the variable substitution (D1=50, after renaming) is consistent with the constraint store of the consumer {D1>0, D1<150}, the execution continues. Then edge(b,Y,D2), {..., D2#=<99} is called, which unifies with edge(b,a,D2),

[4]Note that TLCP(Q) does not make use of ProjStore since the entailment is checked using the current constraint store.

{D2#>=25, D2#=<35}. The constraint solver adds the constraints onto D2 to the current constraint store and since it is consistent, the second answer is collected by new_answer/0. Then lookup_table/3 stores the variable substitution {Y=a} and returns [D] as the list of constrained variables. store_projection/2 returns ([V0], [V0#>75,V0#<85]) and since the list of projected stored associated to Y=a is empty, the answer ⟨{Y=a}, ([V0],[V0#>75,V0#<85])⟩ is saved.

The tabling engine resumes for the second time the consumer, which consumes the second answer. apply_answer([D1], ([D1],[D1#>75,D1#<85])) succeeds and executions continues. edge(a,Y,D2), {..., D2#=<64} is then called and unifies with edge(a,b,50). and new_answer/0 collects the third answer. Then lookup_table/3 stores the variable substitution {Y=b} and store_projection/2 returns ([V0], [V0#>125,V0#<135]). Since the list of projected stores associated to Y=b is empty, the answer ⟨{Y=b}, ([V0],[V0#>75,V0#<85])⟩ is saved.

For the third time the tabling engine resumes the consumer, which consumes a new answer. apply_answer([D1], ([D1], [D1#>75,D1#<85])) succeeds and the execution continues. Then edge(b,Y,D2), {...,D2#=<15} is called and fails. Since the generator has exhausted all the answers and it does not have any more dependencies, it is marked as complete with complete/0 and it will consume the three collected answers of the query.

3.5 Implementation Improvements

The design we have presented strives for simplicity. There are however several improvements that can be done which enhance this basic design, but which we preferred not to introduce before in order not to make the presentation more cumbersome. We will present them now, with the understanding that they do not change the general ideas we have presented so far.

Two-Step Projection.

Projection (store_projection/2 is usually the most costly operation of the TCLP interface, but it is only mandatory when a call is a generator, which we know with entailment checking. We have however placed it before the entailment phase because the constraint solver can use the projection operation to compute some information needed the check entailment, so that instead of repeating that computation for each previous generator it is done only once, and because the benefits of avoiding the recomputation of a consumer can balance the effort of computing its projection.

In our implementation, projection is performed in two steps: an operation aux_projection(Vars, Aux is executed before the entailment phase which returns in Aux the information needed in to check entailment, and an operation final_projection(Vars, Aux, ProjStore) is executed after the entailment checking when it fails (which means that the call is a generator), which returns the projected constraint store in ProjStore using the information in Aux.

For symmetry, a similar mechanism is used with answers: aux_ans_projection/2 is performed before the entailment check for answers and final_ans_projection/3 after the answer entailment check.

The TCLP(Q) interface in Fig. 7 takes advantage of this improvement in the call entailment phase because clpq_entail/1 does not need any information from the current constraint store to evaluate the entailment. Therefore when a call is entailed by a previous generator, the projection is not needed. However, in the answer comparison phase the projected constraint store is needed because entailment can be checked in both directions. The TCLP(Q) implementation of the projection with these improvements is:

```
aux_projection(_,_).
final_projection(V,_,(F,St)) :- clpqr_dump_constraints(V,F,St).

aux_ans_projection(V,(F,St)) :- clpqr_dump_constraints(V,F,St).
final_ans_projection(_,(F,St),(F,St)).
```

Partial Projections.

The Mod TCLP framework allows to define a constraint solver interface that instead of the projection of a constraint store executes a partial projection. This feature is helpful for constraint domains which have no projection operation (or weak projection only) [17] and for those that do but have a prohibitive cost.

A partial projection of a constraint store CSt onto a set of variables V, $V \subseteq vars(CSt)$ is a constraint store PSt with $vars(PSt) \subseteq V$ and where some of the constraints in the full projection are omitted. As a result, CSt is entailed by PSt ($CSt \sqsubseteq PSt$), so any solution of CSt is also a solution of PSt, but not every valuations over V which is a solution in PSt is also a partial solution of CSt.

In the Mod TCLP framework, a partial projection can be used to identify the generator $\langle C_{gen}, PartialSt_{gen}\rangle$. Answer entailment can also be checked using the partial projection $PartialSt_{ans}$ of the constraint store $Store_{ans}$ of the new call, because if $Store_{ans} \sqsubseteq PartialSt_{ans}$ and if it is the case that $PartialSt_{ans} \sqsubseteq PartialSt_{gen}$ it is also true that $Store_{ans} \sqsubseteq PartialSt_{gen}$.

But it is important to note that to guarantee completeness the generator should be executed with the partial projected constraint store $PartialSt_{gen}$ as the current store (without any other constraint). Because the fact that $Store_{gen} \sqsubseteq PartialSt_{gen}$ and $Store_{ans} \sqsubseteq PartialSt_{gen}$ does not always imply that $Store_{ans} \sqsubseteq Store_{gen}$. This execution of the generator will produce all the answers valid for the consumers, thereby ensuring completeness. However it may also produce some answers not consistent with the initial constraint store $Store_{gen}$ of the generator call, which would lead to incorrect solutions. To guarantee correctness, the constraint solver should be sound and fail when it is requested to add a constraint (e.g., one of the answers) inconsistent with $Store_{gen}$.

The predicates current_store/3 and reinstall_store/2, which could be provided by external constraint solvers, can be used to guarantee the correct use of partial projections. With current_store(+Vars, +ProjStore, -CuSt) the constraint solver updates the constraint store of the generator to be ProjStore and returns in CuSt the information needed to recover the omitted constraints later on. With reinstall_store(+Vars, +CuSt), the constraint solver restores the state the constraint store had before the generator started to consume its answers.

Call abstraction [20] is an extreme case of partial projection, where the constraint store associated to the tabled call is not taken into account to execute it, losing benefits of the constraints since it has to compute all the possible results and then filter them through the call-time constraint store.

Improve Answer Checking with Aggregations.

answer_compare/4 is used to discard redundant answers and, as a result, it may also reduce how many times consumers are resumed. A variant of answer_compare/4 can be used to modulate what Mod TCLP does when a new answer is received: when two answers are compared, they can be substituted for a new one. This feature can be made available with a new call answer_check(+Vars, +ProjStore, +$ProjStore_{ans}$, -Res, -$ProjStore_{new}$) that can return new in Res and, in that case, the engine removes $ProjStore_{ans}$ and saves $ProjStore_{new}$ instead of ProjStore.

This can be used for a variety of purposes. For example, two answers A_1 and A_2 which are points in a lattice can be *combined*

and $A_1 \sqcup A_2$ saved instead. This can be used to store abstractions of answers, which will lose some information, but which may be acceptable in some applications. The predicate which decides how the answers should be combined is provided by the programmer in the TCLP interface, and whether the results this gives are logically consistent depends ultimately on the programmer.

4. INTEGRATION AND EXPERIMENTAL EVALUATION

The design we are presenting is likely to bring more flexibility to a system with tabled constraints at a reasonably price in implementation effort. We will support this claim with several examples. As it is usual with these cases, additional flexibility comes also with a performance price, which we will also evaluate. To perform this double validation, we have used four solvers with different characteristics in their implementation: a constraint solver for difference constraints (Section 4.1), ported from [2] and completely written in C; the well-known attribute-based implementations of CLP(Q) and CLP(R) [11], used in many Prolog systems (Section 4.2); and a new solver for a constraint system over finite lattices (Section 4.3).

These use cases and a full Ciao Prolog distribution, including the libraries and interface presented in this paper, are available at http://goo.gl/vWRV15. All experiments were done using that distribution on a Mac OS-X 10.9.5 machine with a 2,66 GHz Intel Core 2 Duo processor. The run time results are given in milliseconds (ms).

4.1 Difference Constraints

Difference constraints CLP($D_{\leq}$) is a simple but relatively powerful constraint system whose constraints are of the form $X - Y \leq d$ where $X, Y, d \in \mathbb{Z}$, X, Y are variables, and d is a constant.

A system of difference constraints can modeled with a weighted graph which is satisfiable if there are no cycles with negative weight. A solver for this constraint system can be based on shortest path algorithms [6] where the constraint store is represented as an $n \times n$ matrix A of distances. The implementation uses attributed variables to relate Prolog variables with their representation in the matrix by having as attribute its index in the matrix.

The projection onto a set of variables V extracts a sub-matrix A' containing all pairs (v_1, v_2) s.t. v_1, $v_2 \in V$. To speed up the projection, it is represented as a vector of length |V| containing the index of each v_i in A. For example, if the indexes in A of the variables [X,Y,Z,T,W] are [1,2,3,4,5], the projection onto the set of variables [T,X,Y] is represented with the vector (4,1,2).

The TCLP($D_{\leq}$) interface uses the triple (Index, Size, Store) to identify the projected constraint store. Index is the memory address of the vector with the indexes of the constrained variables of the call/answer, Size is the length/number of constrained variables, and Store is the memory address of a copy of the sub-matrix which represents the projected constraint store. This copy is only needed by the generators in the entailment phase where the constraint solver calls call_entail(_, (I, S, _), (_, _, St)):-... to check entailment. At this point, the projected constraint store of the previous generator St points to a copy of the submatrix because the main matrix had presumably been modified during the execution and the indexes of variables of the generator may correspond to other variables.

Since the copy of the sub-matrix is only needed for the generators, it is possible to increase performance and reduce memory footprint using the projection in two steps described in section 3.5 and copying the sub-matrix only when the entailment phase fails.

The solver does not use the memory areas of the tabling engine, and the Prolog machinery cannot restore the current constraint stores of the generators on backtracking. Therefore the constraint solver uses current_store/3 to return a reference to a copy of the current constraint store (the matrix A) and reinstall_store/2 to make a previously saved copy be the current constraint store.

	CLP($D_{\leq}$)	Orig. TCLP($D_{\leq}$)	Mod TCLP($D_{\leq}$)
truckload(300)	40452	2903	7268
truckload(200)	4179	1015	2239
truckload(100)	145	140	259
step_bound(30)	-	2657	1469
step_bound(20)	-	2170	1267
step_bound(10)	-	917	845

Table 2: Run time (ms) using the difference constraint solver for truckload/4 and step_bound/1.

Performance Evaluation.

The original TCLP implementation [2] was done on the same platform as ours and we are reusing the same solver. Since that implementation was deeply intertwined with the tabling engine and had a very low overhead, we will use it to compare the costs of adopting a more modular framework against the original implementation, we use two benchmarks:

truckload(P, Load, Dest, Time) (from [20]): it solves a shipment problem with a call. We use P = 30, Dest = chicago, using Load as parameter to vary its complexity. truckload/4 does not need tabling, but it speeds up if tabling is used.

step_bound(Init, Dest, Steps, Limit) : a left-recursive graph reachability program similar to dist/3 which constrains the total number (Limit) of edge traversals. step_bound/4 needs tabling in the case of graphs with cycles.

Table 2 shows that truckload/4 incurs a nearly three-fold increase in execution time with respect to the initial non-modular TCLP($D_{\leq}$) implementation. This is mainly due to the overhead of the execution control, which in our current implementation goes from the tabling engine (in C) level to the interface level (in Prolog), which calls back the constraint solver (in C). In the initial implementation, communication did not move between levels, and was directly done from C to C level (which made it nearly impossible to plug other constraint solvers).

step_bound/4 is however less efficient in the initial TCLP($D_{\leq}$) implementation than in the current one (and cannot be executed in CLP(Q) as the graph has cycles). The reason is that our modular design made it possible to implement more easily non-trivial operations such as discarding more particular answers already stored in the table (enabled by the $=<$ and $>$ results returned by answer_compare/4), and step_bound/1 takes advantage of this optimization, which is not very significative in the case of truckload/4.

To validate this optimization we executed the programs truckload/4 and step_bound/1 using three different answer management strategies:

- $\emptyset$ answer_compare/4 does not check answer entailment.
- $\leftarrow$ answer_compare/4 only checks if new answers are entailed by previous answers. If so, the new answer is discarded. This was already implemented in the original TLCP implementation.
- $\rightarrow$ answer_compare/4 only checks if new answers entail previous answers. If so, the previous answer is flagged as removed.
- $\leftrightarrow$ answer_compare/4 checks entailment in both directions.

The results in Table 3 confirm that in the examples studied, and despite the cost of these strategies, the number of answers returned and the computation time is reduced. In truckload/4 the number of answers does not change drastically; although in can vary in one

	∅		←		→		↔	
	Time	Ans.	Time	Ans.	Time	Ans.	Time	Ans.
truckload(300)	742039	14999	7806	41	7780	30	7268	5
truckload(200)	11785	1520	2314	23	2354	18	2239	5
truckload(100)	300	58	263	6	263	9	259	3
step_bound(30)	–	–	8450	252	–	–	1469	25
step_bound(20)	–	–	6859	242	38107	441	1267	25
step_bound(10)	–	–	2846	165	8879	221	845	25

Table 3: Run time (ms) and total # answers returned. '∅', '←', '→' and '↔' define the answer management strategy.

∅	# Sav.	# Dis.	# Rem.	# Ret.
truckload(300)	448538	0	0	14999
truckload(200)	52349	0	0	1520
truckload(100)	2464	0	0	58
←	# Sav.	# Dis.	# Rem.	# Ret.
truckload(300)	67503	9971	0	41
truckload(200)	16456	1325	0	23
truckload(100)	1525	52	0	6
step_bound(30)	44549	716826	0	252
step_bound(20)	37548	599259	0	242
step_bound(10)	15625	242351	0	165
→	# Sav.	# Dis.	# Rem.	# Ret.
truckload(300)	75272	0	9460	30
truckload(200)	17568	0	1298	18
truckload(100)	1490	0	49	9
step_bound(30)	>1145690	0	>1074071	–
step_bound(20)	946309	0	891078	441
step_bound(10)	294728	0	276867	221
↔	# Sav.	# Dis.	# Rem.	# Ret.
truckload(300)	48524	6596	1740	5
truckload(200)	13550	1046	240	5
truckload(100)	1343	45	10	3
step_bound(30)	9697	74528	4571	25
step_bound(20)	9352	71658	4371	25
step_bound(10)	6650	56935	3019	25

Table 4: # Answers: saved (Sav.), discarded (Dis.) , removed (Rem.) and returned to the query (Ret.)

order of magnitude, the number of returned solutions is not very large, which means that the impact in execution time is not very important. It is notwithstanding interesting to note that there is no slowdown when using the more complex strategy.

The case of step_bound/4 is interesting: the worst strategy is "→" (up to the point that the execution runs out of memory for the largest case), followed by "←", and then by "↔", which is the best one, both in terms of execution time and number of solutions.

The internal reasons of these differences can be deduced from Table 4, where, for each of the benchmark cases and strategies, we show how many of the generated answers are saved, discarded before being inserted, or removed after being inserted. Note that the number of generated answers is not always the same since, as discussed before, fewer saved answers wake up fewer consumers. It is clear that the "↔" strategy has the best profile, while "→" generates many more candidate answers than either of the other two — in excess on one million for step_bound(30). Not using any entailment strategy ("∅") is clearly a very bad option.

As an example of the usefulness of obtaining the most general correct answer, Fig. 9 shows a "shortest distance" program sd/3 [3]. For a query such as ?- sd(a, c, Dist). the system reported in [3] returns as answers a series of constraints Dist #>= C_k, one of which reflects the smallest distance between a and c, but we do not know in advance which one.

On the other hand, the ↔ strategy only returns one answer (the most general one) which corresponds to the tightest bound for the shortest distance between two nodes. Therefore, we have the following pairs query ⇝ answer: ?- sd(a,c,D). ⇝ D #>= 3.0, ?- sd(a,b,D). ⇝ D #>= 2.0, and ?- sd(d,a,D). ⇝ D #>= 2.3.

	LP	Tab	CLP	TCLP	
Right rec.	1917	291	200	184	Without
Left rec.	–	144	–	45	cycles
Right rec.	–	–	4261	1027	With
Left rec.	–	–	–	420	cycles

Table 5: Run time (ms) for dist/3. A '–' means no termination.

4.2 CLP(Q) and CLP(R)

Holzbauer's CLP(Q) and CLP(R) [11] is a very well known Prolog extension for Constraint Logic Programming over the rationals and over the reals. The TCLP interface for CLP(R) is similar to the interface of CLP(Q) presented in Fig. 7. The only difference is that the CLP(Q) predicates clpq_entailed/1 and clpq_meta/1 have to be replaced by their CLP(R) counterparts clpr_entailed/1 and clpr_meta/1. We will therefore evaluate the behavior of the CLP(R) and CLP(Q) plugins.

Performance Evaluation.

We will first use the CLP program in Fig. 1, with which we evaluated completeness in Table 1, to evaluate the performance in the same cases. We will use a graph of 25 nodes without cycles (with 584 edges) or with cycles (with 785 edges). Table 5 confirms that TCLP(Q), besides being terminating all cases, as we already shown, it is also the faster implementation. It also shows, in line with the common experience in tabling, that left-recursive implementations are usually faster and preferable for tabled systems.

We have also used the doubly-recursive Fibonacci program -? fib(N, F). Tabling makes forward evaluation of this program be linear in time, instead of exponential. We will however run it backwards — finding the index of some Fibonacci number using exactly the sme code. When executed with CLP in a non-tabled systems, this program is also exponential: it deploys a system of equations which can be solved when there are enough equations to reach to cover the doubly recursive execution from the base case to the required Fibonacci number. However, tabling with entailment can make backwards Fibonacci run in linear time, which we confirm experimentally.

```
:- table sd/3.

sd(X,Y,D) :-
      edge(X,Y,D0),
      D #>= D0.
sd(X,Y,D) :-
      sd(X,Z,D1),
      edge(Z,Y,D2),
      D #>= D1+D2.
```

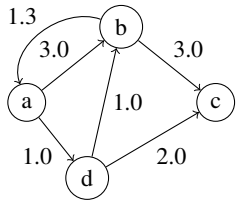

Figure 9: sd/3 – "shortest distance" program [3].

	version A	version B	
	TCLP($D_{\leq}$)	TCLP(Q)	TCLP(R)
fib(P, 832040)	147	58	26
fib(P, 28657)	68	37	16
fib(P, 610)	25	18	8
fib(P, 89)	13	12	5

Table 6: Run time (ms) for the fib/2: version A with difference constraint and version B with CLP(Q) and CLP(R).

```
:                              :
fib(N1, F1),                   F #= F1 + F2,
fib(N2, F2),                   fib(N1, F1),
F #= F1 + F2.                  fib(N2, F2).
```

Figure 10: Two versions of fib/2: TCLP($D_{\leq}$) (left) vs. TCLP(Q) and TCLP(R) (right).

We use TCLP($D_{\leq}$), TCLP(Q), and TCLP(R). In the second and third case the constraint F #= F1 + F2 appears as early as possible (Fig. 10, right). However, for TCLP($D_{\leq}$) we cannot have three variables in the constraint, and therefore we need to move the expression to the end of the body (Fig. 10, left). This can be detrimental to the performance of TCLP($D_{\leq}$), as value propagation in the constraints has less effect. The results are in Table 6, where CLP(R) is the faster version. Although the solver is basically the same as that for CLP(Q), it obtains advantage because operations with floating point numbers are directly done at the CPU level, instead of being done by software as it the case for the rationals. However, there is a drawback: floating point arithmetic is not precise, and when CLP(R) approximates its results, it can raise (depending on the particular program) non-termination results. That is the case of the usual version of Fibonacci, where queries such as ?- fibo(K, 23416728348467685) terminate correctly with CLP(Q), but do not return (in under five minutes) with CLP(R) since lack of accuracy makes a termination condition not to hold.

This example also highlights that the modularity of TCLP makes it possible to choose the best constraint solver for the problem at hand, and that decision should not always be based on the performance of the constraint solver, but on its adequacy.

4.3 Constraints over Finite Lattices

A lattice is a triplet $(\mathbb{S}, \sqcup, \sqcap)$ where $\mathbb{S}$ is a set of points and join ($\sqcup$) and meet ($\sqcap$) are two internal operations that follow the commutative, associative and absorption laws. $(\mathbb{S}, \sqsubseteq)$ is a poset where $\forall a,b \in \mathbb{S}\,.\, a \sqsubseteq b$ if $a = a \sqcap b$ or $b = a \sqcup b$ and $\exists \bot, \top \in \mathbb{S}$ such as $\forall\, a \in \mathbb{S}\,.\, \bot \sqsubseteq a \sqsubseteq \top$.

Constraints over finite lattices CLP(L) where the values are elements in a finite lattice. In this example we use this lattice as the domain of an abstract interpreter, and therefore we assume the existence of abstract operations between the element of the lattice which are counterpart of the operations in the concrete domain. Therefore, the constraints that we deal with in this constraint system are, on one hand, those which come from the topological relationship of the lattice elements and, on the other hand, those which are induced by the operations in the abstract domain.

The lattice layer provides the constraint $Y \sqsubseteq X$ with $X,Y \in \mathbb{S}$ and the projection operation which, is performed using Fourier's algorithm [17] for variable elimination: the projection of $\{X \sqsubseteq d,\ Y \sqsubseteq X\}$ onto $\{Y\}$ is $\{Y \sqsubseteq d\}$). This layer also provides entailment checking and the operator to apply the answer-projected constraints.

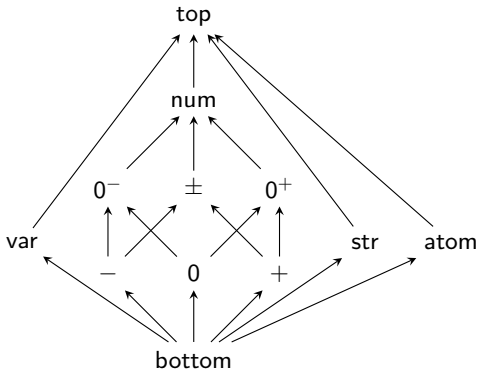

Figure 11: *Signs* abstract domain.

The domain layer should define (at least) the operators $\sqcup$ and $\sqcap$, which are used by the lattice layer to define a partial order. The domain layer can make use of an interface provided by the lattice layer to define the operations among the points of the lattice. The operations of the domain execute, upon changes in the domain of the variables, the propagator of the lattice layer. This propagator re-executes previous domain operations which involve these variable(s) until no changes in the domain of the variables happen.

Performance Evaluation.

To evaluate the use of CLP(L), we use two versions of a simple abstract interpreter. This interpreter executes logic programs using an abstract domain as values for the program variables. The abstract domain we have used by in this example is the *signs* abstract domain shown in Fig. 11. The results of the abstract interpreter are a safe approximation of the runtime values of the variables in the concrete program. We use two versions of the interpreter:

Tabling This version uses tabling, which guarantees termination since the abstract domain is finite.

TCLP This version uses the CLP(L) constraint solver to carry the abstract domain and the constraints over the variables. The main difference with the Tabling version is that the constraint solver automatically executes the propagator upon changes in the domains of the variables and solves the equations posed by the program.

We used two programs as input to our abstract interpreter:

takeuchi/m is a Prolog implementation of the n-dimensional generalization of Takeuchi recurrence algorithm [16]. Fig. 12 shows its mathematical definition and its Prolog implementation. The program is parameterized by the number of input arguments n, and it returns in its last argument the function result.

sentinel/m (Fig. 13) is a variant of a synthetic program presented in [8]. It receives as input its first argument (the *Sentinel*) and the next n arguments $A_1, \ldots A_n$ are a ring-ordered[5] series of numbers. The output are arguments $B_1,\ldots,B_n$, which correspond to a circular shift of $A_1,\ldots,A_n$ such that on success $B_i < B_{i+1}$ for all $i < n$ and: (i) if $Sentinel = 0$, the first half of B_i are negative and the second half are positive; (ii) if $Sentinel < 0$, $B_i < Sentinel$ for all i; and (iii) if $Sentinel > 0$, $B_i > Sentinel$ for all i.

Table 7 shows the results of the execution times for takeuchi/m parametrized by the dimension of the function, n ($m = n+1$), and sentinel/m parametrized by n, the length of the ring ($m = 2n+1$).

[5]I.e., there is a j such that $A_j < A_{j+1}, A_{j+1} < A_{j+2}, \ldots, A_n < A_1, A_1 < A_2, \ldots, A_{j-2} < A_{j-1}$.

$$t(x_1,x_2,\ldots,x_n) = \text{if } x_1 \leq x_2 \text{ then } x_2 \text{ else } t(t(x_1-1,x_2,\ldots,x_n),\ldots,t(x_n-1,x_1,\ldots,x_{n-1}))$$

```
takeuchi(X1, X2,...Xn, R) :- X1 < X2, R = X2.
takeuchi(X1, X2,...Xn, R) :- X1 =:= X2, R = X2.
takeuchi(X1, X2,...Xn, R) :- X1 > X2,
        N1 is X1 - 1, takeuchi(N1, X2,...Xn, R1),
        N2 is X2 - 1, takeuchi(N2, X3,...X1, R2),
        ...,
        Nn is Xn - 1, takeuchi(Nn, X1,...Xn-1, Rn),
        takeuchi(R1, R2, ...Rn, R).
```

Figure 12: n-dimensional Takeuchi function and its Prolog implementation.

```
sentinel(Sentinel , A1,...An, B1,...Bn) :- Sentinel =:= 0,
        ring(A1,...An, B1,...Bn),
        B1 < B2, ... Bn-1 < Bn, Bn/2 < Sentinel, Sentinel < Bn/2+1.
sentinel(Sentinel , A1,...An, B1,...Bn) :- Sentinel < 0,
        ring(A1,...An, B1,...Bn),
        B1 < B2, ... Bn-1 < Bn, Bn < Sentinel.
sentinel(Sentinel , A1,...An, B1,...Bn) :- Sentinel > 0,
        ring(A1,...An, B1,...Bn),
        B1 < B2, ... Bn-1 < Bn, B1 > Sentinel.

ring(A1,...An, B1,...Bn) :- B1 = A1, ... Bn = An.
ring(A1,...An, B1,...Bn) :- A1 > A2,
        ring(A2,...An, A1, B1,...Bn).
ring(A1,...An, B1,...Bn) :-
        ring(An, A1,...An-1, B1,...Bn).
```

Figure 13: sentinel/m program.

In both examples the TCLP version of the interpreter is faster than the tabled interpreter without constraints: the latter has to evaluate each permutation completely in the recursive predicates, while the former can suspend and save computation time using results from a previous, more general, call.

Let us examine an example. For a variable A the representation A^{abs} means that $A \sqsubseteq abs$. On one hand in the TCLP interpreter when an initial goal ring($A_1^{top}, \ldots A_n^{top}, B_1^{top}, \ldots B_n^{top}$) is interpreted, the first clause of ring/2n produces the first answer. Then the interpreter continues with the second clause, interprets the goal $A_1 > A_2$ and starts the evaluation of ring($A_2^{num} \ldots A_n^{top}, A_1^{num}, B_1^{top}, \ldots B_n^{top}$). Since num $\sqsubseteq$ top, this new call is entailed by the previous one and TCLP suspends this execution. Then the interpreter continues with the third clause, which starts the evaluation of ring($A_n^{top}, A_1^{top}, \ldots A_{n-1}1^{top}, B_1^{top}, \ldots B_n^{top}$), and TCLP also suspends the execution. Since the generator has no more clauses to evaluate, TCLP resumes the suspended execution with the previously obtained answer. Each consumer produces a new answer but since they are at least as particular as the previous one, they are discarded.

On the other hand in the tabling interpreter when the initial goal is evaluated with $A_1^{top},\ldots,A_n^{top},B_1^{top},\ldots,B_n^{top}$ as entry substitution, also the first answer is produced. Then the interpreter continues with the second clause, interprets the goal $A_1 > A_2$ and starts the evaluation of the recursive call with the entry substitution $A_1^{num},\ldots,A_n^{top},B_1^{num},\ldots,B_n^{top}$. However tabling does not suspend the execution because it is not a *variant* call of the previous one, which results in an increase of the computation time.

Table 8 shows the results of interpreting sentinel/m in two different scenarios: without any restrictions in the abstract substitution of the variables ($Sentinel^{top}, A_i^{top}, B_i^{top}$) or adding the restriction Sentinel$^+$ before the analysis. When the restriction is added before the analysis (as an entry substitution), execution time is reduced in approximately the same relative amount in both cases.

	takeuchi/m ($m = n+1$)			sentinel/m ($m = 2n+1$)		
	n=3	n=6	n=8	n=4	n=6	n=8
Tabling	2.42	13.75	31.44	30.99	218.93	1375.13
TCLP	3.12	5.85	8.09	4.56	6.53	9.23

Table 7: Run time (ms) for analyze(takeuchi/m) and for analyze(sentinel/m)

		sentinel/m ($m = 2n+1$)		
		n=4	n=6	n=8
Tabling	without restrictions	30.99	218.93	1375.13
	restrictions before	13.39	98.80	749.38
TCLP	without restrictions	4.56	6.53	9.23
	restrictions before	2.85	3.31	5.29

Table 8: Run time (ms) for analyze(takeuchi/m) and for analyze(sentinel/m)

If the restriction is added after the analysis, there is no significant change in the execution time of the tabled interpreter (not shown in the table). In the case of the tabled, non-constraint interpreter, this restriction is not propagated to the rest of the variables. However, in the case of the constraint-based interpreter, this additional restriction can be propagated after the interpreter has finished, giving more precision to the result.

5. RELATED WORK

The initial ideas of tabling and constraints originate in [15], where a variant of Datalog featuring constraints was proposed. The time and space problems associated with the bottom-up evaluation of Datalog were studied in [25], where a top-down evaluation strategy featuring tabling was proposed.

XSB [22] was the first logic programming system which provided tabled CLP as a generic feature, instead of resorting to ad-hoc adaptations. This was done by extending XSB with attributed variables [3], one of the most popular mechanism to implement constraint solvers in Prolog. However, one of its drawbacks is that it only uses variant call checking (including for goals with constraints), instead of entailment checking of calls / answers. This makes programs terminate in fewer cases and take longer in other cases. This is similar to what happens in tabled logic programs with and without subsumption [21].

A general framework for CHR under tabled evaluation is described in [20]. This approach brings the flexibility that CHR provides for writing constraint solvers, but it also lacks call entailment checking and enforces total call abstraction: all constraints are removed from calls before executing them, which can result in non-termination w.r.t. systems which use entailment. Besides, the need to change the representation between CHR and Herbrand terms takes a toll in performance.

Failure Tabled CLP [7] implements a verification-oriented system which has several points in common with TCLP. Interestingly, it can learn from failed derivations and uses interpolants instead of constraint projection to generate conditions for reuse. It will however not terminate in some cases even with the addition of counter to implement a mechanism akin to iterative deepening.

Last, the original TCLP proposal [2] features entailment checking for calls and (partially) for answers, executes calls with all the constraints, and has good performance. However, it did not clearly state which operations must be present in the constraint solver, which made it difficult to extend, and is not focused on a modular design which, for example, made it difficult to implement specific answer management strategies.

6. CONCLUSIONS

We have presented an approach to include constraint solvers in logic programming systems with tabling. Our main goal is to make adding additional constraint solvers easier. In order to achieve this, we determined the services that a constraint solver should provide to a tabling engine. The constraint solver has freedom to implement them as it wishes, and have been designed to cover many possible implementations. To validate our design we have interfaced one solver previously written in C (difference constraints), two existing, classical solvers (CLP(Q/R)), and a new solver (constraints over finite lattices), and we have found the integration to be easy — certainly easier than with other designs, given the capabilities that our system provides.

We evaluate its performance in a series of benchmarks. In some of them large savings are attained w.r.t. non-tabled/tabled executions — even taking into account the penalty to pay for the additional flexibility and modularity. We are in any case confident that there is still ample space to improve the efficiency of the implementation, since we are presenting an initial prototype in which we internally gave more importance to the cleanliness of the code and the design.

7. REFERENCES

[1] Witold Charatonik, Supratik Mukhopadhyay, and Andreas Podelski. Constraint-based infinite model checking and tabulation for stratified clp. In Peter J. Stuckey, editor, *ICLP*, volume 2401 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2002.

[2] P. Chico de Guzmán, M. Carro, M. Hermenegildo, and P. Stuckey. A General Implementation Framework for Tabled CLP. In Tom Schrijvers and Peter Thiemann, editors, *FLOPS'12*, number 7294 in LNCS, pages 104–119. Springer Verlag, May 2012.

[3] Baoqiu Cui and David Scott Warren. A System for Tabled Constraint Logic Programming. In *Computational Logic*, pages 478–492, 2000.

[4] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems – A Case Study. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, pages 117–126, New York, USA, 1996. ACM Press.

[5] Juliana Freire, Terrance Swift, and David Scott Warren. Beyond Depth-First Strategies: Improving Tabled Logic Programs through Alternative Scheduling. *Journal of Functional and Logic Programming*, 1998(3), 1998.

[6] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully Dynamic Shortest Paths and Negative Cycles Detection on Digraphs with Arbitrary Arc Weights. In *ESA*, pages 320–331, 1998.

[7] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Failure tabled constraint logic programming by interpolation. *TPLP*, 13(4-5):593–607, 2013.

[8] Samir Genaim, Michael Codish, and Jacob Howe. Worst-Case Groundness Analysis Using Definite Boolean Functions. *TPLP*, 1(05):611–615, 2001.

[9] OWL Working Group. Owl web ontology language guide. http://www.w3.org/TR/owl-guide/. Retrieved on 23 Sep. 2015.

[10] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *TPLP*, 12(1–2):219–252, 2012. http://arxiv.org/abs/1102.5497.

[11] C. Holzbaur. OFAI clp(q,r) manual, edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.

[12] J. Jaffar and M.J. Maher. Constraint LP: A Survey. *JLP*, 19/20:503–581, 1994.

[13] G. Janssens and K. Sagonas. On the Use of Tabling for Abstract Interpretation: An Experiment with Abstract Equation Systems. In *TPD*, April 1998.

[14] Tadashi Kanamori and Tadashi Kawamura. Abstract Interpretation Based on OLDT Resolution. *Journal of Logic Programming*, 15:1–30, 1993.

[15] Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint Query Languages. *J. Comput. Syst. Sci.*, 51(1):26–52, 1995.

[16] Donald E Knuth. Textbook examples of recursion. *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 207–230, 1991.

[17] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998.

[18] Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, and D.S. Warren. Efficient Model Checking Using Tabled Resolution. In *CAV*, volume 1254 of *LNCS*, pages 143–154. Springer Verlag, 1997.

[19] I.V. Ramakrishnan, P. Rao, K.F. Sagonas, T. Swift, and D.S. Warren. Efficient Tabling Mechanisms for Logic Programs. In *ICLP*, pages 697–711, 1995.

[20] Tom Schrijvers, Bart Demoen, and David Scott Warren. TCHR: a Framework for Tabled CLP. *TPLP*, 8(4):491–526, 2008.

[21] Terrance Swift and David Scott Warren. Tabling with answer subsumption: Implementation, applications and performance. In Tomi Janhunen and Ilkka Niemelä, editors, *JELIA*, volume 6341 of *Lecture Notes in Computer Science*, pages 300–312. Springer, 2010.

[22] Terrance Swift and David Scott Warren. XSB: Extending Prolog with Tabled Logic Programming. *TPLP*, 12(1-2):157–187, 2012.

[23] H. Tamaki and M. Sato. OLD Resol. with Tabulation. In *ICLP*, pages 84–98. LNCS, 1986.

[24] David Toman. Constraint Databases and Program Analysis Using Abstract Interpretation. In *CDTA*, volume 1191 of *LNCS*, pages 246–262, 1997.

[25] David Toman. Memoing Evaluation for Constraint Extensions of Datalog. *Constraints*, 2(3/4):337–359, 1997.

[26] D. S. Warren. Memoing for Logic Programs. *CACM*, 35(3):93–111, 1992.

[27] R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *JICSLP*, pages 684–699. MIT Press, August 1988.

[28] Neng-Fa Zhou, Yoshitaka Kameya, and Taisuke Sato. Mode-directed tabling for dynamic programming, machine learning, and constraint solving. In *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, volume 2, pages 213–218. IEEE, 2010.

[29] Youyong Zou, Tim Finin, and Harry Chen. F-OWL: An Inference Engine for Semantic Web. In *Formal Approaches to Agent-Based Systems*, volume 3228 of *Lecture Notes in Computer Science*, pages 238–248. Springer Verlag, January 2005.